Statistics for Computational Economics

Thomas J. Sargent and John Stachurski

Jan 31, 2024

CONTENTS

This website presents a set of lectures on statistics for computational economics.

Part I

Elementary Statistics

CHAPTER

ONE

PANDAS FOR PANEL DATA

Contents

1.1 Overview

In an earlier lecture on pandas, we looked at working with simple data sets.

Econometricians often need to work with more complex data sets, such as panels.

Common tasks include

- Importing data, cleaning it and reshaping it across several axes.
- Selecting a time series or cross-section from a panel.
- Grouping and summarizing data.

`pandas` (derived from 'panel' and 'data') contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a CSV file and reshaping the resulting `DataFrame` with `pivot_table` to build a `MultiIndex`.

Additional detail will be added to our `DataFrame` using pandas' `merge` function, and data will be summarized with the `groupby` function.

1.2 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`.

The dataset can be accessed with the following link:

```
url1 = 'https://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
 ↪static/lecture_specific/pandas_panel/realwage.csv'
```

```
import pandas as pd

# Display 6 columns for viewing purposes
pd.set_option('display.max_columns', 6)

# Reduce decimal points to 2
pd.options.display.float_format = '{:,.2f}'.format

realwage = pd.read_csv(url1)
```

Let's have a look at what we've got to work with

```
realwage.head()  # Show first 5 rows
```

```
   Unnamed: 0        Time  Country                                    Series  \
0           0  2006-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
1           1  2007-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
2           2  2008-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
3           3  2009-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
4           4  2010-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs

  Pay period     value
0     Annual 17,132.44
1     Annual 18,100.92
2     Annual 17,747.41
3     Annual 18,580.14
4     Annual 18,755.83
```

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data.

We will use `pivot_table` to create a wide format panel, with a `MultiIndex` to handle higher dimensional data.

`pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a `MultiIndex` in our column axis

```
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

```
Country                                    Australia         \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                    Annual Hourly
Time
```

(continues on next page)

(continued from previous page)

```
2006-01-01                                  20,410.65  10.33
2007-01-01                                  21,087.57  10.67
2008-01-01                                  20,718.24  10.48
2009-01-01                                  20,984.77  10.62
2010-01-01                                  20,879.33  10.57

Country                                                                ...  \
Series     In 2015 constant prices at 2015 USD exchange rates          ...
Pay period                                                  Annual     ...
Time                                                                   ...
2006-01-01                                                 23,826.64   ...
2007-01-01                                                 24,616.84   ...
2008-01-01                                                 24,185.70   ...
2009-01-01                                                 24,496.84   ...
2010-01-01                                                 24,373.76   ...

Country                                 United States  \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                     Hourly
Time
2006-01-01                                      6.05
2007-01-01                                      6.24
2008-01-01                                      6.78
2009-01-01                                      7.58
2010-01-01                                      7.88

Country
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                                  Annual Hourly
Time
2006-01-01                                                 12,594.40   6.05
2007-01-01                                                 12,974.40   6.24
2008-01-01                                                 14,097.56   6.78
2009-01-01                                                 15,756.42   7.58
2010-01-01                                                 16,391.31   7.88

[5 rows x 128 columns]
```

To more easily filter our time series data, later on, we will convert the index into a `DateTimeIndex`

```
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

```
pandas.core.indexes.datetimes.DatetimeIndex
```

The columns contain multiple levels of indexing, known as a `MultiIndex`, with levels being ordered hierarchically (Country > Series > Pay period).

A `MultiIndex` is the simplest and most flexible way to manage panel data in pandas

```
type(realwage.columns)
```

```
pandas.core.indexes.multi.MultiIndex
```

```
realwage.columns.names
```

```
FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```
realwage['United States'].head()
```

```
Series     In 2015 constant prices at 2015 USD PPPs         \
Pay period                                    Annual Hourly
Time
2006-01-01                                 12,594.40   6.05
2007-01-01                                 12,974.40   6.24
2008-01-01                                 14,097.56   6.78
2009-01-01                                 15,756.42   7.58
2010-01-01                                 16,391.31   7.88

Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                              Annual Hourly
Time
2006-01-01                                           12,594.40   6.05
2007-01-01                                           12,974.40   6.24
2008-01-01                                           14,097.56   6.78
2009-01-01                                           15,756.42   7.58
2010-01-01                                           16,391.31   7.88
```

Stacking and unstacking levels of the `MultiIndex` will be used throughout this lecture to reshape our dataframe into a format we need.

`.stack()` rotates the lowest level of the column `MultiIndex` to the row index (`.unstack()` works in the opposite direction - try it out)

```
realwage.stack().head()
```

```
Country                                                 Australia  \
Series                 In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                       20,410.65
           Hourly                                           10.33
2007-01-01 Annual                                       21,087.57
           Hourly                                           10.67
2008-01-01 Annual                                       20,718.24

Country                                                                    \
Series                 In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                 23,826.64
           Hourly                                                     12.06
2007-01-01 Annual                                                 24,616.84
           Hourly                                                     12.46
2008-01-01 Annual                                                 24,185.70

Country                                                   Belgium  ...  \
Series                 In 2015 constant prices at 2015 USD PPPs  ...
```

(continues on next page)

(continued from previous page)

```
Time       Pay period                                          ...
2006-01-01 Annual                                   21,042.28  ...
           Hourly                                       10.09  ...
2007-01-01 Annual                                   21,310.05  ...
           Hourly                                       10.22  ...
2008-01-01 Annual                                   21,416.96  ...

Country                                                 United Kingdom  \
Series                In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                               20,376.32
           Hourly                                                    9.81
2007-01-01 Annual                                               20,954.13
           Hourly                                                   10.07
2008-01-01 Annual                                               20,902.87

Country                                          United States  \
Series                In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                   12,594.40
           Hourly                                        6.05
2007-01-01 Annual                                   12,974.40
           Hourly                                        6.24
2008-01-01 Annual                                   14,097.56

Country
Series                In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                               12,594.40
           Hourly                                                    6.05
2007-01-01 Annual                                               12,974.40
           Hourly                                                    6.24
2008-01-01 Annual                                               14,097.56

[5 rows x 64 columns]
```

We can also pass in an argument to select the level we would like to stack

```
realwage.stack(level='Country').head()
```

```
Series               In 2015 constant prices at 2015 USD PPPs         \
Pay period                                             Annual Hourly
Time       Country
2006-01-01 Australia                                20,410.65  10.33
           Belgium                                  21,042.28  10.09
           Brazil                                    3,310.51   1.41
           Canada                                   13,649.69   6.56
           Chile                                     5,201.65   2.22

Series               In 2015 constant prices at 2015 USD exchange rates
Pay period                                                       Annual Hourly
Time       Country
2006-01-01 Australia                                            23,826.64  12.06
           Belgium                                              20,228.74   9.70
           Brazil                                                2,032.87   0.87
```

(continues on next page)

(continued from previous page)

```
            Canada                                                 14,335.12   6.89
            Chile                                                   3,333.76   1.42
```

Using a `DatetimeIndex` makes it easy to select a particular time period.

Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data

```
realwage.loc['2015'].stack(level=(1, 2)).transpose().head()
```

```
Time                                       2015-01-01         \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                    Annual Hourly
Country
Australia                                   21,715.53  10.99
Belgium                                     21,588.12  10.35
Brazil                                       4,628.63   2.00
Canada                                      16,536.83   7.95
Chile                                        6,633.56   2.80

Time
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                                Annual Hourly
Country
Australia                                                 25,349.90  12.83
Belgium                                                   20,753.48   9.95
Brazil                                                     2,842.28   1.21
Canada                                                    17,367.24   8.35
Chile                                                      4,251.49   1.81
```

For the rest of lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars.

To create our filtered dataframe (`realwage_f`), we can use the `xs` method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case)

```
realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD exchange␣
↪rates'),
                         level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

```
Country     Australia  Belgium  Brazil  ...  Turkey  United Kingdom  \
Time                                    ...
2006-01-01      12.06     9.70    0.87  ...    2.27            9.81
2007-01-01      12.46     9.82    0.92  ...    2.26           10.07
2008-01-01      12.24     9.87    0.96  ...    2.22           10.04
2009-01-01      12.40    10.21    1.03  ...    2.28           10.15
2010-01-01      12.34    10.05    1.08  ...    2.30            9.96

Country     United States
Time
2006-01-01           6.05
2007-01-01           6.24
2008-01-01           6.78
2009-01-01           7.58
```

(continues on next page)

(continued from previous page)

```
2010-01-01           7.88

[5 rows x 32 columns]
```

1.3 Merging Dataframes and Filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, we'll add the continent of each country to `realwage_f` with the `merge` function.

The dataset can be accessed with the following link:

```
url2 = 'https://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/countries.csv'
```

```
worlddata = pd.read_csv(url2, sep=';')
worlddata.head()
```

```
    Country (en) Country (de)        Country (local)  ... Deathrate  \
0    Afghanistan  Afghanistan  Afganistan/Afqanestan  ...     13.70
1          Egypt      Ägypten                  Misr  ...      4.70
2  Åland Islands  Ålandinseln                 Åland  ...      0.00
3        Albania     Albanien              Shqipëria  ...      6.70
4        Algeria     Algerien     Al-Jaza'ir/Algérie  ...      4.30

  Life expectancy                                                Url
0           51.30  https://www.laenderdaten.info/Asien/Afghanista...
1           72.70  https://www.laenderdaten.info/Afrika/Aegypten/...
2            0.00  https://www.laenderdaten.info/Europa/Aland/ind...
3           78.30  https://www.laenderdaten.info/Europa/Albanien/...
4           76.80  https://www.laenderdaten.info/Afrika/Algerien/...

[5 rows x 17 columns]
```

First, we'll select just the country and continent variables from `worlddata` and rename the column to 'Country'

```
worlddata = worlddata[['Country (en)', 'Continent']]
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

```
         Country Continent
0    Afghanistan      Asia
1          Egypt    Africa
2  Åland Islands    Europe
3        Albania    Europe
4        Algeria    Africa
```

We want to merge our new dataframe, `worlddata`, with `realwage_f`.

The pandas `merge` function allows dataframes to be joined together by rows.

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes

```
realwage_f.transpose().head()
```

```
Time       2006-01-01  2007-01-01  2008-01-01  ...  2014-01-01  2015-01-01  \
Country                                        ...
Australia       12.06       12.46       12.24  ...       12.67       12.83
Belgium          9.70        9.82        9.87  ...       10.01        9.95
Brazil           0.87        0.92        0.96  ...        1.21        1.21
Canada           6.89        6.96        7.24  ...        8.22        8.35
Chile            1.42        1.45        1.44  ...        1.76        1.81

Time       2016-01-01
Country
Australia       12.98
Belgium          9.76
Brazil           1.24
Canada           8.48
Chile            1.91

[5 rows x 11 columns]
```

We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join.

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`.

This is illustrated by the red shading in the following diagram

We will also need to specify where the country name is located in each dataframe, which will be the `key` that is used to merge the dataframes 'on'.

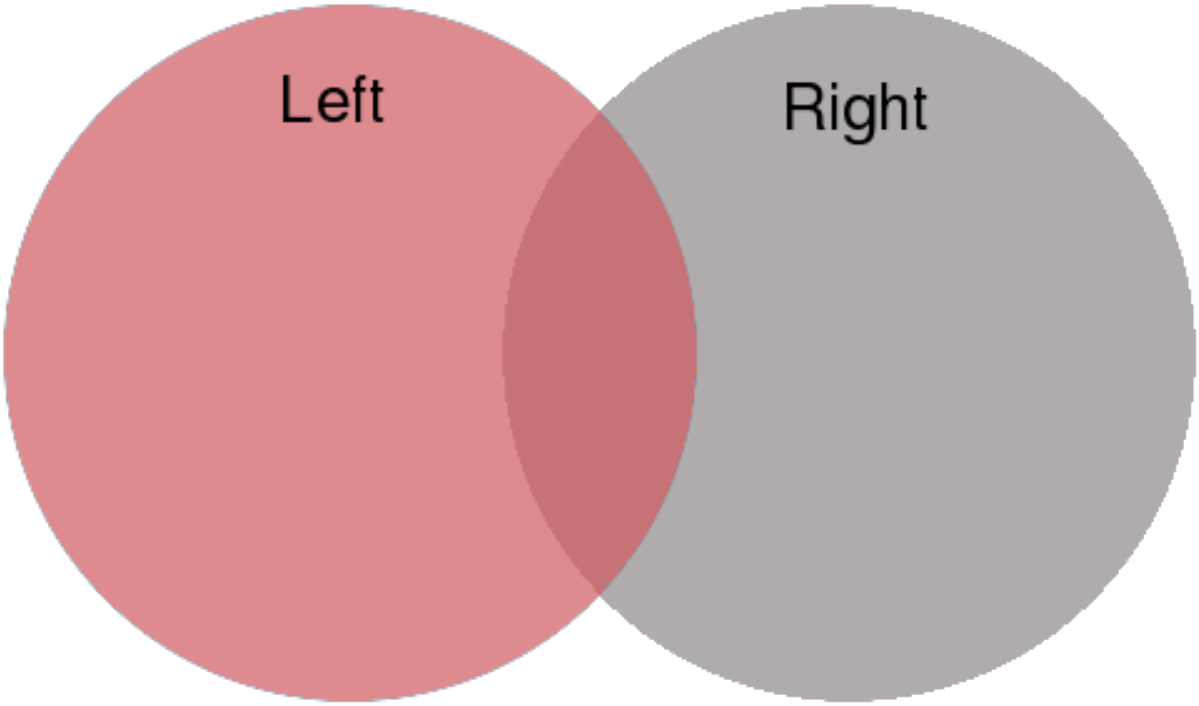
Our 'left' dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`.

Our 'right' dataframe (`worlddata`) contains countries in the 'Country' column, so we set `right_on='Country'`

```
merged = pd.merge(realwage_f.transpose(), worlddata,
                  how='left', left_index=True, right_on='Country')
merged.head()
```

```
        2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
17.00                 12.06                12.46                12.24  ...
23.00                  9.70                 9.82                 9.87  ...
32.00                  0.87                 0.92                 0.96  ...
100.00                 6.89                 6.96                 7.24  ...
38.00                  1.42                 1.45                 1.44  ...

        2016-01-01 00:00:00    Country      Continent
```

(continues on next page)



(continued from previous page)

```
17.00                 12.98  Australia      Australia
23.00                  9.76    Belgium         Europe
32.00                  1.24     Brazil  South America
100.00                 8.48     Canada  North America
38.00                  1.91      Chile  South America

[5 rows x 13 columns]
```

Countries that appeared in `realwage_f` but not in `worlddata` will have `NaN` in the Continent column.

To check whether this has occurred, we can use `.isnull()` on the continent column and filter the merged dataframe

```
merged[merged['Continent'].isnull()]
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...
NaN                 0.23                 0.45                 0.39  ...
NaN                 1.50                 1.64                 1.71  ...

     2016-01-01 00:00:00            Country  Continent
NaN                 5.28              Korea        NaN
NaN                 0.55  Russian Federation        NaN
NaN                 2.08     Slovak Republic        NaN

[3 rows x 13 columns]
```

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents.

`.map()` will match countries in `merged['Country']` with their continent from the dictionary.

Notice how countries not in our dictionary are mapped with `NaN`

```
missing_continents = {'Korea': 'Asia',
                      'Russian Federation': 'Europe',
                      'Slovak Republic': 'Europe'}

merged['Country'].map(missing_continents)
```

```
17.00        NaN
23.00        NaN
32.00        NaN
100.00       NaN
38.00        NaN
108.00       NaN
41.00        NaN
225.00       NaN
53.00        NaN
58.00        NaN
45.00        NaN
68.00        NaN
233.00       NaN
86.00        NaN
88.00        NaN
91.00        NaN
NaN         Asia
117.00       NaN
122.00       NaN
123.00       NaN
138.00       NaN
153.00       NaN
151.00       NaN
174.00       NaN
175.00       NaN
NaN       Europe
NaN       Europe
198.00       NaN
200.00       NaN
227.00       NaN
241.00       NaN
240.00       NaN
Name: Country, dtype: object
```

We don't want to overwrite the entire series with this mapping.

`.fillna()` only fills in `NaN` values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged

```
merged['Continent'] = merged['Continent'].fillna(merged['Country'].map(missing_
↪continents))

# Check for whether continents were correctly mapped

merged[merged['Country'] == 'Korea']
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...

     2016-01-01 00:00:00  Country  Continent
NaN                 5.28    Korea       Asia

[1 rows x 13 columns]
```

We will also combine the Americas into a single continent - this will make our visualization nicer later on.

To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```
replace = ['Central America', 'North America', 'South America']

for country in replace:
    merged['Continent'].replace(to_replace=country,
                                value='America',
                                inplace=True)
```

Now that we have all the data we want in a single `DataFrame`, we will reshape it back into panel form with a `MultiIndex`.

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on.

By default, levels will be sorted top-down

```
merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

```
                      2006-01-01  2007-01-01  2008-01-01  ...  2014-01-01  \
Continent Country                                         ...
America   Brazil            0.87        0.92        0.96  ...        1.21
          Canada            6.89        6.96        7.24  ...        8.22
          Chile             1.42        1.45        1.44  ...        1.76
          Colombia          1.01        1.02        1.01  ...        1.13
          Costa Rica         NaN         NaN         NaN  ...        2.41

                      2015-01-01  2016-01-01
Continent Country
America   Brazil            1.21        1.24
          Canada            8.35        8.48
          Chile             1.81        1.91
          Colombia          1.13        1.12
          Costa Rica        2.56        2.63

[5 rows x 11 columns]
```

While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format

```
merged.columns
```

```
Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
       2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
       2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
       2015-01-01 00:00:00, 2016-01-01 00:00:00],
      dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`

```
merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns
```

```
DatetimeIndex(['2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
               '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
               '2014-01-01', '2015-01-01', '2016-01-01'],
              dtype='datetime64[ns]', name='Time', freq=None)
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose `merged`

```
merged = merged.transpose()
merged.head()
```

```
Continent  America                ...   Europe
Country     Brazil Canada Chile   ... Slovenia Spain United Kingdom
Time                              ...
2006-01-01    0.87   6.89  1.42   ...     3.92  3.99           9.81
2007-01-01    0.92   6.96  1.45   ...     3.88  4.10          10.07
2008-01-01    0.96   7.24  1.44   ...     3.96  4.14          10.04
2009-01-01    1.03   7.67  1.52   ...     4.08  4.32          10.15
2010-01-01    1.08   7.94  1.56   ...     4.81  4.30           9.96

[5 rows x 32 columns]
```

1.4 Grouping and Summarizing Data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as `.mean()` or `.max()`.

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

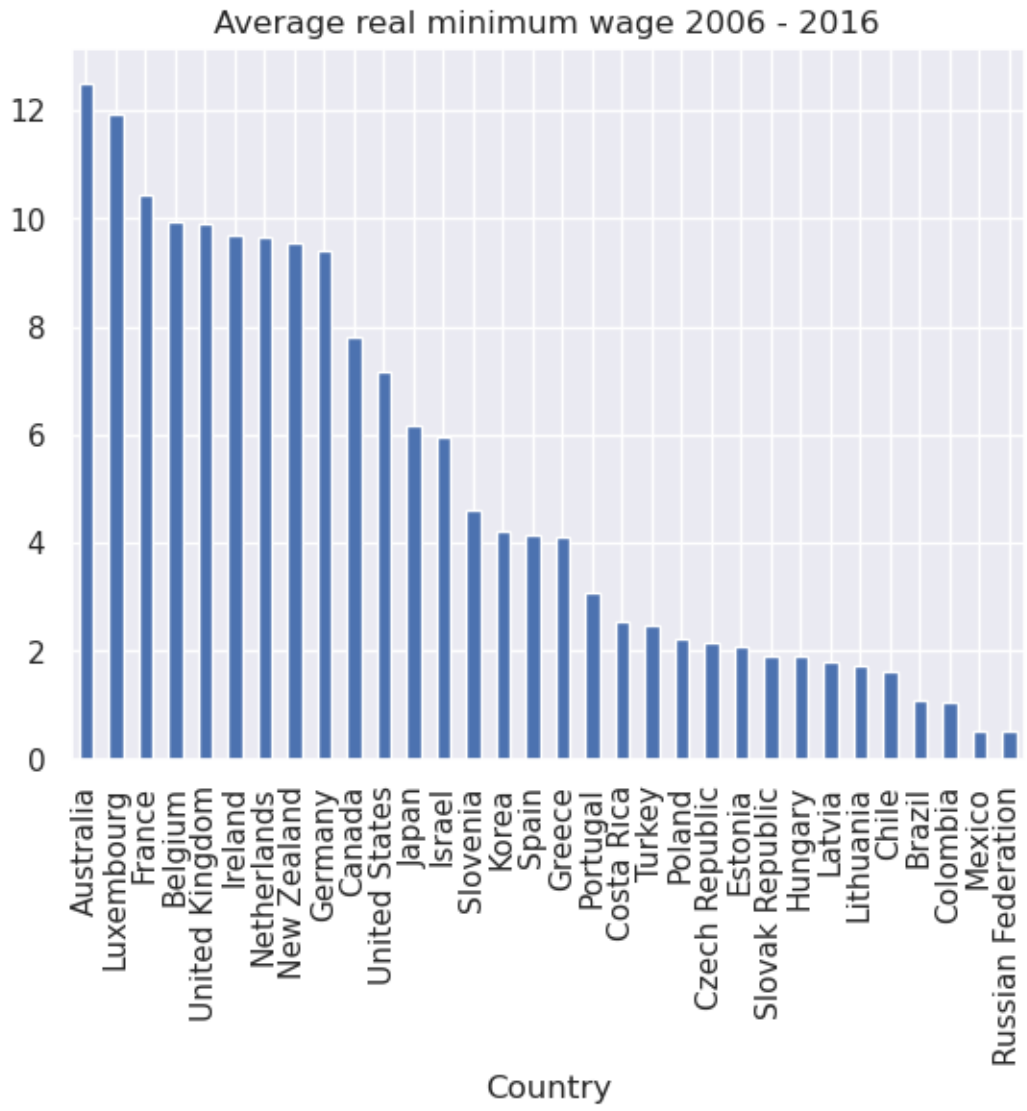
```
merged.mean().head(10)
```

```
Continent  Country
America    Brazil          1.09
           Canada          7.82
           Chile           1.62
           Colombia        1.07
           Costa Rica      2.53
           Mexico          0.53
           United States   7.15
Asia       Israel          5.95
           Japan           6.18
           Korea           4.22
dtype: float64
```

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
```

```
merged.mean().sort_values(ascending=False).plot(kind='bar',
                                                title="Average real minimum wage 2006
 ↪- 2016")

# Set country labels
country_labels = merged.mean().sort_values(ascending=False).index.get_level_values(
 ↪'Country').tolist()
plt.xticks(range(0, len(country_labels)), country_labels)
plt.xlabel('Country')

plt.show()
```



Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time)

```
merged.mean(axis=1).head()
```

```
Time
2006-01-01   4.69
2007-01-01   4.84
2008-01-01   4.90
2009-01-01   5.08
2010-01-01   5.11
dtype: float64
```

We can plot this time series as a line graph

```
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```

We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
merged.groupby(level='Continent', axis=1).mean().head()
```

```
Continent   America  Asia  Australia  Europe
Time
```

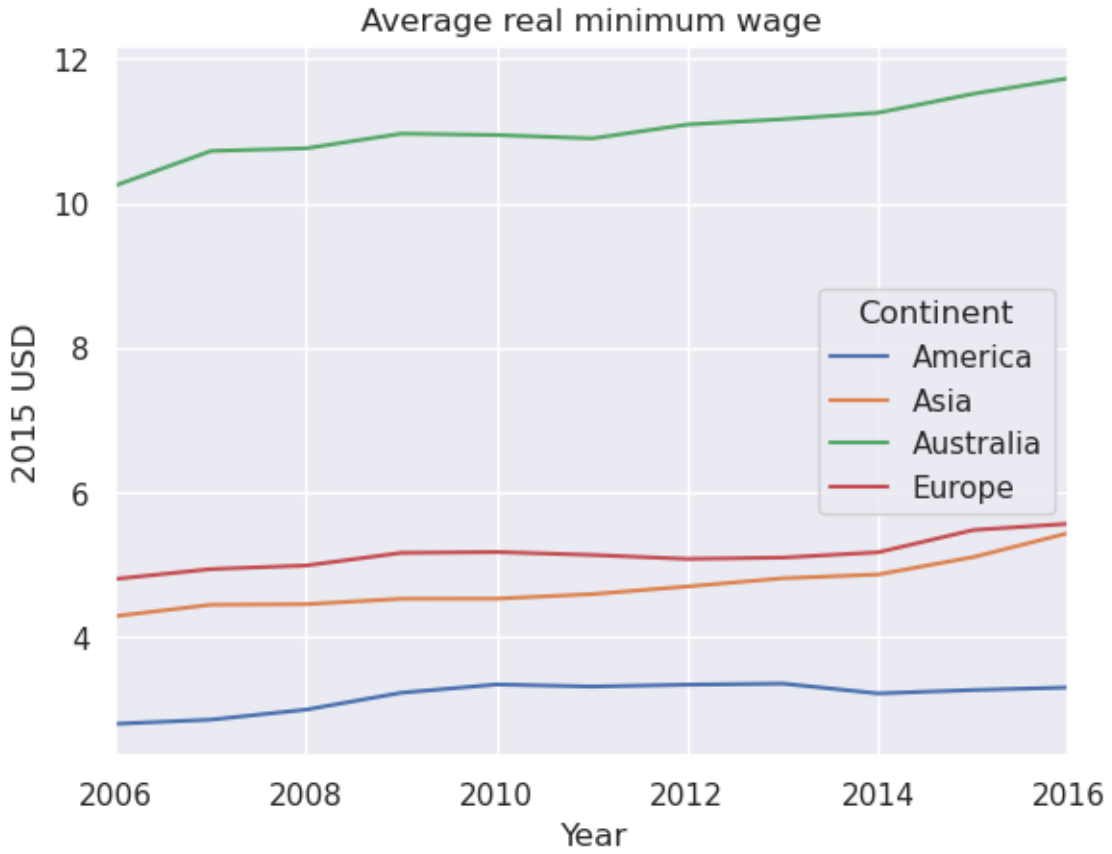
(continues on next page)

(continued from previous page)

```
2006-01-01     2.80  4.29      10.25    4.80
2007-01-01     2.85  4.44      10.73    4.94
2008-01-01     2.99  4.45      10.76    4.99
2009-01-01     3.23  4.53      10.97    5.16
2010-01-01     3.34  4.53      10.95    5.17
```

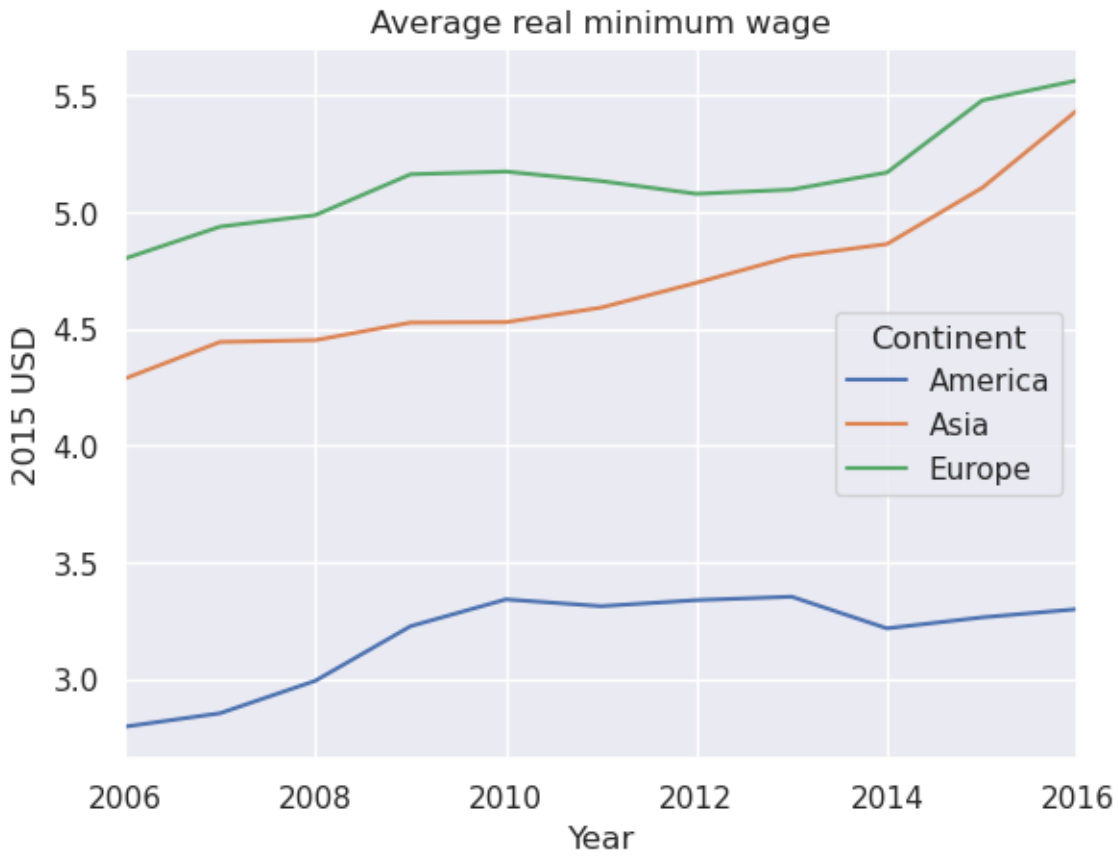
We can plot the average minimum wages in each continent as a time series

```
merged.groupby(level='Continent', axis=1).mean().plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



We will drop Australia as a continent for plotting purposes

```
merged = merged.drop('Australia', level='Continent', axis=1)
merged.groupby(level='Continent', axis=1).mean().plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



`.describe()` is useful for quickly retrieving a number of common summary statistics

```
merged.stack().describe()
```

```
Continent  America  Asia  Europe
count        69.00 44.00  200.00
mean          3.19  4.70    5.15
std           3.02  1.56    3.82
min           0.52  2.22    0.23
25%           1.03  3.37    2.02
50%           1.44  5.48    3.54
75%           6.96  5.95    9.70
max           8.48  6.65   12.39
```

This is a simplified way to use `groupby`.

Using `groupby` generally follows a 'split-apply-combine' process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The `groupby` method achieves the first step of this process, creating a new `DataFrameGroupBy` object with data split into groups.

Let's split `merged` by continent again, this time using the `groupby` function, and name the resulting object `grouped`

```
grouped = merged.groupby(level='Continent', axis=1)
grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fd5f2f5f110>
```

Calling an aggregation method on the object applies the function to each group, the results of which are combined in a new data structure.

For example, we can return the number of countries in our dataset for each continent using `.size()`.

In this case, our new data structure is a `Series`

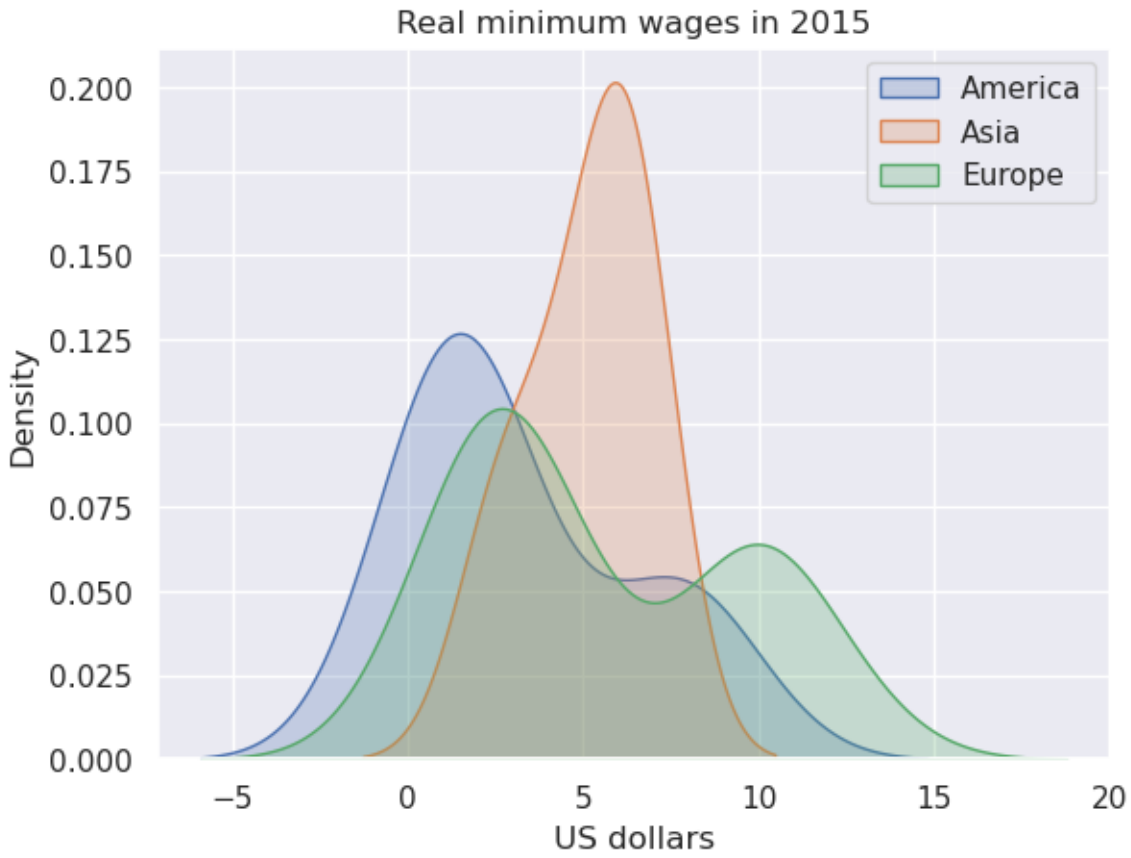
```
grouped.size()
```

```
Continent
America     7
Asia        4
Europe     19
dtype: int64
```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent.

`grouped.groups.keys()` will return the keys from the `groupby` object

```
continents = grouped.groups.keys()

for continent in continents:
    sns.kdeplot(grouped.get_group(continent).loc['2015'].unstack(), label=continent,␣
 ↪fill=True)

plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.legend()
plt.show()
```



1.5 Final Remarks

This lecture has provided an introduction to some of pandas' more advanced features, including multiindices, merging, grouping and plotting.

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures.

1.6 Exercises

Exercise 1.6.1

In these exercises, you'll work with a dataset of employment rates in Europe by age and sex from Eurostat.

The dataset can be accessed with the following link:

```
url3 = 'https://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/employ.csv'
```

Reading in the CSV file returns a panel dataset in long format. Use `.pivot_table()` to construct a wide format dataframe with a `MultiIndex` in the columns.

Start off by exploring the dataframe and the variables available in the `MultiIndex` levels.

Write a program that quickly returns all values in the `MultiIndex`.

Solution to Exercise 1.6.1

```
employ = pd.read_csv(url3)
employ = employ.pivot_table(values='Value',
                            index=['DATE'],
                            columns=['UNIT','AGE', 'SEX', 'INDIC_EM', 'GEO'])
employ.index = pd.to_datetime(employ.index) # ensure that dates are datetime format
employ.head()
```

```
UNIT       Percentage of total population                  ...  \
AGE                   From 15 to 24 years                  ...
SEX                                 Females                ...
INDIC_EM                Active population                  ...
GEO                                 Austria Belgium Bulgaria  ...
DATE                                                       ...
2007-01-01                            56.00   31.60    26.00  ...
2008-01-01                            56.20   30.80    26.10  ...
2009-01-01                            56.20   29.90    24.80  ...
2010-01-01                            54.00   29.80    26.60  ...
2011-01-01                            54.80   29.80    24.80  ...

UNIT                                          Thousand persons           \
AGE                                        From 55 to 64 years
SEX                                                      Total
INDIC_EM   Total employment (resident population concept - LFS)
GEO                                                Switzerland   Turkey
DATE
2007-01-01                                                 NaN   1,282.00
2008-01-01                                                 NaN   1,354.00
2009-01-01                                                 NaN   1,449.00
2010-01-01                                              640.00   1,583.00
2011-01-01                                              661.00   1,760.00

UNIT
AGE
SEX
INDIC_EM
GEO        United Kingdom
DATE
2007-01-01       4,131.00
2008-01-01       4,204.00
2009-01-01       4,193.00
2010-01-01       4,186.00
2011-01-01       4,164.00

[5 rows x 1440 columns]
```

This is a large dataset so it is useful to explore the levels and variables available

```
employ.columns.names
```

```
FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop

```
for name in employ.columns.names:
    print(name, employ.columns.get_level_values(name).unique())
```

```
UNIT Index(['Percentage of total population', 'Thousand persons'], dtype='object',␣
 ↪name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],␣
 ↪dtype='object', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population␣
 ↪concept - LFS)'], dtype='object', name='INDIC_EM')
GEO Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
       'Denmark', 'Estonia', 'Euro area (17 countries)',
       'Euro area (18 countries)', 'Euro area (19 countries)',
       'European Union (15 countries)', 'European Union (27 countries)',
       'European Union (28 countries)', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

Exercise 1.6.2

Filter the above dataframe to only include employment as a percentage of 'active population'.

Create a grouped boxplot using `seaborn` of employment rates in 2015 by age group and sex.

Hint: `GEO` includes both areas and countries.

Solution to Exercise 1.6.2

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
employ.columns = employ.columns.swaplevel(0,-1)
employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in `GEO` which are not countries.

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in `GEO` that begin with 'Euro'

```
geo_list = employ.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in geo_list if not x.startswith('Euro')]
employ = employ[countries]
employ.columns.get_level_values('GEO').unique()
```

```
Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
       'Denmark', 'Estonia', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

Select only percentage employed in the active population from the dataframe

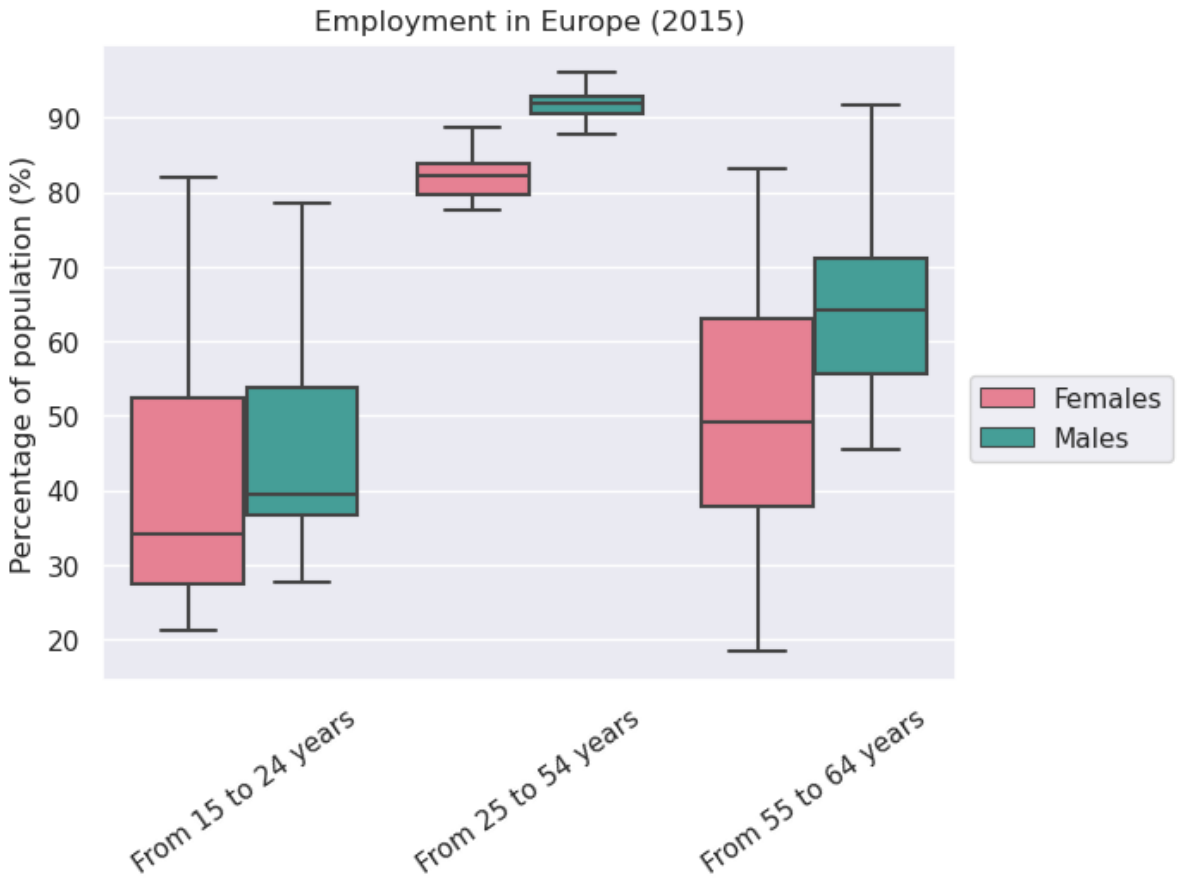
```
employ_f = employ.xs(('Percentage of total population', 'Active population'),
                     level=('UNIT', 'INDIC_EM'),
                     axis=1)
employ_f.head()
```

```
GEO                    Austria                ...      United Kingdom        \
AGE         From 15 to 24 years                ... From 55 to 64 years
SEX                     Females Males Total   ...              Females Males
DATE                                          ...
2007-01-01                56.00 62.90 59.40   ...                49.90 68.90
2008-01-01                56.20 62.90 59.50   ...                50.20 69.80
2009-01-01                56.20 62.90 59.50   ...                50.60 70.30
2010-01-01                54.00 62.60 58.30   ...                51.10 69.20
2011-01-01                54.80 63.60 59.20   ...                51.30 68.40

GEO
AGE
SEX         Total
DATE
2007-01-01 59.30
2008-01-01 59.80
2009-01-01 60.30
2010-01-01 60.00
2011-01-01 59.70

[5 rows x 306 columns]
```

Drop the 'Total' value before creating the grouped boxplot

```
employ_f = employ_f.drop('Total', level='SEX', axis=1)
```

```
box = employ_f.loc['2015'].unstack().reset_index()
sns.boxplot(x="AGE", y=0, hue="SEX", data=box, palette=("husl"), showfliers=False)
plt.xlabel('')
plt.xticks(rotation=35)
plt.ylabel('Percentage of population (%)')
plt.title('Employment in Europe (2015)')
plt.legend(bbox_to_anchor=(1,0.5))
plt.show()
```


Employment in Europe (2015)
Percentage of population (%)
90
80
70
60
50
40
30
20
From 15 to 24 years
From 25 to 54 years
From 55 to 64 years
Females
Males

CHAPTER

TWO

LINEAR REGRESSION IN PYTHON

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install linearmodels
```

2.1 Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables.

In this lecture, we'll use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models.

Along the way, we'll discuss a variety of topics, including

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper [AJR01].

- You can download a copy here.

In the paper, the authors emphasize the importance of institutions in economic development.

The main contribution is the use of settler mortality rates as a source of *exogenous* variation in institutional differences.

Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col
from linearmodels.iv import IV2SLS
import seaborn as sns
sns.set_theme()
```

2.1.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [Woo15].

2.2 Simple Linear Regression

[AJR01] wish to determine whether or not differences in institutions can help to explain observed economic outcomes.

How do we measure *institutional differences* and *economic outcomes*?

In this paper,

- economic outcomes are proxied by log GDP per capita in 1995, adjusted for exchange rates.
- institutional differences are proxied by an index of protection against expropriation on average over 1985-95, constructed by the Political Risk Services Group.

These variables and other data used in the paper are available for download on Daron Acemoglu's webpage.

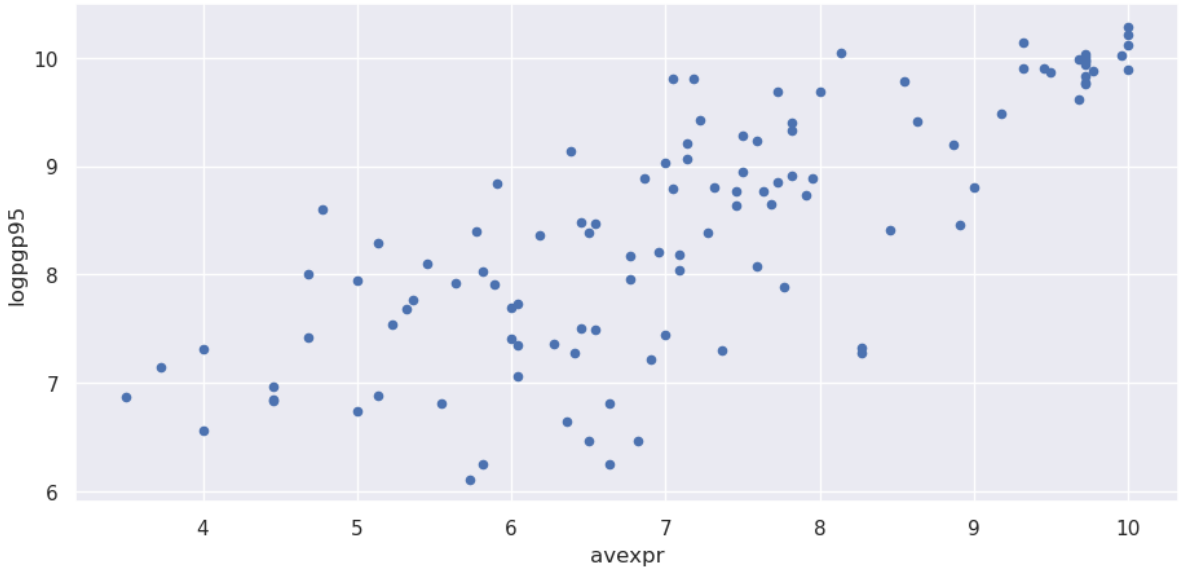
We will use pandas' `.read_stata()` function to read in data contained in the `.dta` files to dataframes

```
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable1.dta?raw=true')
df1.head()
```

```
  shortnam   euro1900  excolony    avexpr  logpgp95  cons1  cons90  democ00a  \
0      AFG   0.000000       1.0       NaN       NaN    1.0     2.0       1.0
1      AGO   8.000000       1.0  5.363636  7.770645    3.0     3.0       0.0
2      ARE   0.000000       1.0  7.181818  9.804219    NaN     NaN       NaN
3      ARG  60.000004       1.0  6.386364  9.133459    1.0     6.0       3.0
4      ARM   0.000000       0.0       NaN  7.682482    NaN     NaN       NaN

   cons00a    extmort4    logem4  loghjypl  baseco
0      1.0   93.699997  4.540098       NaN     NaN
1      1.0  280.000000  5.634789 -3.411248     1.0
2      NaN         NaN       NaN       NaN     NaN
3      3.0   68.900002  4.232656 -0.872274     1.0
4      NaN         NaN       NaN       NaN     NaN
```

Let's use a scatterplot to see whether any obvious relationship exists between GDP per capita and the protection against expropriation index

```
df1.plot(x='avexpr', y='logpgp95', kind='scatter')
plt.show()
```



The plot shows a fairly strong positive relationship between protection against expropriation and log GDP per capita.

Specifically, if higher protection against expropriation is a measure of institutional quality, then better institutions appear to be positively correlated with better economic outcomes (higher GDP per capita).

Given the plot, choosing a linear model to describe this relationship seems like a reasonable assumption.

We can write our model as

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + u_i$$

where:

- β_0 is the intercept of the linear trend line on the y-axis
- β_1 is the slope of the linear trend line, representing the *marginal effect* of protection against risk on log GDP per capita
- u_i is a random error term (deviations of observations from the linear trend due to factors not included in the model)

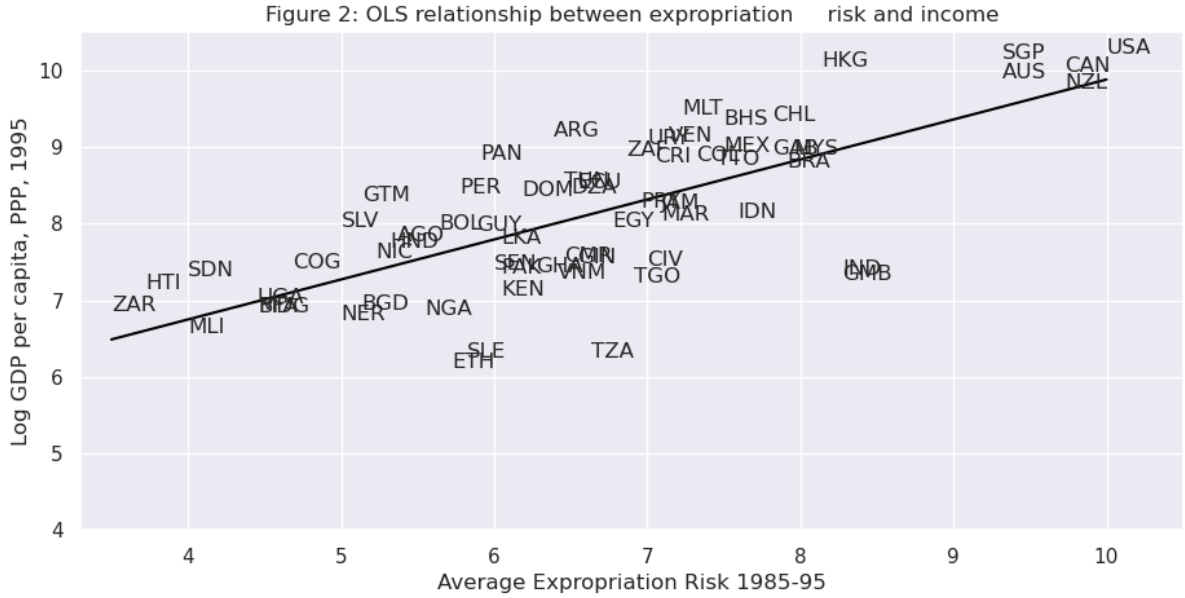
Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in [AJR01])

```
# Dropping NA's is required to use numpy's polyfit
df1_subset = df1.dropna(subset=['logpgp95', 'avexpr'])

# Use only 'base sample' for plotting purposes
df1_subset = df1_subset[df1_subset['baseco'] == 1]

X = df1_subset['avexpr']
y = df1_subset['logpgp95']
labels = df1_subset['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([3.3,10.5])
ax.set_ylim([4,10.5])
ax.set_xlabel('Average Expropriation Risk 1985-95')
ax.set_ylabel('Log GDP per capita, PPP, 1995')
ax.set_title('Figure 2: OLS relationship between expropriation \
    risk and income')
plt.show()
```



The most common technique to estimate the parameters (β's) of the linear model is Ordinary Least Squares (OLS).

As the name implies, an OLS model is solved by finding the parameters that minimize *the sum of squared residuals*, i.e.

$$\min_{\hat{\beta}} \sum_{i=1}^{N} \hat{u}_i^2$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable.

To estimate the constant term β_0, we need to add a column of 1's to our dataset (consider the equation if β_0 was replaced with $\beta_0 x_i$ and $x_i = 1$)

```
df1['const'] = 1
```

Now we can construct our model in `statsmodels` using the OLS function.

We will use `pandas` dataframes with `statsmodels`, however standard arrays can also be used as arguments

```
reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], \
    missing='drop')
type(reg1)
```

```
statsmodels.regression.linear_model.OLS
```

So far we have simply constructed our model.

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$

```
results = reg1.fit()
type(results)
```

```
statsmodels.regression.linear_model.RegressionResultsWrapper
```

We now have the fitted regression model stored in `results`.

To view the OLS regression results, we can call the `.summary()` method.

Note that an observation was mistakenly dropped from the results in the original paper (see the note located in `maketable2.do` from Acemoglu's webpage), and thus the coefficients differ slightly.

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.611
Model:                            OLS   Adj. R-squared:                  0.608
Method:                 Least Squares   F-statistic:                     171.4
Date:                Tue, 30 Jan 2024   Prob (F-statistic):           4.16e-24
Time:                        06:46:15   Log-Likelihood:                -119.71
No. Observations:                 111   AIC:                             243.4
Df Residuals:                     109   BIC:                             248.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.6261      0.301     15.391      0.000       4.030       5.222
avexpr         0.5319      0.041     13.093      0.000       0.451       0.612
==============================================================================
Omnibus:                        9.251   Durbin-Watson:                   1.689
Prob(Omnibus):                  0.010   Jarque-Bera (JB):                9.170
Skew:                          -0.680   Prob(JB):                       0.0102
Kurtosis:                       3.362   Cond. No.                         33.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

From our results, we see that

- The intercept $\hat{\beta}_0 = 4.63$.
- The slope $\hat{\beta}_1 = 0.53$.

- The positive $\hat{\beta}_1$ parameter estimate implies that. institutional quality has a positive effect on economic outcomes, as we saw in the figure.
- The p-value of 0.000 for $\hat{\beta}_1$ implies that the effect of institutions on GDP is statistically significant (using p < 0.05 as a rejection rule).
- The R-squared value of 0.611 indicates that around 61% of variation in log GDP per capita is explained by protection against expropriation.

Using our parameter estimates, we can now write our estimated relationship as

$$\widehat{logpgp95}_i = 4.63 + 0.53 \; avexpr_i$$

This equation describes the line that best fits our data, as shown in Figure 2.

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection.

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38.

```
mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
```

```
6.515625
```

```
predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
```

```
8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set $constant = 1$ and $avexpr_i = mean\_expr$

```
results.predict(exog=[1, mean_expr])
```

```
array([8.09156367])
```

We can obtain an array of predicted $logpgp95_i$ for every value of $avexpr_i$ in our dataset by calling `.predict()` on our results.

Plotting the predicted values against $avexpr_i$ shows that the predicted values lie along the linear line that we fitted above.

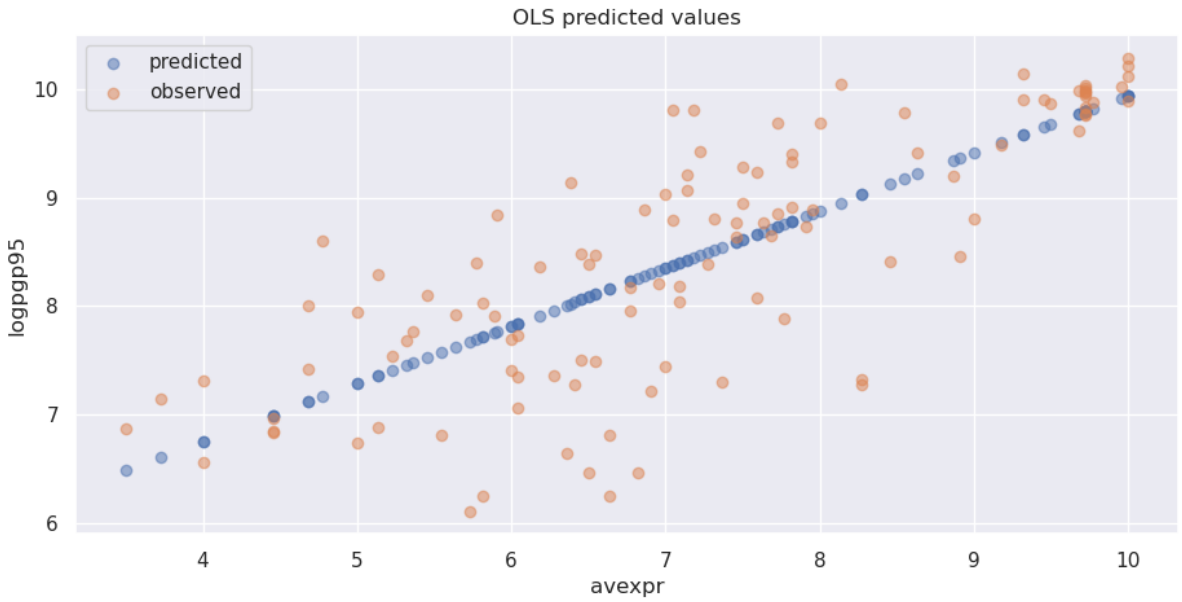
The observed values of $logpgp95_i$ are also plotted for comparison purposes

```
# Drop missing observations from whole sample

df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])

# Plot predicted values

fix, ax = plt.subplots()
ax.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5,
        label='predicted')

# Plot observed values
```

(continues on next page)

(continued from previous page)

```
ax.scatter(df1_plot['avexpr'], df1_plot['logpgp95'], alpha=0.5,
        label='observed')

ax.legend()
ax.set_title('OLS predicted values')
ax.set_xlabel('avexpr')
ax.set_ylabel('logpgp95')
plt.show()
```



2.3 Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model.

Leaving out variables that affect $logpgp95_i$ will result in **omitted variable bias**, yielding biased and inconsistent parameter estimates.

We can extend our bivariate regression model to a **multivariate regression model** by adding in other factors that may affect $logpgp95_i$.

[AJR01] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```
df2 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable2.dta?raw=true')
```

(continues on next page)

(continued from previous page)

```
# Add constant term to dataset
df2['const'] = 1

# Create lists of variables to be used in each regression
X1 = ['const', 'avexpr']
X2 = ['const', 'avexpr', 'lat_abst']
X3 = ['const', 'avexpr', 'lat_abst', 'asia', 'africa', 'other']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df2['logpgp95'], df2[X1], missing='drop').fit()
reg2 = sm.OLS(df2['logpgp95'], df2[X2], missing='drop').fit()
reg3 = sm.OLS(df2['logpgp95'], df2[X3], missing='drop').fit()
```

Now that we have fitted our model, we will use `summary_col` to display the results in a single table (model numbers correspond to those in the paper)

```
info_dict={'R-squared' : lambda x: f"{x.rsquared:.2f}",
           'No. observations' : lambda x: f"{int(x.nobs):d}"}

results_table = summary_col(results=[reg1,reg2,reg3],
                            float_format='%0.2f',
                            stars = True,
                            model_names=['Model 1',
                                         'Model 3',
                                         'Model 4'],
                            info_dict=info_dict,
                            regressor_order=['const',
                                             'avexpr',
                                             'lat_abst',
                                             'asia',
                                             'africa'])

results_table.add_title('Table 2 - OLS Regressions')

print(results_table)
```

```
        Table 2 - OLS Regressions
=========================================
                Model 1 Model 3 Model 4
-----------------------------------------
const           4.63*** 4.87*** 5.85***
                (0.30)  (0.33)  (0.34)
avexpr          0.53*** 0.46*** 0.39***
                (0.04)  (0.06)  (0.05)
lat_abst                0.87*   0.33
                        (0.49)  (0.45)
asia                            -0.15
                                (0.15)
africa                          -0.92***
                                (0.17)
other                           0.30
                                (0.37)
R-squared       0.61    0.62    0.72
R-squared Adj.  0.61    0.62    0.70
R-squared       0.61    0.62    0.72
```

(continues on next page)

(continued from previous page)

```
No. observations 111     111     111
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

2.4 Endogeneity

As [AJR01] discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates.

Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression.

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (ie. it should not directly affect the dependent variable, otherwise it would be correlated with u_i due to omitted variable bias)

The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences.

The main contribution of [AJR01] is the use of settler mortality rates to instrument for institutional differences.

They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today.

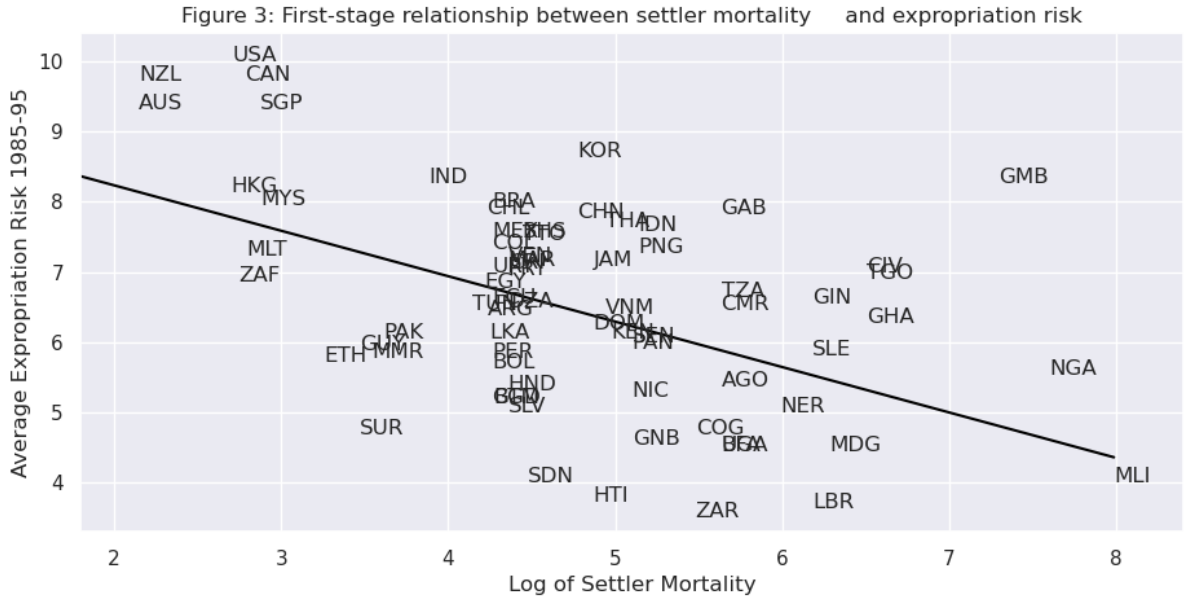
Using a scatterplot (Figure 3 in [AJR01]), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors' hypothesis and satisfying the first condition of a valid instrument.

```
# Dropping NA's is required to use numpy's polyfit
df1_subset2 = df1.dropna(subset=['logem4', 'avexpr'])

X = df1_subset2['logem4']
y = df1_subset2['avexpr']
labels = df1_subset2['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
```

(continues on next page)

(continued from previous page)

```
        color='black')

ax.set_xlim([1.8,8.4])
ax.set_ylim([3.3,10.4])
ax.set_xlabel('Log of Settler Mortality')
ax.set_ylabel('Average Expropriation Risk 1985-95')
ax.set_title('Figure 3: First-stage relationship between settler mortality \
    and expropriation risk')
plt.show()
```



The second condition may not be satisfied if settler mortality rates in the 17th to 19th centuries have a direct effect on current GDP (in addition to their indirect effect through institutions).

For example, settler mortality rates may be related to the current disease environment in a country, which could affect current economic performance.

[AJR01] argue this is unlikely because:

- The majority of settler deaths were due to malaria and yellow fever and had a limited effect on local people.
- The disease burden on local people in Africa or India, for example, did not appear to be higher than average, supported by relatively high population densities in these areas before colonization.

As we appear to have a valid instrument, we can use 2SLS regression to obtain consistent and unbiased parameter estimates.

First stage

The first stage involves regressing the endogenous variable ($avexpr_i$) on the instrument.

The instrument is the set of all exogenous variables in our model (and not just the variable we have replaced).

Using model 1 as an example, our instrument is simply a constant and settler mortality rates $logem4_i$.

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 logem4_i + v_i$$

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```
# Import and select the data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable4.dta?raw=true')
df4 = df4[df4['baseco'] == 1]

# Add a constant variable
df4['const'] = 1

# Fit the first stage regression and print summary
results_fs = sm.OLS(df4['avexpr'],
                    df4[['const', 'logem4']],
                    missing='drop').fit()
print(results_fs.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  avexpr   R-squared:                       0.270
Model:                            OLS   Adj. R-squared:                  0.258
Method:                 Least Squares   F-statistic:                     22.95
Date:                Tue, 30 Jan 2024   Prob (F-statistic):           1.08e-05
Time:                        06:46:17   Log-Likelihood:                -104.83
No. Observations:                  64   AIC:                             213.7
Df Residuals:                      62   BIC:                             218.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.3414      0.611     15.296      0.000       8.121      10.562
logem4        -0.6068      0.127     -4.790      0.000      -0.860      -0.354
==============================================================================
Omnibus:                        0.035   Durbin-Watson:                   2.003
Prob(Omnibus):                  0.983   Jarque-Bera (JB):                0.172
Skew:                           0.045   Prob(JB):                        0.918
Kurtosis:                       2.763   Cond. No.                         19.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Second stage

We need to retrieve the predicted values of $avexpr_i$ using `.predict()`.

We then replace the endogenous variable $avexpr_i$ with the predicted values $\widehat{avexpr}_i$ in the original linear model.

Our second stage regression is thus

$$logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
df4['predicted_avexpr'] = results_fs.predict()

results_ss = sm.OLS(df4['logpgp95'],
```

(continues on next page)

(continued from previous page)

```
                    df4[['const', 'predicted_avexpr']]).fit()
print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.477
Model:                            OLS   Adj. R-squared:                  0.469
Method:                 Least Squares   F-statistic:                     56.60
Date:                Tue, 30 Jan 2024   Prob (F-statistic):           2.66e-10
Time:                        06:46:17   Log-Likelihood:                -72.268
No. Observations:                  64   AIC:                             148.5
Df Residuals:                      62   BIC:                             152.9
Df Model:                           1
Covariance Type:            nonrobust
================================================================================
                     coef    std err          t      P>|t|      [0.025      0.
↪975]
--------------------------------------------------------------------------------
↪-
const                1.9097      0.823      2.320      0.024       0.264       3.
↪555
predicted_avexpr     0.9443      0.126      7.523      0.000       0.693       1.
↪195
==============================================================================
Omnibus:                       10.547   Durbin-Watson:                   2.137
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.010
Skew:                          -0.790   Prob(JB):                      0.00407
Kurtosis:                       4.277   Cond. No.                         58.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of institutions on economic outcomes.

The result suggests a stronger positive relationship than what the OLS results indicated.

Note that while our parameter estimates are correct, our standard errors are not and for this reason, computing 2SLS 'manually' (in stages with OLS) is not recommended.

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of `statsmodels`

Note that when using `IV2SLS`, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

```
iv = IV2SLS(dependent=df4['logpgp95'],
            exog=df4['const'],
            endog=df4['avexpr'],
            instruments=df4['logem4']).fit(cov_type='unadjusted')

print(iv.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Dep. Variable:                logpgp95   R-squared:                      0.1870
Estimator:                     IV-2SLS   Adj. R-squared:                 0.1739
No. Observations:                   64   F-statistic:                    37.568
Date:                Tue, Jan 30 2024   P-value (F-stat)                0.0000
Time:                        06:46:17   Distribution:                  chi2(1)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          1.9097     1.0106     1.8897     0.0588     -0.0710      3.8903
avexpr         0.9443     0.1541     6.1293     0.0000      0.6423      1.2462
==============================================================================

Endogenous: avexpr
Instruments: logem4
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can help to explain differences in income levels across countries today.

[AJR01] use a marginal effect of 0.94 to calculate that the difference in the index between Chile and Nigeria (ie. institutional quality) implies up to a 7-fold difference in income, emphasizing the significance of institutions in economic development.

2.5 Summary

We have demonstrated basic OLS and 2SLS regression in `statsmodels` and `linearmodels`.

If you are familiar with R, you may want to use the formula interface to `statsmodels`, or consider using r2py to call R from within Python.

2.6 Exercises

Exercise 2.6.1

In the lecture, we think the original model suffers from endogeneity bias due to the likely effect income has on institutional development.

Although endogeneity is often best identified by thinking about the data and model, we can formally test for endogeneity using the **Hausman test**.

We want to test for correlation between the endogenous variable, $avexpr_i$, and the errors, u_i

$$H_0 : Cov(avexpr_i, u_i) = 0 \quad (no\ endogeneity)$$
$$H_1 : Cov(avexpr_i, u_i) \neq 0 \quad (endogeneity)$$

This test is running in two stages.

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + \upsilon_i$$

Second, we retrieve the residuals $\hat{\upsilon}_i$ and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha\hat{\upsilon}_i + u_i$$

If α is statistically significant (with a p-value < 0.05), then we reject the null hypothesis and conclude that $avexpr_i$ is endogenous.

Using the above information, estimate a Hausman test and interpret your results.

Solution to Exercise 2.6.1

```
# Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable4.dta?raw=true')

# Add a constant term
df4['const'] = 1

# Estimate the first stage regression
reg1 = sm.OLS(endog=df4['avexpr'],
              exog=df4[['const', 'logem4']],
              missing='drop').fit()

# Retrieve the residuals
df4['resid'] = reg1.resid

# Estimate the second stage residuals
reg2 = sm.OLS(endog=df4['logpgp95'],
              exog=df4[['const', 'avexpr', 'resid']],
              missing='drop').fit()

print(reg2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                logpgp95   R-squared:                       0.689
Model:                            OLS   Adj. R-squared:                  0.679
Method:                 Least Squares   F-statistic:                     74.05
Date:                Tue, 30 Jan 2024   Prob (F-statistic):           1.07e-17
Time:                        06:46:17   Log-Likelihood:                -62.031
No. Observations:                  70   AIC:                             130.1
Df Residuals:                      67   BIC:                             136.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.4782      0.547      4.530      0.000       1.386       3.570
avexpr         0.8564      0.082     10.406      0.000       0.692       1.021
resid         -0.4951      0.099     -5.017      0.000      -0.692      -0.298
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Omnibus:                       17.597   Durbin-Watson:                   2.086
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               23.194
Skew:                          -1.054   Prob(JB):                     9.19e-06
Kurtosis:                       4.873   Cond. No.                         53.8
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

The output shows that the coefficient on the residuals is statistically significant, indicating $avexpr_i$ is endogenous.

Exercise 2.6.2

The OLS parameter β can also be estimated using matrix algebra and `numpy` (you may need to review the numpy lecture to complete this exercise).

The linear equation we want to estimate is (written in matrix form)

$$y = X\beta + u$$

To solve for the unknown parameter β, we want to minimize the sum of squared residuals

$$\min_{\hat{\beta}} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write

$$\min_{\hat{\beta}} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

Solving this optimization problem gives the solution for the $\hat{\beta}$ coefficients

$$\hat{\beta} = (X'X)^{-1}X'y$$

Using the above information, compute $\hat{\beta}$ from model 1 using `numpy` - your results should be the same as those in the `statsmodels` output from earlier in the lecture.

Solution to Exercise 2.6.2

```
# Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
 ↪static/lecture_specific/ols/maketable1.dta?raw=true')
df1 = df1.dropna(subset=['logpgp95', 'avexpr'])

# Add a constant term
df1['const'] = 1

# Define the X and y variables
y = np.asarray(df1['logpgp95'])
X = np.asarray(df1[['const', 'avexpr']])

# Compute β_hat
```

(continues on next page)

(continued from previous page)

```
β_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector β_hat
print(f'β_0 = {β_hat[0]:.2}')
print(f'β_1 = {β_hat[1]:.2}')
```

```
β_0 = 4.6
β_1 = 0.53
```

It is also possible to use `np.linalg.inv(X.T @ X) @ X.T @ y` to solve for β, however `.solve()` is preferred as it involves fewer computations.

CHAPTER

THREE

MAXIMUM LIKELIHOOD ESTIMATION

Contents

3.1 Overview

In a *previous lecture*, we estimated the relationship between dependent and explanatory variables using linear regression.

But what if a linear relationship is not an appropriate assumption for our model?

One widely used alternative is maximum likelihood estimation, which involves specifying a class of distributions, indexed by unknown parameters, and then using the data to pin down these parameter values.

The benefit relative to linear regression is that it allows more flexibility in the probabilistic relationships between variables.

Here we illustrate maximum likelihood by replicating Daniel Treisman's (2016) paper, Russia's Billionaires, which connects the number of billionaires in a country to its economic characteristics.

The paper concludes that Russia has a higher number of billionaires than economic factors such as market size and tax rate predict.

We'll require the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numpy import exp
from scipy.special import factorial
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import statsmodels.api as sm
from statsmodels.api import Poisson
from scipy.stats import norm
from statsmodels.iolib.summary2 import summary_col
```

3.1.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

3.2 Set Up and Assumptions

Let's consider the steps we need to go through in maximum likelihood estimation and how they pertain to this study.

3.2.1 Flow of Ideas

The first step with maximum likelihood estimation is to choose the probability distribution believed to be generating the data.

More precisely, we need to make an assumption as to which *parametric class* of distributions is generating the data.

- e.g., the class of all normal distributions, or the class of all gamma distributions.

Each such class is a family of distributions indexed by a finite number of parameters.

- e.g., the class of normal distributions is a family of distributions indexed by its mean $\mu \in (-\infty, \infty)$ and standard deviation $\sigma \in (0, \infty)$.

We'll let the data pick out a particular element of the class by pinning down the parameters.

The parameter estimates so produced will be called **maximum likelihood estimates**.

3.2.2 Counting Billionaires

Treisman [Tre16] is interested in estimating the number of billionaires in different countries.

The number of billionaires is integer-valued.

Hence we consider distributions that take values only in the nonnegative integers.

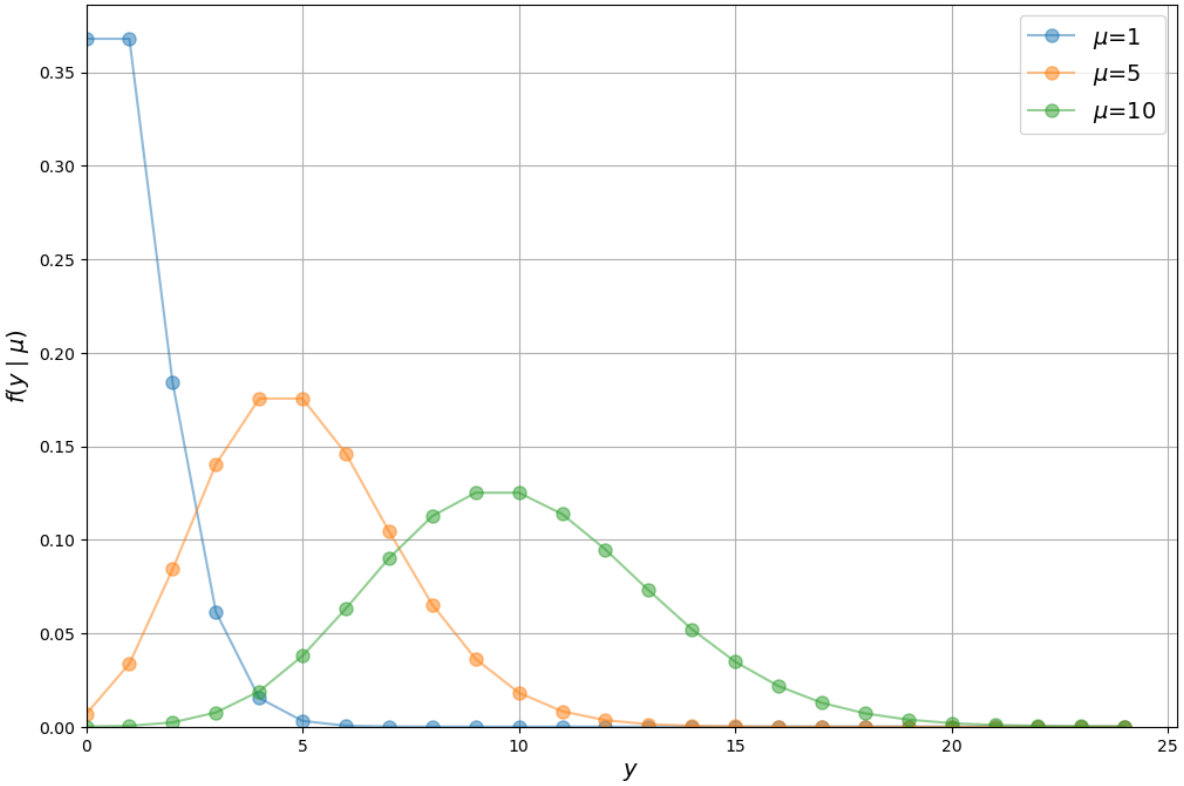
(This is one reason least squares regression is not the best tool for the present problem, since the dependent variable in linear regression is not restricted to integer values)

One integer distribution is the Poisson distribution, the probability mass function (pmf) of which is

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \qquad y = 0, 1, 2, \ldots, \infty$$

We can plot the Poisson distribution over y for different values of μ as follows

```
poisson_pmf = lambda y, μ: μ**y / factorial(y) * exp(-μ)
y_values = range(0, 25)

fig, ax = plt.subplots(figsize=(12, 8))

for μ in [1, 5, 10]:
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(y_values,
            distribution,
            label=f'$\mu$={μ}',
            alpha=0.5,
            marker='o',
            markersize=8)

ax.grid()
ax.set_xlabel('$y$', fontsize=14)
ax.set_ylabel('$f(y \mid \mu)$', fontsize=14)
ax.axis(xmin=0, ymin=0)
ax.legend(fontsize=14)

plt.show()
```



Notice that the Poisson distribution begins to resemble a normal distribution as the mean of y increases.

Let's have a look at the distribution of the data we'll be working with in this lecture.

Treisman's main source of data is *Forbes'* annual rankings of billionaires and their estimated net worth.

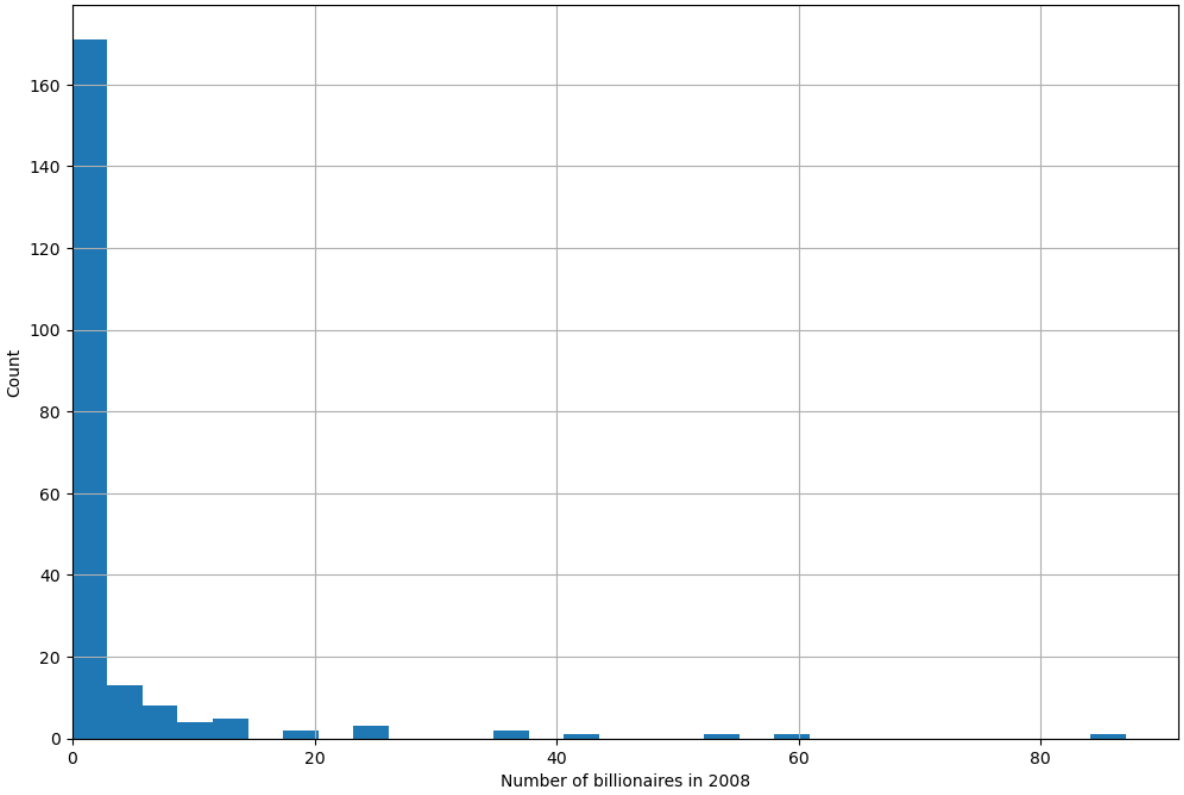
The dataset `mle/fp.dta` can be downloaded from here or its AER page.

```
pd.options.display.max_columns = 10

# Load in data and view
df = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/mle/fp.dta?raw=true')
df.head()
```

```
          country  ccode    year    cyear  numbil  ...   topint08     rintr  \
0  United States    2.0  1990.0  21990.0     NaN  ...  39.799999  4.988405
1  United States    2.0  1991.0  21991.0     NaN  ...  39.799999  4.988405
2  United States    2.0  1992.0  21992.0     NaN  ...  39.799999  4.988405
3  United States    2.0  1993.0  21993.0     NaN  ...  39.799999  4.988405
4  United States    2.0  1994.0  21994.0     NaN  ...  39.799999  4.988405

   noyrs  roflaw  nrrents
0   20.0    1.61      NaN
1   20.0    1.61      NaN
2   20.0    1.61      NaN
3   20.0    1.61      NaN
4   20.0    1.61      NaN

[5 rows x 36 columns]
```

Using a histogram, we can view the distribution of the number of billionaires per country, `numbil0`, in 2008 (the United States is dropped for plotting purposes)

```
numbil0_2008 = df[(df['year'] == 2008) & (
    df['country'] != 'United States')].loc[:, 'numbil0']

plt.subplots(figsize=(12, 8))
plt.hist(numbil0_2008, bins=30)
plt.xlim(left=0)
plt.grid()
plt.xlabel('Number of billionaires in 2008')
plt.ylabel('Count')
plt.show()
```



From the histogram, it appears that the Poisson assumption is not unreasonable (albeit with a very low μ and some outliers).

3.3 Conditional Distributions

In Treisman's paper, the dependent variable — the number of billionaires y_i in country i — is modeled as a function of GDP per capita, population size, and years membership in GATT and WTO.

Hence, the distribution of y_i needs to be conditioned on the vector of explanatory variables $\mathbf{x}_i$.

The standard formulation — the so-called *poisson regression* model — is as follows:

$$f(y_i \mid \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \qquad y_i = 0, 1, 2, \ldots, \infty. \tag{3.1}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik})$$

To illustrate the idea that the distribution of y_i depends on $\mathbf{x}_i$ let's run a simple simulation.

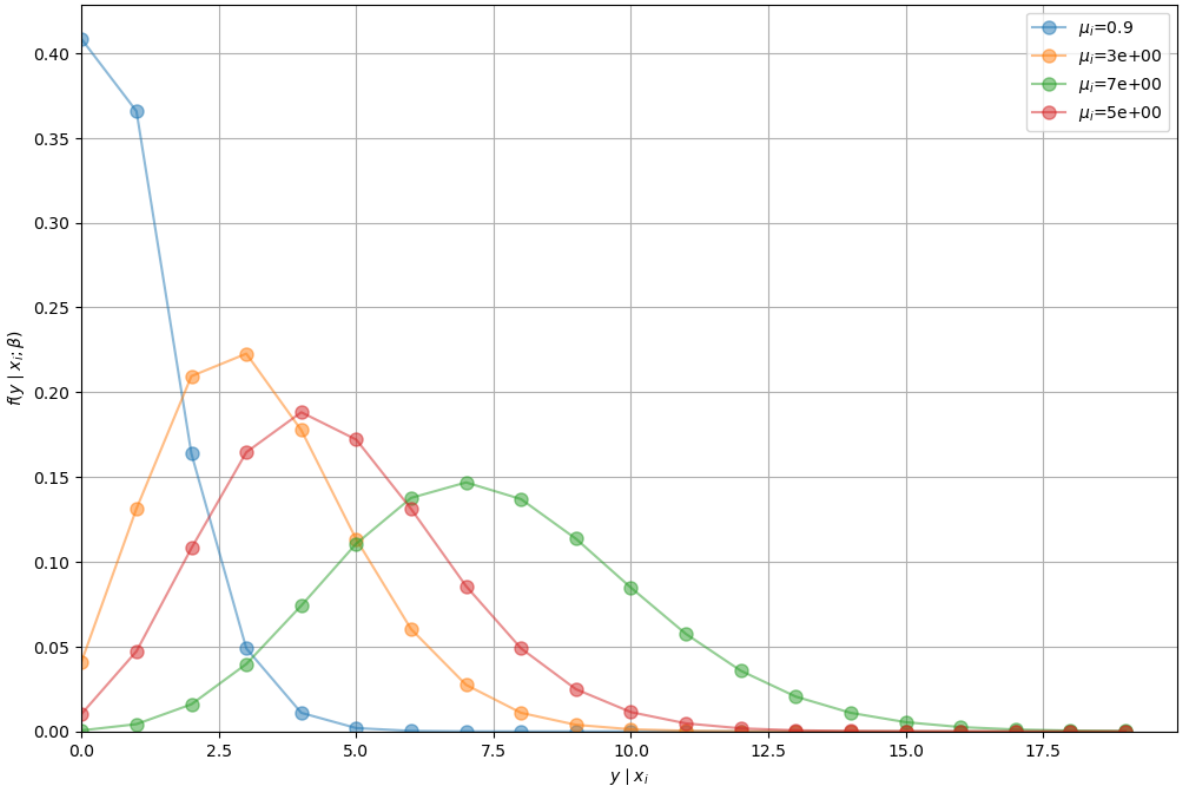
We use our `poisson_pmf` function from above and arbitrary values for β and $\mathbf{x}_i$

```
y_values = range(0, 20)

# Define a parameter vector with estimates
β = np.array([0.26, 0.18, 0.25, -0.1, -0.22])

# Create some observations X
datasets = [np.array([0, 1, 1, 1, 2]),
```

(continues on next page)

(continued from previous page)

```
            np.array([2, 3, 2, 4, 0]),
            np.array([3, 4, 5, 3, 2]),
            np.array([6, 5, 4, 4, 7])]


fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    μ = exp(X @ β)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(y_values,
            distribution,
            label=f'$\mu_i$={μ:.1}',
            marker='o',
            markersize=8,
            alpha=0.5)

ax.grid()
ax.legend()
ax.set_xlabel('$y \mid x_i$')
ax.set_ylabel(r'$f(y \mid x_i; \beta )$')
ax.axis(xmin=0, ymin=0)
plt.show()
```



We can see that the distribution of y_i is conditional on $\mathbf{x}_i$ (μ_i is no longer constant).

3.4 Maximum Likelihood Estimation

In our model for number of billionaires, the conditional distribution contains 4 ($k = 4$) parameters that we need to estimate.

We will label our entire parameter vector as β where

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

To estimate the model using MLE, we want to maximize the likelihood that our estimate $\hat{\beta}$ is the true parameter β.

Intuitively, we want to find the $\hat{\beta}$ that best fits our data.

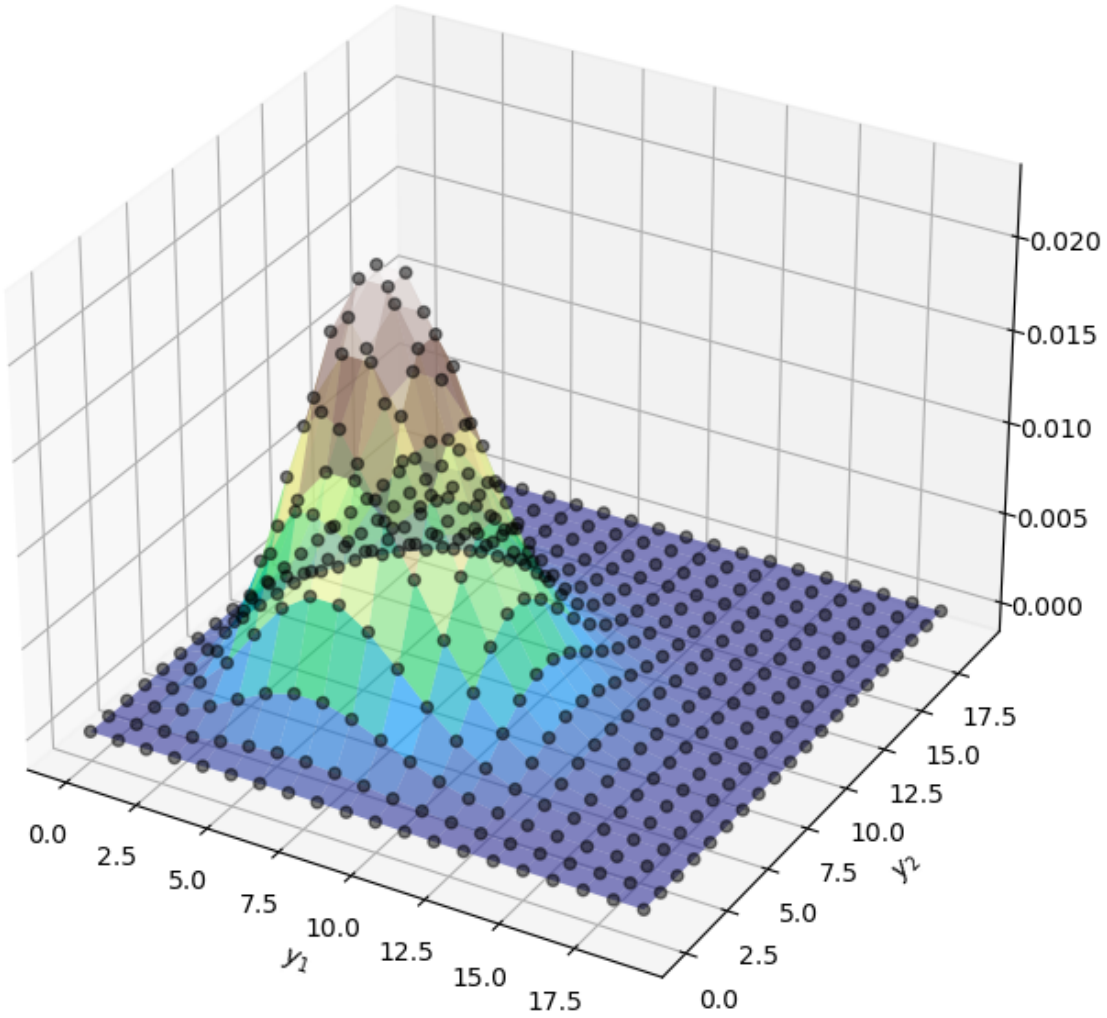
First, we need to construct the likelihood function $\mathcal{L}(\beta)$, which is similar to a joint probability density function.

Assume we have some data $y_i = \{y_1, y_2\}$ and $y_i \sim f(y_i)$.

If y_1 and y_2 are independent, the joint pmf of these data is $f(y_1, y_2) = f(y_1) \cdot f(y_2)$.

If y_i follows a Poisson distribution with $\lambda = 7$, we can visualize the joint pmf like so

```
def plot_joint_poisson(μ=7, y_n=20):
    yi_values = np.arange(0, y_n, 1)

    # Create coordinate points of X and Y
    X, Y = np.meshgrid(yi_values, yi_values)

    # Multiply distributions together
    Z = poisson_pmf(X, μ) * poisson_pmf(Y, μ)

    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z.T, cmap='terrain', alpha=0.6)
    ax.scatter(X, Y, Z.T, color='black', alpha=0.5, linewidths=1)
    ax.set(xlabel='$y_1$', ylabel='$y_2$')
    ax.set_zlabel('$f(y_1, y_2)$', labelpad=10)
    plt.show()

plot_joint_poisson(μ=7, y_n=20)
```



Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written as

$$f(y_1, y_2, \dots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

y_i is conditional on both the values of $\mathbf{x}_i$ and the parameters β.

The likelihood function is the same as the joint pmf, but treats the parameter β as a random variable and takes the observations $(y_i, \mathbf{x}_i)$ as given

$$\begin{aligned}\mathcal{L}(\beta \mid y_1, y_2, \dots, y_n \; ; \; \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) &= \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ &= f(y_1, y_2, \dots, y_n \mid \; \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta)\end{aligned}$$

Now that we have our likelihood function, we want to find the $\widehat{\beta}$ that yields the maximum likelihood value

$$\max_{\beta} \mathcal{L}(\beta)$$

In doing so it is generally easier to maximize the log-likelihood (consider differentiating $f(x) = x\exp(x)$ vs. $f(x) = \log(x) + x$).

Given that taking a logarithm is a monotone increasing transformation, a maximizer of the likelihood function will also be a maximizer of the log-likelihood function.

In our case the log-likelihood is

$$\begin{aligned}\log \mathcal{L}(\beta) &= \log\Big(f(y_1;\beta)\cdot f(y_2;\beta)\cdot ... \cdot f(y_n;\beta)\Big) \\ &= \sum_{i=1}^{n} \log f(y_i;\beta) \\ &= \sum_{i=1}^{n} \log\Big(\frac{\mu_i^{y_i}}{y_i!}e^{-\mu_i}\Big) \\ &= \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n}\mu_i - \sum_{i=1}^{n}\log y!\end{aligned}$$

The MLE of the Poisson to the Poisson for $\hat{\beta}$ can be obtained by solving

$$\max_{\beta}\Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n}\mu_i - \sum_{i=1}^{n}\log y!\Big)$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

3.5 MLE with Numerical Methods

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates.

One such numerical method is the Newton-Raphson algorithm.

Our goal is to find the maximum likelihood estimate $\hat{\beta}$.

At $\hat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

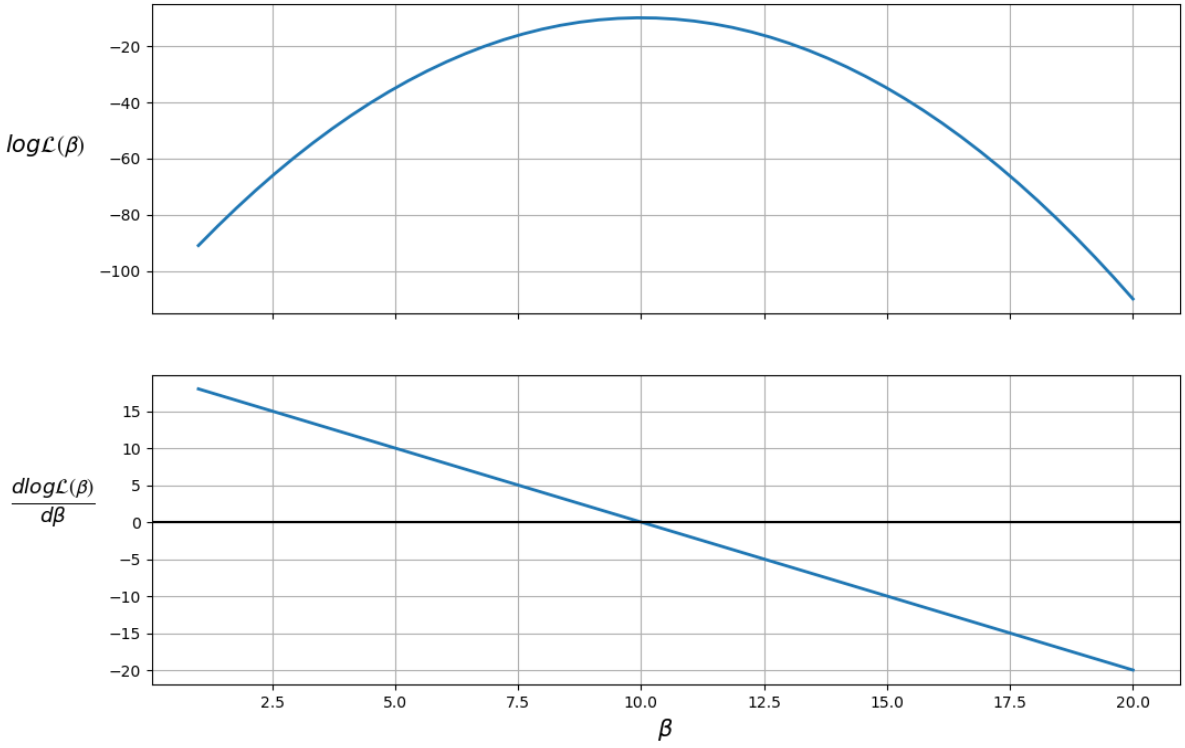
$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

```
β = np.linspace(1, 20)
logL = -(β - 10) ** 2 - 10
dlogL = -2 * β + 20

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL, lw=2)
ax2.plot(β, dlogL, lw=2)

ax1.set_ylabel(r'$log \mathcal{L(\beta)}$',
               rotation=0,
               labelpad=35,
               fontsize=15)
ax2.set_ylabel(r'$\frac{dlog \mathcal{L(\beta)}}{d \beta}$ ',
               rotation=0,
```

(continues on next page)

(continued from previous page)

```
                labelpad=35,
                fontsize=19)
ax2.set_xlabel(r'$\beta$', fontsize=15)
ax1.grid(), ax2.grid()
plt.axhline(c='black')
plt.show()
```



The plot shows that the maximum likelihood value (the top plot) occurs when $\frac{d \log \mathcal{L}(\beta)}{d\beta} = 0$ (the bottom plot).

Therefore, the likelihood is maximized when $\beta = 10$.

We can also ensure that this value is a *maximum* (as opposed to a minimum) by checking that the second derivative (slope of the bottom plot) is negative.

The Newton-Raphson algorithm finds a point where the first derivative is 0.

To use the algorithm, we take an initial guess at the maximum value, β_0 (the OLS parameter estimates might be a reasonable guess), then

1. Use the updating rule to iterate the algorithm

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}(\beta_{(k)})G(\beta_{(k)})$$

   where:

$$G(\beta_{(k)}) = \frac{d \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)}}$$

$$H(\beta_{(k)}) = \frac{d^2 \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)} d\beta'_{(k)}}$$

2. Check whether $\beta_{(k+1)} - \beta_{(k)} < tol$

- If true, then stop iterating and set $\hat{\beta} = \beta_{(k+1)}$
- If false, then update $\beta_{(k+1)}$

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ ie. where the first derivative is equal to 0.

(In practice, we stop iterating when the difference is below a small tolerance threshold)

Let's have a go at implementing the Newton-Raphson algorithm.

First, we'll create a class called `PoissonRegression` so we can easily recompute the values of the log likelihood, gradient and Hessian for every iteration

```
class PoissonRegression:

    def __init__(self, y, X, β):
        self.X = X
        self.n, self.k = X.shape
        # Reshape y as a n_by_1 column vector
        self.y = y.reshape(self.n,1)
        # Reshape β as a k_by_1 column vector
        self.β = β.reshape(self.k,1)

    def μ(self):
        return np.exp(self.X @ self.β)

    def logL(self):
        y = self.y
        μ = self.μ()
        return np.sum(y * np.log(μ) - μ - np.log(factorial(y)))

    def G(self):
        y = self.y
        μ = self.μ()
        return X.T @ (y - μ)

    def H(self):
        X = self.X
        μ = self.μ()
        return -(X.T @ (μ * X))
```

Our function `newton_raphson` will take a `PoissonRegression` object that has an initial guess of the parameter vector β_0.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

Iteration will end when either:

- The difference between the parameter and the updated parameter is below a tolerance level.
- The maximum number of iterations has been achieved (meaning convergence is not achieved).

So we can get an idea of what's going on while the algorithm is running, an option `display=True` is added to print out values at each iteration.

```
def newton_raphson(model, tol=1e-3, max_iter=1000, display=True):

    i = 0
```

(continues on next page)

(continued from previous page)

```
    error = 100  # Initial error value

    # Print header of output
    if display:
        header = f'{"Iteration_k":<13}{"Log-likelihood":<16}{"θ":<60}'
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while np.any(error > tol) and i < max_iter:
        H, G = model.H(), model.G()
        β_new = model.β - (np.linalg.inv(H) @ G)
        error = np.abs(β_new - model.β)
        model.β = β_new

        # Print iterations
        if display:
            β_list = [f'{t:.3}' for t in list(model.β.flatten())]
            update = f'{i:<13}{model.logL():<16.8}{β_list}'
            print(update)

        i += 1

    print(f'Number of iterations: {i}')
    print(f'β_hat = {model.β.flatten()}')

    # Return a flat array for β (instead of a k_by_1 column vector)
    return model.β.flatten()
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in $\mathbf{X}$.

```
X = np.array([[1, 2, 5],
              [1, 1, 3],
              [1, 4, 2],
              [1, 5, 2],
              [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
init_β = np.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(y, X, β=init_β)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪------
0             -4.3447622      ['-1.49', '0.265', '0.244']
1             -3.5742413      ['-3.38', '0.528', '0.474']
2             -3.3999526      ['-5.06', '0.782', '0.702']
```

(continues on next page)

(continued from previous page)

```
3            -3.3788646      ['-5.92', '0.909', '0.82']
4            -3.3783559      ['-6.07', '0.933', '0.843']
5            -3.3783555      ['-6.08', '0.933', '0.843']
6            -3.3783555      ['-6.08', '0.933', '0.843']
Number of iterations: 7
β_hat = [-6.07848573  0.9334028   0.84329677]
```

As this was a simple model with few observations, the algorithm achieved convergence in only 7 iterations.

You can see that with each iteration, the log-likelihood value increased.

Remember, our objective was to maximize the log-likelihood function, which the algorithm has worked to achieve.

Also, note that the increase in $\log \mathcal{L}(\beta_{(k)})$ becomes smaller with each iteration.

This is because the gradient is approaching 0 as we reach the maximum, and therefore the numerator in our updating equation is becoming smaller.

The gradient vector should be close to 0 at $\hat{\beta}$

```
poi.G()
```

```
array([[-2.54574140e-13],
       [-6.44040377e-13],
       [-4.99100761e-13]])
```

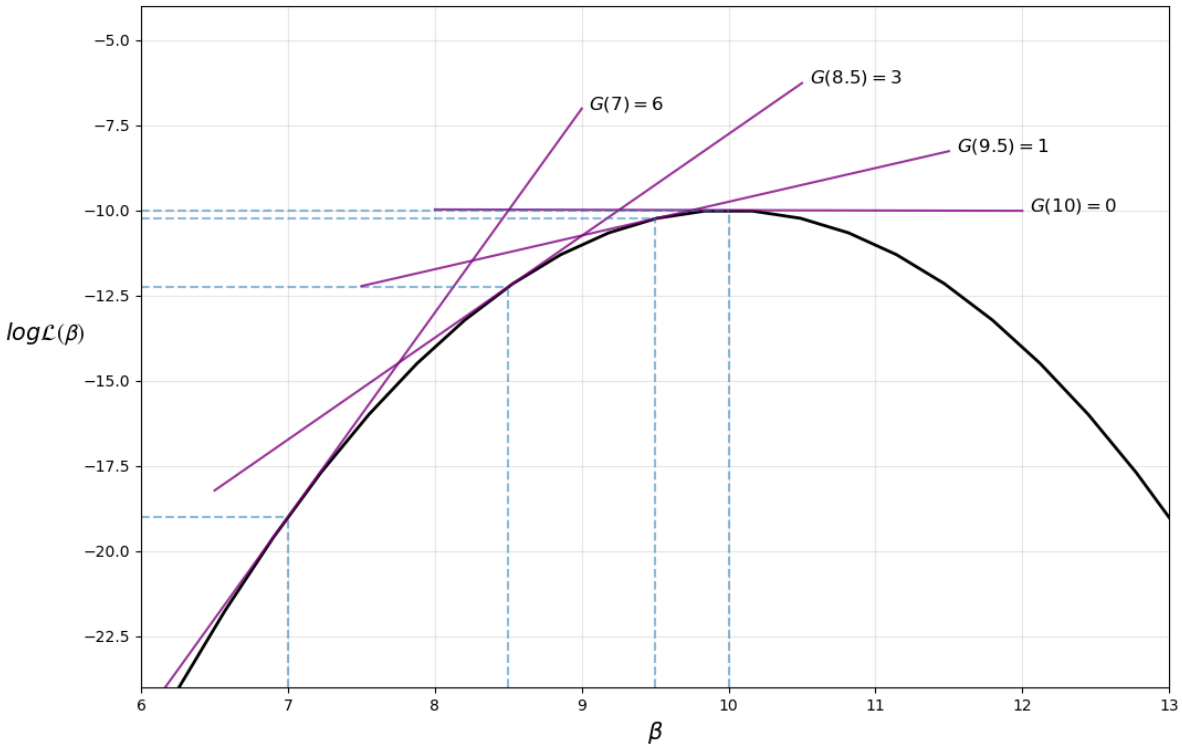
The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

```
logL = lambda x: -(x - 10) ** 2 - 10

def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β+a)
    x = np.array([[β, 1], [β+a, 1]])
    m, c = np.linalg.lstsq(x, np.array([y1, y2]), rcond=None)[0]
    return m, c

β = np.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c='black')

for β in [7, 8.5, 9.5, 10]:
    β_line = np.linspace(β-2, β+2)
    m, c = find_tangent(β)
    y = m * β_line + c
    ax.plot(β_line, y, '-', c='purple', alpha=0.8)
    ax.text(β+2.05, y[-1], f'$G({β}) = {abs(m):.0f}$', fontsize=12)
    ax.vlines(β, -24, logL(β), linestyles='--', alpha=0.5)
    ax.hlines(logL(β), 6, β, linestyles='--', alpha=0.5)

ax.set(ylim=(-24, -4), xlim=(6, 13))
ax.set_xlabel(r'$\beta$', fontsize=15)
ax.set_ylabel(r'$log \mathcal{L(\beta)}$',
              rotation=0,
              labelpad=25,
              fontsize=15)
```

(continues on next page)

(continued from previous page)

```
ax.grid(alpha=0.3)
plt.show()
```



Note that our implementation of the Newton-Raphson algorithm is rather basic — for more robust implementations see, for example, scipy.optimize.

3.6 Maximum Likelihood Estimation with `statsmodels`

Now that we know what's going on under the hood, we can apply MLE to an interesting application.

We'll use the Poisson regression model in `statsmodels` to obtain a richer output with standard errors, test values, and more.

`statsmodels` uses the same algorithm as above to find the maximum likelihood estimates.

Before we begin, let's re-estimate our simple model with `statsmodels` to confirm we obtain the same coefficients and log-likelihood value.

```
X = np.array([[1, 2, 5],
              [1, 1, 3],
              [1, 4, 2],
              [1, 5, 2],
              [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

stats_poisson = Poisson(y, X).fit()
print(stats_poisson.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.675671
         Iterations 7
                          Poisson Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                        Poisson   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Tue, 30 Jan 2024   Pseudo R-squ.:                  0.2546
Time:                        06:44:48   Log-Likelihood:                -3.3784
converged:                       True   LL-Null:                       -4.5325
Covariance Type:            nonrobust   LLR p-value:                    0.3153
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.0785      5.279     -1.151      0.250     -16.425       4.268
x1             0.9334      0.829      1.126      0.260      -0.691       2.558
x2             0.8433      0.798      1.057      0.291      -0.720       2.407
==============================================================================
```

Now let's replicate results from Daniel Treisman's paper, Russia's Billionaires, mentioned earlier in the lecture.

Treisman starts by estimating equation (3.1), where:

- y_i is $number\ of\ billionaires_i$
- x_{i1} is $\log GDP\ per\ capita_i$
- x_{i2} is $\log population_i$
- x_{i3} is $years\ in\ GATT_i$ – years membership in GATT and WTO (to proxy access to international markets)

The paper only considers the year 2008 for estimation.

We will set up our variables for estimation like so (you should have the data assigned to `df` from earlier in the lecture)

```
# Keep only year 2008
df = df[df['year'] == 2008]

# Add a constant
df['const'] = 1

# Variable sets
reg1 = ['const', 'lngdppc', 'lnpop', 'gattwto08']
reg2 = ['const', 'lngdppc', 'lnpop',
        'gattwto08', 'lnmcap08', 'rintr', 'topint08']
reg3 = ['const', 'lngdppc', 'lnpop', 'gattwto08', 'lnmcap08',
        'rintr', 'topint08', 'nrrents', 'roflaw']
```

Then we can use the `Poisson` function from `statsmodels` to fit the model.

We'll use robust standard errors as in the author's paper

```
# Specify model
poisson_reg = sm.Poisson(df[['numbil0']], df[reg1],
                         missing='drop').fit(cov_type='HC0')
print(poisson_reg.summary())
```

```
Optimization terminated successfully.
         Current function value: 2.226090
         Iterations 9
                          Poisson Regression Results
==============================================================================
Dep. Variable:                 numbil0   No. Observations:                  197
Model:                         Poisson   Df Residuals:                      193
Method:                            MLE   Df Model:                            3
Date:                Tue, 30 Jan 2024   Pseudo R-squ.:                  0.8574
Time:                        06:44:48   Log-Likelihood:                -438.54
converged:                       True   LL-Null:                       -3074.7
Covariance Type:                  HC0   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -29.0495      2.578    -11.268      0.000     -34.103     -23.997
lngdppc        1.0839      0.138      7.834      0.000       0.813       1.355
lnpop          1.1714      0.097     12.024      0.000       0.980       1.362
gattwto08      0.0060      0.007      0.868      0.386      -0.008       0.019
==============================================================================
```

Success! The algorithm was able to achieve convergence in 9 iterations.

Our output indicates that GDP per capita, population, and years of membership in the General Agreement on Tariffs and Trade (GATT) are positively related to the number of billionaires a country has, as expected.

Let's also estimate the author's more full-featured models and display them in a single table

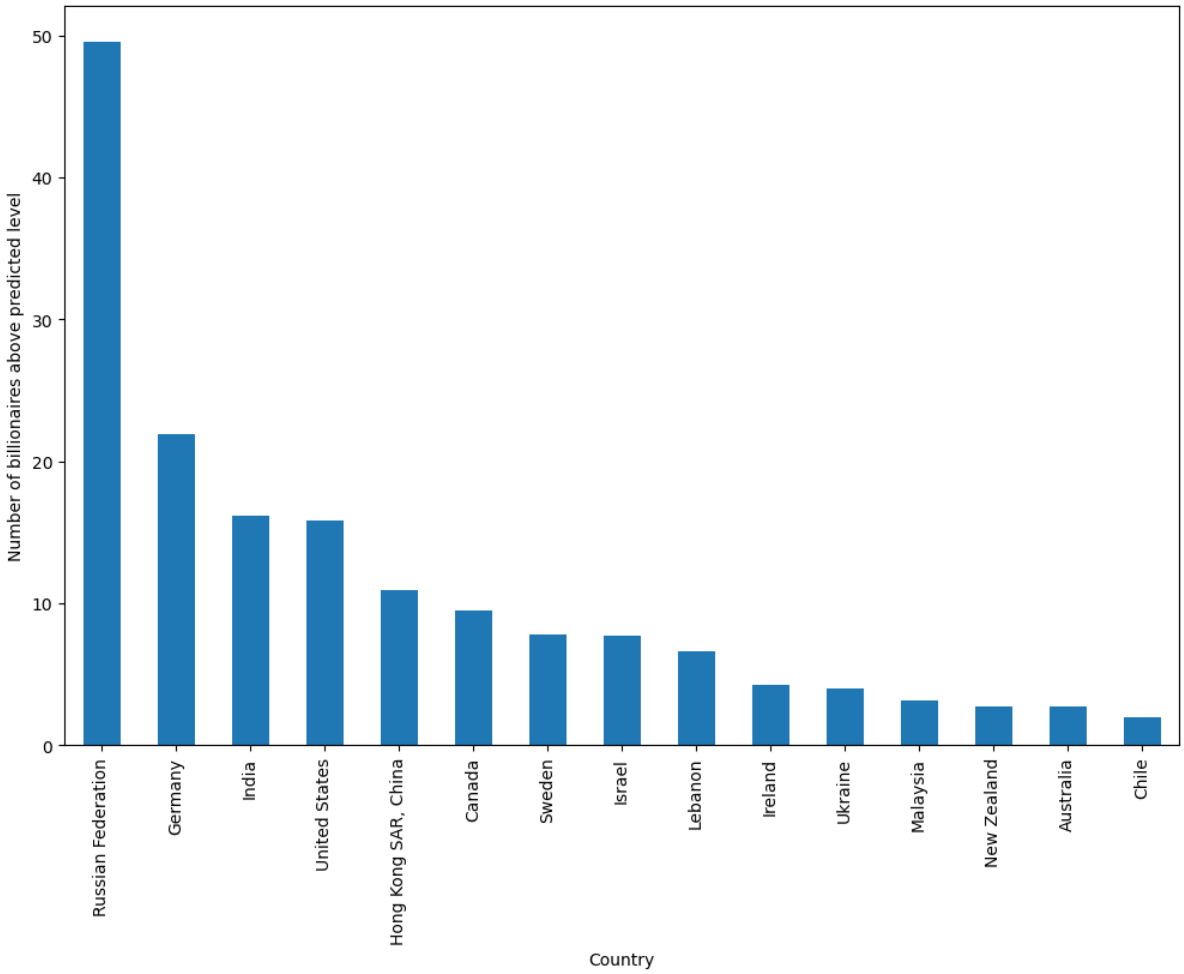
```
regs = [reg1, reg2, reg3]
reg_names = ['Model 1', 'Model 2', 'Model 3']
info_dict = {'Pseudo R-squared': lambda x: f"{x.prsquared:.2f}",
             'No. observations': lambda x: f"{int(x.nobs):d}"}
regressor_order = ['const',
                   'lngdppc',
                   'lnpop',
                   'gattwto08',
                   'lnmcap08',
                   'rintr',
                   'topint08',
                   'nrrents',
                   'roflaw']
results = []

for reg in regs:
    result = sm.Poisson(df[['numbil0']], df[reg],
                        missing='drop').fit(cov_type='HC0',
                                            maxiter=100, disp=0)
    results.append(result)

results_table = summary_col(results=results,
                            float_format='%0.3f',
                            stars=True,
                            model_names=reg_names,
                            info_dict=info_dict,
                            regressor_order=regressor_order)
results_table.add_title('Table 1 - Explaining the Number of Billionaires \
                        in 2008')
print(results_table)
```

```
Table 1 - Explaining the Number of Billionaires                          in 2008
================================================
                  Model 1    Model 2    Model 3
------------------------------------------------
const            -29.050*** -19.444*** -20.858***
                 (2.578)    (4.820)    (4.255)
lngdppc          1.084***   0.717***   0.737***
                 (0.138)    (0.244)    (0.233)
lnpop            1.171***   0.806***   0.929***
                 (0.097)    (0.213)    (0.195)
gattwto08        0.006      0.007      0.004
                 (0.007)    (0.006)    (0.006)
lnmcap08                    0.399**    0.286*
                            (0.172)    (0.167)
rintr                       -0.010     -0.009
                            (0.010)    (0.010)
topint08                    -0.051***  -0.058***
                            (0.011)    (0.012)
nrrents                                -0.005
                                       (0.010)
roflaw                                 0.203
                                       (0.372)
Pseudo R-squared 0.86       0.90       0.90
No. observations 197        131        131
================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

The output suggests that the frequency of billionaires is positively correlated with GDP per capita, population size, stock market capitalization, and negatively correlated with top marginal income tax rate.

To analyze our results by country, we can plot the difference between the predicted an actual values, then sort from highest to lowest and plot the first 15

```
data = ['const', 'lngdppc', 'lnpop', 'gattwto08', 'lnmcap08', 'rintr',
        'topint08', 'nrrents', 'roflaw', 'numbil0', 'country']
results_df = df[data].dropna()

# Use last model (model 3)
results_df['prediction'] = results[-1].predict()

# Calculate difference
results_df['difference'] = results_df['numbil0'] - results_df['prediction']

# Sort in descending order
results_df.sort_values('difference', ascending=False, inplace=True)

# Plot the first 15 data points
results_df[:15].plot('country', 'difference', kind='bar',
                     figsize=(12,8), legend=False)
plt.ylabel('Number of billionaires above predicted level')
plt.xlabel('Country')
plt.show()
```



As we can see, Russia has by far the highest number of billionaires in excess of what is predicted by the model (around 50 more than expected).

Treisman uses this empirical result to discuss possible reasons for Russia's excess of billionaires, including the origination of wealth in Russia, the political climate, and the history of privatization in the years after the USSR.

3.7 Summary

In this lecture, we used Maximum Likelihood Estimation to estimate the parameters of a Poisson model.

`statsmodels` contains other built-in likelihood models such as Probit and Logit.

For further flexibility, `statsmodels` provides a way to specify the distribution manually using the `GenericLikelihoodModel` class - an example notebook can be found here.

3.8 Exercises

Exercise 3.8.1

Suppose we wanted to estimate the probability of an event y_i occurring, given some observations.

We could use a probit regression model, where the pmf of y_i is

$$f(y_i; \beta) = \mu_i^{y_i}(1-\mu_i)^{1-y_i}, \quad y_i = 0, 1$$
$$\text{where} \quad \mu_i = \Phi(\mathbf{x}_i'\beta)$$

Φ represents the *cumulative normal distribution* and constrains the predicted y_i to be between 0 and 1 (as required for a probability).

β is a vector of coefficients.

Following the example in the lecture, write a class to represent the Probit model.

To begin, find the log-likelihood function and derive the gradient and Hessian.

The `scipy` module `stats.norm` contains the functions needed to compute the cmf and pmf of the normal distribution.

Solution to Exercise 3.8.1

The log-likelihood can be written as

$$\log \mathcal{L} = \sum_{i=1}^{n} \left[y_i \log \Phi(\mathbf{x}_i'\beta) + (1-y_i) \log(1 - \Phi(\mathbf{x}_i'\beta)) \right]$$

Using the **fundamental theorem of calculus**, the derivative of a cumulative probability distribution is its marginal distribution

$$\frac{\partial}{\partial s}\Phi(s) = \phi(s)$$

where ϕ is the marginal normal distribution.

The gradient vector of the Probit model is

$$\frac{\partial \log \mathcal{L}}{\partial \beta} = \sum_{i=1}^{n} \Big[y_i \frac{\phi(\mathbf{x}_i'\beta)}{\Phi(\mathbf{x}_i'\beta)} - (1-y_i) \frac{\phi(\mathbf{x}_i'\beta)}{1 - \Phi(\mathbf{x}_i'\beta)} \Big] \mathbf{x}_i$$

The Hessian of the Probit model is

$$\frac{\partial^2 \log \mathcal{L}}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \phi(\mathbf{x}_i'\beta) \Big[y_i \frac{\phi(\mathbf{x}_i'\beta) + \mathbf{x}_i'\beta\Phi(\mathbf{x}_i'\beta)}{[\Phi(\mathbf{x}_i'\beta)]^2} + (1-y_i) \frac{\phi(\mathbf{x}_i'\beta) - \mathbf{x}_i'\beta(1 - \Phi(\mathbf{x}_i'\beta))}{[1 - \Phi(\mathbf{x}_i'\beta)]^2} \Big] \mathbf{x}_i\mathbf{x}_i'$$

Using these results, we can write a class for the Probit model as follows

```
class ProbitRegression:

    def __init__(self, y, X, β):
        self.X, self.y, self.β = X, y, β
        self.n, self.k = X.shape

    def μ(self):
        return norm.cdf(self.X @ self.β.T)
```

(continues on next page)

(continued from previous page)

```
    def ϕ(self):
        return norm.pdf(self.X @ self.β.T)

    def logL(self):
        μ = self.μ()
        return np.sum(y * np.log(μ) + (1 - y) * np.log(1 - μ))

    def G(self):
        μ = self.μ()
        ϕ = self.ϕ()
        return np.sum((X.T * y * ϕ / μ - X.T * (1 - y) * ϕ / (1 - μ)),
                      axis=1)

    def H(self):
        X = self.X
        β = self.β
        μ = self.μ()
        ϕ = self.ϕ()
        a = (ϕ + (X @ β.T) * μ) / μ**2
        b = (ϕ - (X @ β.T) * (1 - μ)) / (1 - μ)**2
        return -(ϕ * (y * a + (1 - y) * b) * X.T) @ X
```

Exercise 3.8.2

Use the following dataset and initial values of β to estimate the MLE with the Newton-Raphson algorithm developed earlier in the lecture

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 1 & 1 \\ 1 & 4 & 3 \\ 1 & 5 & 6 \\ 1 & 3 & 5 \end{bmatrix} \quad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \beta_{(0)} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}$$

Verify your results with `statsmodels` - you can import the Probit function with the following import statement

```
from statsmodels.discrete.discrete_model import Probit
```

Note that the simple Newton-Raphson algorithm developed in this lecture is very sensitive to initial values, and therefore you may fail to achieve convergence with different starting values.

Solution to Exercise 3.8.2

Here is one solution

```
X = np.array([[1, 2, 4],
              [1, 1, 1],
              [1, 4, 3],
              [1, 5, 6],
              [1, 3, 5]])

y = np.array([1, 0, 1, 1, 0])
```

(continues on next page)

(continued from previous page)

```
# Take a guess at initial βs
β = np.array([0.1, 0.1, 0.1])

# Create instance of Probit regression class
prob = ProbitRegression(y, X, β)

# Run Newton-Raphson algorithm
newton_raphson(prob)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
 ↪------
0            -2.3796884      ['-1.34', '0.775', '-0.157']
1            -2.3687526      ['-1.53', '0.775', '-0.0981']
2            -2.3687294      ['-1.55', '0.778', '-0.0971']
3            -2.3687294      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.54625858  0.77778952 -0.09709757]
```

```
array([-1.54625858,  0.77778952, -0.09709757])
```

```
# Use statsmodels to verify results

print(Probit(y, X).fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.473746
         Iterations 6
                          Probit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                         Probit   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Tue, 30 Jan 2024   Pseudo R-squ.:                  0.2961
Time:                        06:44:49   Log-Likelihood:                -2.3687
converged:                       True   LL-Null:                       -3.3651
Covariance Type:            nonrobust   LLR p-value:                    0.3692
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.5463      1.866     -0.829      0.407      -5.204       2.111
x1             0.7778      0.788      0.986      0.324      -0.768       2.323
x2            -0.0971      0.590     -0.165      0.869      -1.254       1.060
==============================================================================
```

CHAPTER

FOUR

ELEMENTARY PROBABILITY WITH MATRICES

This lecture uses matrix algebra to illustrate some basic ideas about probability theory.

After providing somewhat informal definitions of the underlying objects, we'll use matrices and vectors to describe probability distributions.

Among concepts that we'll be studying include

- a joint probability distribution
- marginal distributions associated with a given joint distribution
- conditional probability distributions
- statistical independence of two random variables
- joint distributions associated with a prescribed set of marginal distributions
  - couplings
  - copulas
- the probability distribution of a sum of two independent random variables
  - convolution of marginal distributions
- parameters that define a probability distribution
- sufficient statistics as data summaries

We'll use a matrix to represent a bivariate probability distribution and a vector to represent a univariate probability distribution

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install prettytable
```

As usual, we'll start with some imports

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('retina')
```

4.1 Sketch of Basic Concepts

We'll briefly define what we mean by a **probability space**, a **probability measure**, and a **random variable**.

For most of this lecture, we sweep these objects into the background, but they are there underlying the other objects that we'll mainly focus on.

Let Ω be a set of possible underlying outcomes and let $\omega \in \Omega$ be a particular underlying outcomes.

Let $\mathcal{G} \subset \Omega$ be a subset of Ω.

Let $\mathcal{F}$ be a collection of such subsets $\mathcal{G} \subset \Omega$.

The pair $\Omega, \mathcal{F}$ forms our **probability space** on which we want to put a probability measure.

A **probability measure** μ maps a set of possible underlying outcomes $\mathcal{G} \in \mathcal{F}$ into a scalar number between 0 and 1

- this is the "probability" that X belongs to A, denoted by Prob$\{X \in A\}$.

A **random variable** $X(\omega)$ is a function of the underlying outcome $\omega \in \Omega$.

The random variable $X(\omega)$ has a **probability distribution** that is induced by the underlying probability measure μ and the function $X(\omega)$:

$$\text{Prob}(X \in A) = \int_{\mathcal{G}} \mu(\omega) d\omega \tag{4.1}$$

where $\mathcal{G}$ is the subset of Ω for which $X(\omega) \in A$.

We call this the induced probability distribution of random variable X.

4.2 What Does Probability Mean?

Before diving in, we'll say a few words about what probability theory means and how it connects to statistics.

We also touch on these topics in the quantecon lectures https://python.quantecon.org/prob_meaning.html and https://python.quantecon.org/navy_captain.html.

For much of this lecture we'll be discussing fixed "population" probabilities.

These are purely mathematical objects.

To appreciate how statisticians connect probabilities to data, the key is to understand the following concepts:

- A single draw from a probability distribution
- Repeated independently and identically distributed (i.i.d.) draws of "samples" or "realizations" from the same probability distribution
- A **statistic** defined as a function of a sequence of samples
- An **empirical distribution** or **histogram** (a binned empirical distribution) that records observed **relative frequencies**
- The idea that a population probability distribution is what we anticipate **relative frequencies** will be in a long sequence of i.i.d. draws. Here the following mathematical machinery makes precise what is meant by **anticipated relative frequencies**
  - **Law of Large Numbers (LLN)**
  - **Central Limit Theorem (CLT)**

Scalar example

Consider the following discrete distribution

$$X \sim \{f_i\}_{i=0}^{I-1}, \quad f_i \geqslant 0, \quad \sum_i f_i = 1$$

Draw a sample $x_0, x_1, \ldots, x_{N-1}$, N draws of X from $\{f_i\}_{i=1}^{I}$.

What do the "identical" and "independent" mean in IID or iid ("identically and independently distributed)?

- "identical" means that each draw is from the same distribution.
- "independent" means that the joint distribution equal tthe product of marginal distributions, i.e.,

$$\begin{aligned} \text{Prob}\{x_0 = i_0, x_1 = i_1, \ldots, x_{N-1} = i_{N-1}\} &= \text{Prob}\{x_0 = i_0\} \cdot \cdots \cdot \text{Prob}\{x_{I-1} = i_{I-1}\} \\ &= f_{i_0} f_{i_1} \cdot \cdots \cdot f_{i_{N-1}} \end{aligned}$$

Consider the **empirical distribution**:

$$\begin{aligned} i &= 0, \ldots, I-1, \\ N_i &= \text{number of times } X = i, \\ N &= \sum_{i=0}^{I-1} N_i \quad \text{total number of draws}, \\ \tilde{f}_i &= \frac{N_i}{N} \sim \text{ frequency of draws for which } X = i \end{aligned}$$

Key ideas that justify connecting probability theory with statistics are laws of large numbers and central limit theorems

LLN:

- A Law of Large Numbers (LLN) states that $\tilde{f}_i \to f_i$ as $N \to \infty$

CLT:

- A Central Limit Theorem (CLT) describes a **rate** at which $\tilde{f}_i \to f_i$

Remarks

- For "frequentist" statisticians, **anticipated relative frequency** is **all** that a probability distribution means.
- But for a Bayesian it means something more or different.

4.3 Representing Probability Distributions

A probability distribution $\text{Prob}(X \in A)$ can be described by its **cumulative distribution function (CDF)**

$$F_X(x) = \text{Prob}\{X \leq x\}.$$

Sometimes, but not always, a random variable can also be described by **density function** $f(x)$ that is related to its CDF by

$$\text{Prob}\{X \in B\} = \int_{t \in B} f(t)dt$$

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

Here B is a set of possible X's whose probability we want to compute.

When a probability density exists, a probability distribution can be characterized either by its CDF or by its density.

For a **discrete-valued** random variable

- the number of possible values of X is finite or countably infinite
- we replace a **density** with a **probability mass function**, a non-negative sequence that sums to one
- we replace integration with summation in the formula like (4.1) that relates a CDF to a probability mass function

In this lecture, we mostly discuss discrete random variables.

Doing this enables us to confine our tool set basically to linear algebra.

Later we'll briefly discuss how to approximate a continuous random variable with a discrete random variable.

4.4 Univariate Probability Distributions

We'll devote most of this lecture to discrete-valued random variables, but we'll say a few things about continuous-valued random variables.

4.4.1 Discrete random variable

Let X be a discrete random variable that takes possible values: $i = 0, 1, \ldots, I-1 = \bar{X}$.

Here, we choose the maximum index $I-1$ because of how this aligns nicely with Python's index convention.

Define $f_i \equiv \text{Prob}\{X = i\}$ and assemble the non-negative vector

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{I-1} \end{bmatrix} \tag{4.2}$$

for which $f_i \in [0, 1]$ for each i and $\sum_{i=0}^{I-1} f_i = 1$.

This vector defines a **probability mass function**.

The distribution (4.2) has **parameters** $\{f_i\}_{i=0,1,\cdots,I-2}$ since $f_{I-1} = 1 - \sum_{i=0}^{I-2} f_i$.

These parameters pin down the shape of the distribution.

(Sometimes $I = \infty$.)

Such a "non-parametric" distribution has as many "parameters" as there are possible values of the random variable.

We often work with special distributions that are characterized by a small number parameters.

In these special parametric distributions,

$$f_i = g(i; \theta)$$

where θ is a vector of parameters that is of much smaller dimension than I.

Remarks:

- The concept of **parameter** is intimately related to the notion of **sufficient statistic**.
- Sufficient statistics are nonlinear functions of a data set.
- Sufficient statistics are designed to summarize all **information** about parameters that is contained in a data set.
- They are important tools that AI uses to summarize a **big data** set
- R. A. Fisher provided a rigorous definition of **information** – see https://en.wikipedia.org/wiki/Fisher_information

An example of a parametric probability distribution is a **geometric distribution**.

It is described by

$$f_i = \text{Prob}\{X = i\} = (1 - \lambda)\lambda^i, \quad \lambda \in [0, 1], \quad i = 0, 1, 2, ...$$

Evidently, $\sum_{i=0}^{\infty} f_i = 1$.

Let θ be a vector of parameters of the distribution described by f, then

$$f_i(\theta) \geq 0, \sum_{i=0}^{\infty} f_i(\theta) = 1$$

4.4.2 Continuous random variable

Let X be a continous random variable that takes values $X \in \tilde{X} \equiv [X_U, X_L]$ whose distributions have parameters θ.

$$\text{Prob}\{X \in A\} = \int_{x \in A} f(x; \theta)\, dx; \quad f(x; \theta) \geq 0$$

where A is a subset of $\tilde{X}$ and

$$\text{Prob}\{X \in \tilde{X}\} = 1$$

4.5 Bivariate Probability Distributions

We'll now discuss a bivariate **joint distribution**.

To begin, we restrict ourselves to two discrete random variables.

Let X, Y be two discrete random variables that take values:

$$X \in \{0, ..., I - 1\}$$

$$Y \in \{0, ..., J - 1\}$$

Then their **joint distribution** is described by a matrix

$$F_{I \times J} = [f_{ij}]_{i \in \{0,...,I-1\}, j \in \{0,...,J-1\}}$$

whose elements are

$$f_{ij} = \text{Prob}\{X = i, Y = j\} \geq 0$$

where

$$\sum_i \sum_j f_{ij} = 1$$

4.6 Marginal Probability Distributions

The joint distribution induce marginal distributions

$$\text{Prob}\{X = i\} = \sum_{j=0}^{J-1} f_{ij} = \mu_i, \quad i = 0, \dots, I-1$$

$$\text{Prob}\{Y = j\} = \sum_{i=0}^{I-1} f_{ij} = \nu_j, \quad j = 0, \dots, J-1$$

For example, let a joint distribution over (X, Y) be

$$F = \begin{bmatrix} .25 & .1 \\ .15 & .5 \end{bmatrix} \tag{4.3}$$

The implied marginal distributions are:

$$\begin{aligned} \text{Prob}\{X = 0\} &= .25 + .1 = .35 \\ \text{Prob}\{X = 1\} &= .15 + .5 = .65 \\ \text{Prob}\{Y = 0\} &= .25 + .15 = .4 \\ \text{Prob}\{Y = 1\} &= .1 + .5 = .6 \end{aligned}$$

Digression: If two random variables X, Y are continuous and have joint density $f(x, y)$, then marginal distributions can be computed by

$$\begin{aligned} f(x) &= \int_{\mathbb{R}} f(x, y) dy \\ f(y) &= \int_{\mathbb{R}} f(x, y) dx \end{aligned}$$

4.7 Conditional Probability Distributions

Conditional probabilities are defined according to

$$\text{Prob}\{A \mid B\} = \frac{\text{Prob}\{A \cap B\}}{\text{Prob}\{B\}}$$

where A, B are two events.

For a pair of discrete random variables, we have the **conditional distribution**

$$\text{Prob}\{X = i | Y = j\} = \frac{f_{ij}}{\sum_i f_{ij}} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}}$$

where $i = 0, \dots, I-1, \quad j = 0, \dots, J-1$.

Note that

$$\sum_i \text{Prob}\{X_i = i | Y_j = j\} = \frac{\sum_i f_{ij}}{\sum_i f_{ij}} = 1$$

Remark: The mathematics of conditional probability implies **Bayes' Law**:

$$\text{Prob}\{X = i | Y = j\} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}} = \frac{\text{Prob}\{Y = j | X = i\} \text{Prob}\{X = i\}}{\text{Prob}\{Y = j\}}$$

For the joint distribution (4.3)

$$\text{Prob}\{X = 0 | Y = 1\} = \frac{.1}{.1 + .5} = \frac{.1}{.6}$$

4.8 Statistical Independence

Random variables X and Y are statistically **independent** if

$$\text{Prob}\{X = i, Y = j\} = f_i g_j$$

where

$$\text{Prob}\{X = i\} = f_i \geq 0 \ \sum f_i = 1$$
$$\text{Prob}\{Y = j\} = g_j \geq 0 \ \sum g_j = 1$$

Conditional distributions are

$$\text{Prob}\{X = i|Y = j\} = \frac{f_i g_j}{\sum_i f_i g_j} = \frac{f_i g_j}{g_i} = f_i$$
$$\text{Prob}\{Y = j|X = i\} = \frac{f_i g_j}{\sum_j f_i g_j} = \frac{f_i g_j}{f_i} = g_j$$

4.9 Means and Variances

The mean and variance of a discrete random variable X are

$$\mu_X \equiv \mathbb{E}[X] = \sum_k k\text{Prob}\{X = k\}$$
$$\sigma_X^2 \equiv \mathbb{D}[X] = \sum_k (k - \mathbb{E}[X])^2 \text{Prob}\{X = k\}$$

A continuous random variable having density $f_X(x)$) has mean and variance

$$\mu_X \equiv \mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx$$
$$\sigma_X^2 \equiv \mathbb{D}[X] = \text{E}\left[(X - \mu_X)^2\right] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) dx$$

4.10 Generating Random Numbers

Suppose we have at our disposal a pseudo random number that draws a uniform random variable, i.e., one with probability distribution

$$\text{Prob}\{\tilde{X} = i\} = \frac{1}{I}, \quad i = 0, \ldots, I - 1$$

How can we transform $\tilde{X}$ to get a random variable X for which $\text{Prob}\{X = i\} = f_i, \quad i = 0, \ldots, I - 1$, where f_i is an arbitary discrete probability distribution on $i = 0, 1, \ldots, I - 1$?

The key tool is the inverse of a cumulative distribution function (CDF).

Observe that the CDF of a distribution is monotone and non-decreasing, taking values between 0 and 1.

We can draw a sample of a random variable X with a known CDF as follows:

- draw a random variable u from a uniform distribution on $[0, 1]$
- pass the sample value of u into the **"inverse"** target CDF for X

- X has the target CDF

Thus, knowing the **"inverse"** CDF of a distribution is enough to simulate from this distribution.

Note: The "inverse" CDF needs to exist for this method to work.

The inverse CDF is

$$F^{-1}(u) \equiv \inf\{x \in \mathbb{R} : F(x) \geq u\} \quad (0 < u < 1)$$

Here we use infimum because a CDF is a non-decreasing and right-continuous function.

Thus, suppose that

- U is a uniform random variable $U \in [0, 1]$
- We want to sample a random variable X whose CDF is F.

It turns out that if we use draw uniform random numbers U and then compute X from

$$X = F^{-1}(U),$$

then X is a random variable with CDF $F_X(x) = F(x) = \text{Prob}\{X \leq x\}$.

We'll verify this in the special case in which F is continuous and bijective so that its inverse function exists and can be denoted by F^{-1}.

Note that

$$\begin{aligned} F_X(x) &= \text{Prob}\{X \leq x\} \\ &= \text{Prob}\left\{F^{-1}(U) \leq x\right\} \\ &= \text{Prob}\{U \leq F(x)\} \\ &= F(x) \end{aligned}$$

where the last equality occurs because U is distributed uniformly on $[0, 1]$ while $F(x)$ is a constant given x that also lies on $[0, 1]$.

Let's use `numpy` to compute some examples.

Example: A continuous geometric (exponential) distribution

Let X follow a geometric distribution, with parameter $\lambda > 0$.

Its density function is

$$f(x) = \lambda e^{-\lambda x}$$

Its CDF is

$$F(x) = \int_0^\infty \lambda e^{-\lambda x} = 1 - e^{-\lambda x}$$

Let U follow a uniform distribution on $[0, 1]$.

X is a random variable such that $U = F(X)$.

The distribution X can be deduced from

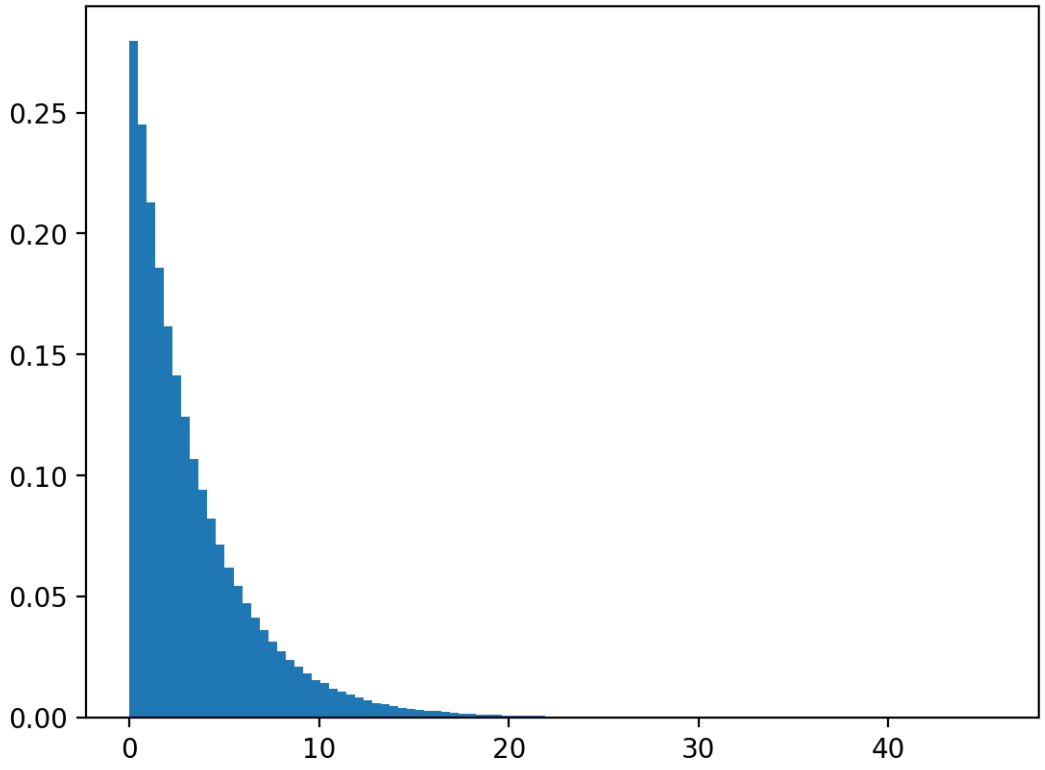
$$\begin{aligned} U = F(X) &= 1 - e^{-\lambda X} \\ \implies \quad -U &= e^{-\lambda X} \\ \implies \quad \log(1-U) &= -\lambda X \\ \implies \quad X &= \frac{(1-U)}{-\lambda} \end{aligned}$$

Let's draw u from $U[0, 1]$ and calculate $x = \frac{log(1-U)}{-\lambda}$.

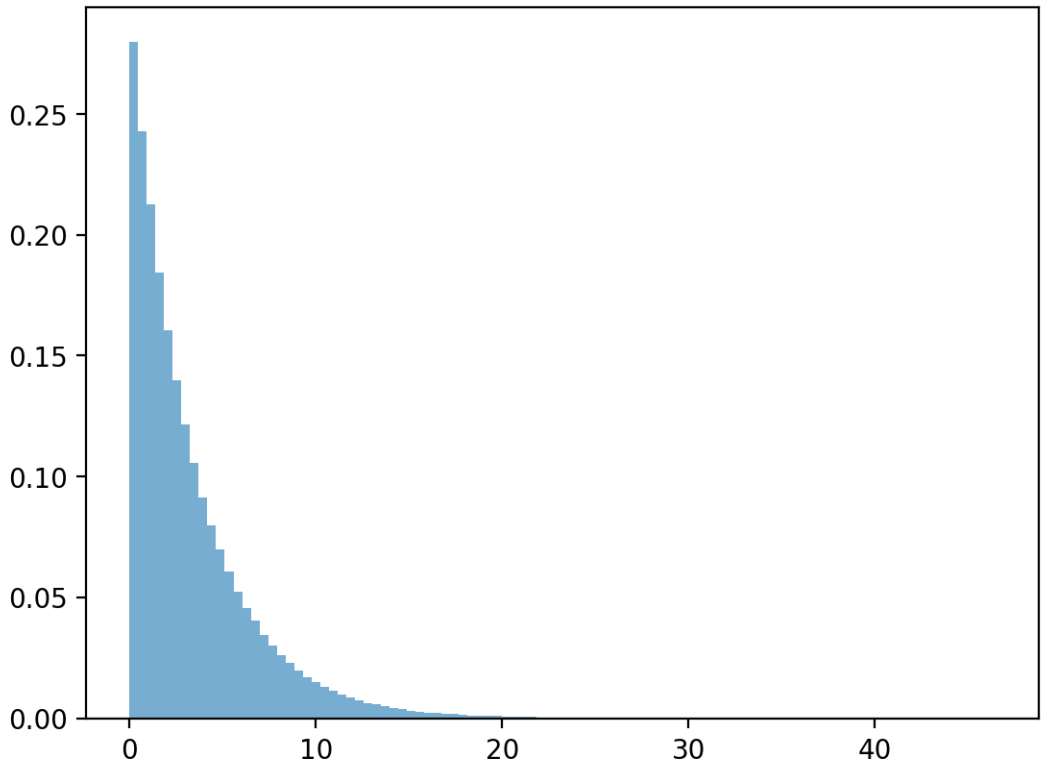
We'll check whether X seems to follow a **continuous geometric** (exponential) distribution.

Let's check with `numpy`.

```
n, λ = 1_000_000, 0.3

# draw uniform numbers
u = np.random.rand(n)

# transform
x = -np.log(1-u)/λ

# draw geometric distributions
x_g = np.random.exponential(1 / λ, n)

# plot and compare
plt.hist(x, bins=100, density=True)
plt.show()
```




```
plt.hist(x_g, bins=100, density=True, alpha=0.6)
plt.show()
```



Geometric distribution

Let X distributed geometrically, that is

$$\text{Prob}(X = i) = (1 - \lambda)\lambda^i, \quad \lambda \in (0, 1), \quad i = 0, 1, ...$$

$$\sum_{i=0}^{\infty} \text{Prob}(X = i) = 1 \longleftrightarrow (1 - \lambda) \sum_{i=0}^{\infty} \lambda^i = \frac{1 - \lambda}{1 - \lambda} = 1$$

Its CDF is given by

$$\begin{aligned} \text{Prob}(X \leq i) &= (1 - \lambda) \sum_{j=0}^{i} \lambda^i \\ &= (1 - \lambda)[\frac{1 - \lambda^{i+1}}{1 - \lambda}] \\ &= 1 - \lambda^{i+1} \\ &= F(X) = F_i \end{aligned}$$

Again, let $\tilde{U}$ follow a uniform distribution and we want to find X such that $F(X) = \tilde{U}$.

Let's deduce the distribution of X from

$$\begin{aligned} \tilde{U} = F(X) &= 1 - \lambda^{x+1} \\ 1 - \tilde{U} &= \lambda^{x+1} \\ log(1 - \tilde{U}) &= (x + 1) \log \lambda \\ \frac{\log(1 - \tilde{U})}{\log \lambda} &= x + 1 \\ \frac{\log(1 - \tilde{U})}{\log \lambda} - 1 &= x \end{aligned}$$

However, $\tilde{U} = F^{-1}(X)$ may not be an integer for any $x \geq 0$.

So let

$$x = \lceil \frac{\log(1 - \tilde{U})}{\log \lambda} - 1 \rceil$$

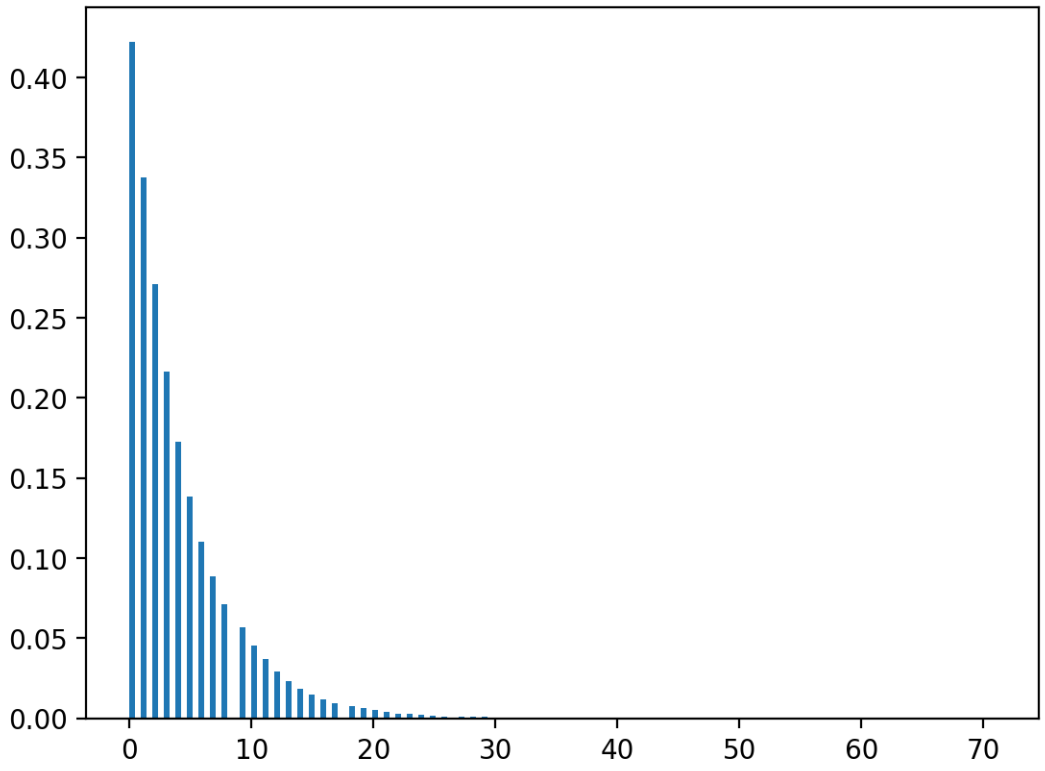
where $\lceil . \rceil$ is the ceiling function.

Thus x is the smallest integer such that the discrete geometric CDF is greater than or equal to $\tilde{U}$.

We can verify that x is indeed geometrically distributed by the following `numpy` program.

Note: The exponential distribution is the continuous analog of geometric distribution.

```
n, λ = 1_000_000, 0.8

# draw uniform numbers
u = np.random.rand(n)

# transform
x = np.ceil(np.log(1-u)/np.log(λ) - 1)

# draw geometric distributions
x_g = np.random.geometric(1-λ, n)

# plot and compare
plt.hist(x, bins=150, density=True)
plt.show()
```



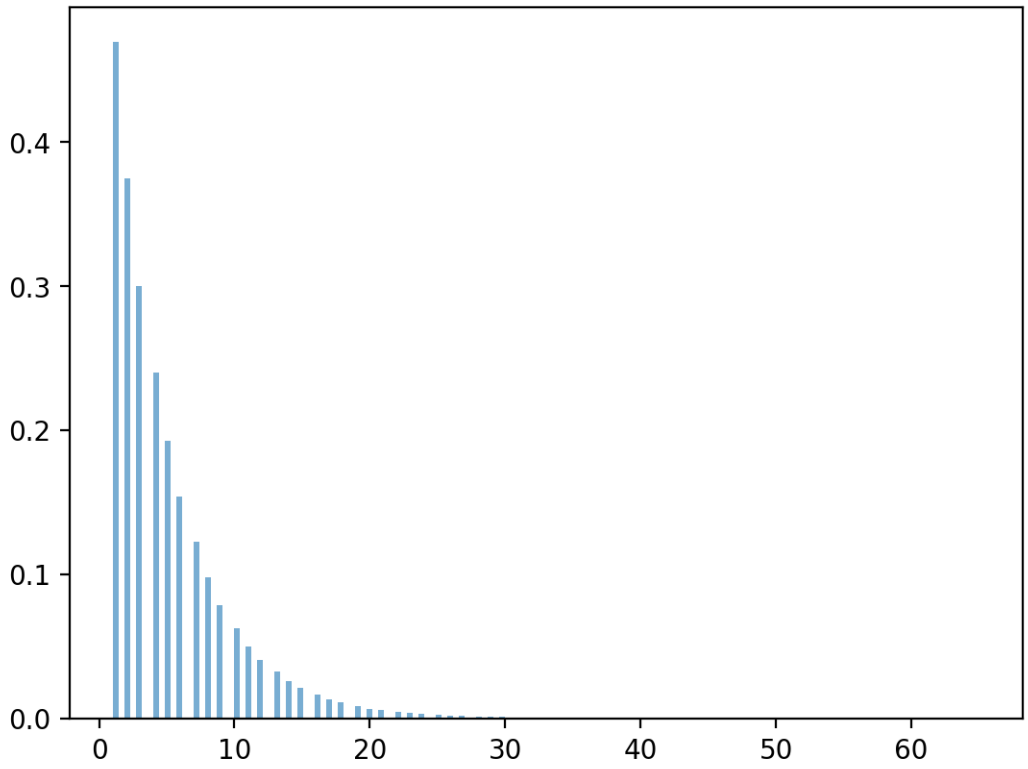
```
np.random.geometric(1-λ, n).max()
```

```
55
```

```
np.log(0.4)/np.log(0.3)
```

```
0.7610560044063083
```

```
plt.hist(x_g, bins=150, density=True, alpha=0.6)
plt.show()
```



4.11 Some Discrete Probability Distributions

Let's write some Python code to compute means and variances of some univariate random variables.

We'll use our code to

- compute population means and variances from the probability distribution
- generate a sample of N independently and identically distributed draws and compute sample means and variances
- compare population and sample means and variances

4.12 Geometric distribution

$$\text{Prob}(X = k) = (1-p)^{k-1}p, k = 1, 2, ...$$

$\Longrightarrow$

$$\mathbb{E}(X) = \frac{1}{p}$$
$$\mathbb{D}(X) = \frac{1-p}{p^2}$$

We draw observations from the distribution and compare the sample mean and variance with the theoretical results.

```
# specify parameters
p, n = 0.3, 1_000_000

# draw observations from the distribution
x = np.random.geometric(p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)

# compare with theoretical results
print("\nThe population mean is: ", 1/p)
print("The population variance is: ", (1-p)/(p**2))
```

```
The sample mean is:  3.328117
The sample variance is:  7.757662234311002

The population mean is:  3.3333333333333335
The population variance is:  7.777777777777778
```

4.12.1 Newcomb–Benford distribution

The **Newcomb–Benford law** fits many data sets, e.g., reports of incomes to tax authorities, in which the leading digit is more likely to be small than large.

See https://en.wikipedia.org/wiki/Benford's_law

A Benford probability distribution is

$$\text{Prob}\{X = d\} = \log_{10}(d+1) - \log_{10}(d) = \log_{10}\left(1 + \frac{1}{d}\right)$$

where $d \in \{1, 2, \cdots, 9\}$ can be thought of as a **first digit** in a sequence of digits.

This is a well defined discrete distribution since we can verify that probabilities are nonnegative and sum to 1.

$$\log_{10}\left(1 + \frac{1}{d}\right) \geq 0, \quad \sum_{d=1}^{9} \log_{10}\left(1 + \frac{1}{d}\right) = 1$$

The mean and variance of a Benford distribution are

$$\mathbb{E}[X] = \sum_{d=1}^{9} d \log_{10}\left(1 + \frac{1}{d}\right) \simeq 3.4402$$

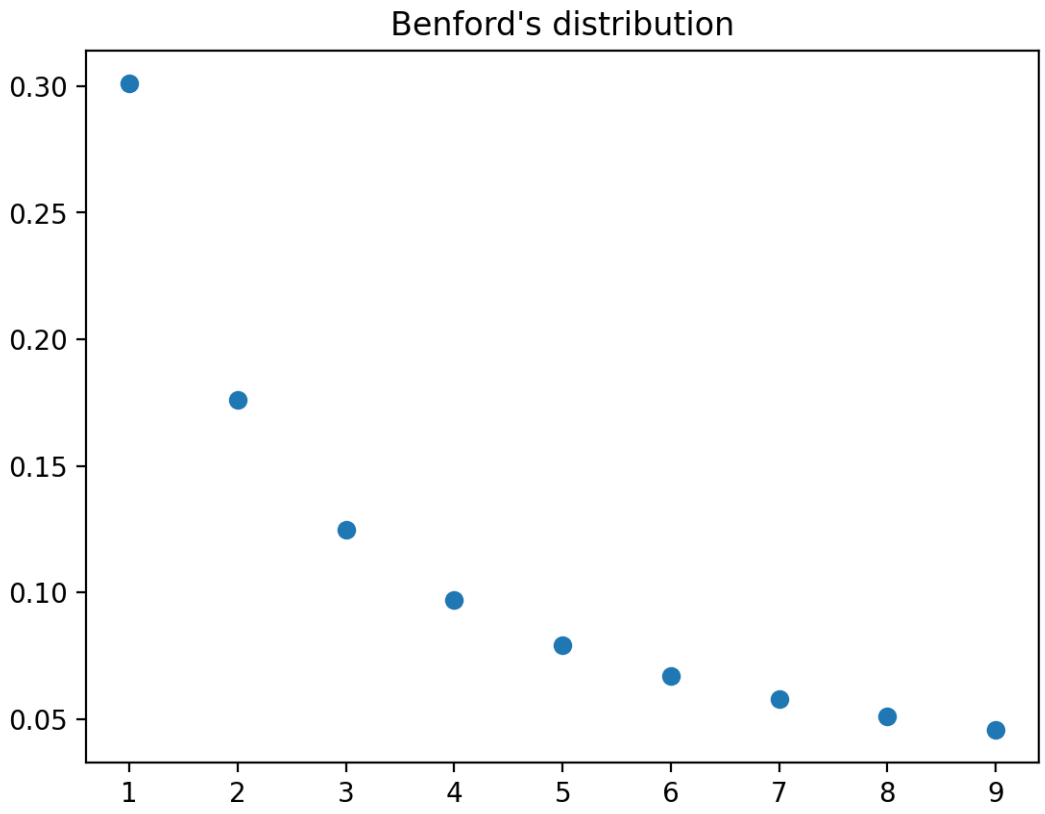
$$\mathbb{V}[X] = \sum_{d=1}^{9} (d - \mathbb{E}[X])^2 \log_{10}\left(1 + \frac{1}{d}\right) \simeq 6.0565$$

We verify the above and compute the mean and variance using `numpy`.

```
Benford_pmf = np.array([np.log10(1+1/d) for d in range(1,10)])
k = np.array(range(1,10))

# mean
mean = np.sum(Benford_pmf * k)

# variance
var = np.sum([(k-mean)**2 * Benford_pmf])

# verify sum to 1
print(np.sum(Benford_pmf))
print(mean)
print(var)
```

```
0.9999999999999999
3.440236967123206
6.056512631375667
```

```
# plot distribution
plt.plot(range(1,10), Benford_pmf, 'o')
plt.title('Benford\'s distribution')
plt.show()
```



4.12.2 Pascal (negative binomial) distribution

Consider a sequence of independent Bernoulli trials.

Let p be the probability of success.

Let X be a random variable that represents the number of failures before we get r success.

Its distribution is

$$X \sim NB(r, p)$$

$$\text{Prob}(X = k; r, p) = \binom{k+r-1}{r-1} p^r (1-p)^k$$

Here, we choose from among $k + r - 1$ possible outcomes because the last draw is by definition a success.

We compute the mean and variance to be

$$\mathbb{E}(X) = \frac{k(1-p)}{p}$$

$$\mathbb{V}(X) = \frac{k(1-p)}{p^2}$$

```
# specify parameters
r, p, n = 10, 0.3, 1_000_000

# draw observations from the distribution
```

(continues on next page)

(continued from previous page)

```
x = np.random.negative_binomial(r, p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", r*(1-p)/p)
print("The population variance is: ", r*(1-p)/p**2)
```

```
The sample mean is:  23.347525
The sample variance is:  77.84080337437506

The population mean is:  23.333333333333336
The population variance is:  77.77777777777779
```

4.13 Continuous Random Variables

4.13.1 Univariate Gaussian distribution

We write

$$X \sim N(\mu, \sigma^2)$$

to indicate the probability distribution

$$f(x|u, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{[-\frac{1}{2\sigma^2}(x-u)^2]}$$

In the below example, we set $\mu = 0, \sigma = 0.1$.

```
# specify parameters
μ, σ = 0, 0.1

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = np.random.normal(μ, σ, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ_hat = np.std(x)

print("The sample mean is: ", μ_hat)
print("The sample standard deviation is: ", σ_hat)
```

```
The sample mean is:  -0.00011309665761054191
The sample standard deviation is:  0.0999698463189021
```

```
# compare
print(μ-μ_hat < 1e-3)
print(σ-σ_hat < 1e-3)
```

```
True
True
```

4.13.2 Uniform Distribution

$$X \sim U[a, b]$$

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ 0, & \text{otherwise} \end{cases}$$

The population mean and variance are

$$\mathbb{E}(X) = \frac{a+b}{2}$$

$$\mathbb{V}(X) = \frac{(b-a)^2}{12}$$

```
# specify parameters
a, b = 10, 20

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = a + (b-a)*np.random.rand(n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", (a+b)/2)
print("The population variance is: ", (b-a)**2/12)
```

```
The sample mean is:  14.999655477794489
The sample variance is:  8.330771268334937

The population mean is:  15.0
The population variance is:  8.333333333333334
```

4.14 A Mixed Discrete-Continuous Distribution

We'll motivate this example with a little story.

Suppose that to apply for a job you take an interview and either pass or fail it.

You have 5% chance to pass an interview and you know your salary will uniformly distributed in the interval 300~400 a day only if you pass.

We can describe your daily salary as a discrete-continuous variable with the following probabilities:

$$P(X = 0) = 0.95$$

$$P(300 \leq X \leq 400) = \int_{300}^{400} f(x)\, dx = 0.05$$

$$f(x) = 0.0005$$

Let's start by generating a random sample and computing sample moments.

```
x = np.random.rand(1_000_000)
# x[x > 0.95] = 100*x[x > 0.95]+300
x[x > 0.95] = 100*np.random.rand(len(x[x > 0.95]))+300
x[x <= 0.95] = 0

μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
```

```
The sample mean is:  17.346745634474946
The sample variance is:  5809.800328592666
```

The analytical mean and variance can be computed:

$$\begin{aligned}
\mu &= \int_{300}^{400} xf(x)dx \\
&= 0.0005 \int_{300}^{400} xdx \\
&= 0.0005 \times \frac{1}{2}x^2 \Big|_{300}^{400}
\end{aligned}$$

$$\begin{aligned}
\sigma^2 &= 0.95 \times (0 - 17.5)^2 + \int_{300}^{400} (x - 17.5)^2 f(x)dx \\
&= 0.95 \times 17.5^2 + 0.0005 \int_{300}^{400} (x - 17.5)^2 dx \\
&= 0.95 \times 17.5^2 + 0.0005 \times \frac{1}{3}(x - 17.5)^3 \Big|_{300}^{400}
\end{aligned}$$

```
mean = 0.0005*0.5*(400**2 - 300**2)
var = 0.95*17.5**2+0.0005/3*((400-17.5)**3-(300-17.5)**3)
print("mean: ", mean)
print("variance: ", var)
```

```
mean:  17.5
variance:  5860.416666666666
```

4.15 Matrix Representation of Some Bivariate Distributions

Let's use matrices to represent a joint distribution, conditional distribution, marginal distribution, and the mean and variance of a bivariate random variable.

The table below illustrates a probability distribution for a bivariate random variable.

$$F = [f_{ij}] = \begin{bmatrix} 0.3 & 0.2 \\ 0.1 & 0.4 \end{bmatrix}$$

Marginal distributions are

$$\text{Prob}(X = i) = \sum_j f_{ij} = u_i$$

$$\text{Prob}(Y = j) = \sum_i f_{ij} = v_j$$

Below we draw some samples confirm that the "sampling" distribution agrees well with the "population" distribution.

Sample results:

```
# specify parameters
xs = np.array([0, 1])
ys = np.array([10, 20])
f = np.array([[0.3, 0.2], [0.1, 0.4]])
f_cum = np.cumsum(f)

# draw random numbers
p = np.random.rand(1_000_000)
x = np.vstack([xs[1]*np.ones(p.shape), ys[1]*np.ones(p.shape)])
# map to the bivariate distribution

x[0, p < f_cum[2]] = xs[1]
x[1, p < f_cum[2]] = ys[0]

x[0, p < f_cum[1]] = xs[0]
x[1, p < f_cum[1]] = ys[1]

x[0, p < f_cum[0]] = xs[0]
x[1, p < f_cum[0]] = ys[0]
print(x)
```

```
[[ 1.  1.  0. ...  1.  1.  1.]
 [10. 20. 20. ... 20. 10. 20.]]
```

Here, we use exactly the inverse CDF technique to generate sample from the joint distribution F.

```
# marginal distribution
xp = np.sum(x[0, :] == xs[0])/1_000_000
yp = np.sum(x[1, :] == ys[0])/1_000_000
```

(continues on next page)

(continued from previous page)

```
# print output
print("marginal distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([xs[0], xp])
xmtb.add_row([xs[1], 1-xp])
print(xmtb)

print("\nmarginal distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([ys[0], yp])
ymtb.add_row([ys[1], 1-yp])
print(ymtb)
```

```
marginal distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.499646 |
|    1    | 0.500354 |
+---------+----------+

marginal distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.399697 |
|    20   | 0.600303 |
+---------+----------+
```

```
# conditional distributions
xc1 = x[0, x[1, :] == ys[0]]
xc2 = x[0, x[1, :] == ys[1]]
yc1 = x[1, x[0, :] == xs[0]]
yc2 = x[1, x[0, :] == xs[1]]

xc1p = np.sum(xc1 == xs[0])/len(xc1)
xc2p = np.sum(xc2 == xs[0])/len(xc2)
yc1p = np.sum(yc1 == ys[0])/len(yc1)
yc2p = np.sum(yc2 == ys[0])/len(yc2)

# print output
print("conditional distribution for x")
xctb = pt.PrettyTable()
xctb.field_names = ['y_value', 'prob(x=0)', 'prob(x=1)']
xctb.add_row([ys[0], xc1p, 1-xc1p])
xctb.add_row([ys[1], xc2p, 1-xc2p])
print(xctb)

print("\nconditional distribution for y")
yctb = pt.PrettyTable()
yctb.field_names = ['x_value',  'prob(y=10)', 'prob(y=20)']
yctb.add_row([xs[0], yc1p, 1-yc1p])
```

(continues on next page)

(continued from previous page)

```
yctb.add_row([xs[1], yc2p, 1-yc2p])
print(yctb)
```

```
conditional distribution for x
+---------+--------------------+--------------------+
| y_value |      prob(x=0)     |      prob(x=1)     |
+---------+--------------------+--------------------+
|    10   |  0.7502533168875423 | 0.24974668311245773 |
|    20   | 0.33278527676856523 |  0.6672147232314347 |
+---------+--------------------+--------------------+

conditional distribution for y
+---------+--------------------+--------------------+
| x_value |      prob(y=10)    |     prob(y=20)     |
+---------+--------------------+--------------------+
|    0    |  0.6001729224290798 | 0.3998270775709202 |
|    1    | 0.19950475063654932 | 0.8004952493634507 |
+---------+--------------------+--------------------+
```

Let's calculate population marginal and conditional probabilities using matrix algebra.

$$\begin{bmatrix} & \vdots & y_1 & y_2 & \vdots & x \\ \cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\ x_1 & \vdots & 0.3 & 0.2 & \vdots & 0.5 \\ x_2 & \vdots & 0.1 & 0.4 & \vdots & 0.5 \\ \cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\ y & \vdots & 0.4 & 0.6 & \vdots & 1 \end{bmatrix}$$

$\Longrightarrow$

(1) Marginal distribution:

$$\begin{bmatrix} var & \vdots & var_1 & var_2 \\ \cdots & \vdots & \cdots & \cdots \\ x & \vdots & 0.5 & 0.5 \\ \cdots & \vdots & \cdots & \cdots \\ y & \vdots & 0.4 & 0.6 \end{bmatrix}$$

(2) Conditional distribution:

$$\begin{bmatrix} x & \vdots & x_1 & x_2 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ y = y_1 & \vdots & \frac{0.3}{0.4} = 0.75 & \frac{0.1}{0.4} = 0.25 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ y = y_2 & \vdots & \frac{0.2}{0.6} \approx 0.33 & \frac{0.4}{0.6} \approx 0.67 \end{bmatrix}$$

$$\begin{bmatrix} y & \vdots & y_1 & y_2 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ x = x_1 & \vdots & \frac{0.3}{0.5} = 0.6 & \frac{0.2}{0.5} = 0.4 \\ \cdots\cdots & \vdots & \cdots\cdots & \cdots\cdots \\ x = x_2 & \vdots & \frac{0.1}{0.5} = 0.2 & \frac{0.4}{0.5} = 0.8 \end{bmatrix}$$

These population objects closely resemble sample counterparts computed above.

Let's wrap some of the functions we have used in a Python class for a general discrete bivariate joint distribution.

```
class discrete_bijoint:

    def __init__(self, f, xs, ys):
        '''initialization
        -----------------
        parameters:
        f: the bivariate joint probability matrix
        xs: values of x vector
        ys: values of y vector
        '''
        self.f, self.xs, self.ys = f, xs, ys

    def joint_tb(self):
        '''print the joint distribution table'''
        xs = self.xs
        ys = self.ys
        f = self.f
        jtb = pt.PrettyTable()
        jtb.field_names = ['x_value/y_value', *ys, 'marginal sum for x']
        for i in range(len(xs)):
            jtb.add_row([xs[i], *f[i, :], np.sum(f[i, :])])
        jtb.add_row(['marginal_sum for y', *np.sum(f, 0), np.sum(f)])
        print("\nThe joint probability distribution for x and y\n", jtb)
        self.jtb = jtb

    def draw(self, n):
        '''draw random numbers
        ----------------------
        parameters:
        n: number of random numbers to draw
        '''
        xs = self.xs
        ys = self.ys
        f_cum = np.cumsum(self.f)
        p = np.random.rand(n)
        x = np.empty([2, p.shape[0]])
        lf = len(f_cum)
        lx = len(xs)-1
        ly = len(ys)-1
        for i in range(lf):
            x[0, p < f_cum[lf-1-i]] = xs[lx]
            x[1, p < f_cum[lf-1-i]] = ys[ly]
            if ly == 0:
                lx -= 1
                ly = len(ys)-1
            else:
                ly -= 1
        self.x = x
        self.n = n

    def marg_dist(self):
        '''marginal distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xmp = [np.sum(x[0, :] == xs[i])/n for i in range(len(xs))]
```

(continues on next page)

(continued from previous page)

```
        ymp = [np.sum(x[1, :] == ys[i])/n for i in range(len(ys))]

        # print output
        xmtb = pt.PrettyTable()
        ymtb = pt.PrettyTable()
        xmtb.field_names = ['x_value', 'x_prob']
        ymtb.field_names = ['y_value', 'y_prob']
        for i in range(max(len(xs), len(ys))):
            if i < len(xs):
                xmtb.add_row([xs[i], xmp[i]])
            if i < len(ys):
                ymtb.add_row([ys[i], ymp[i]])
        xmtb.add_row(['sum', np.sum(xmp)])
        ymtb.add_row(['sum', np.sum(ymp)])
        print("\nmarginal distribution for x\n", xmtb)
        print("\nmarginal distribution for y\n", ymtb)

        self.xmp = xmp
        self.ymp = ymp

    def cond_dist(self):
        '''conditional distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xcp = np.empty([len(ys), len(xs)])
        ycp = np.empty([len(xs), len(ys)])
        for i in range(max(len(ys), len(xs))):
            if i < len(ys):
                xi = x[0, x[1, :] == ys[i]]
                idx = xi.reshape(len(xi), 1) == xs.reshape(1, len(xs))
                xcp[i, :] = np.sum(idx, 0)/len(xi)
            if i < len(xs):
                yi = x[1, x[0, :] == xs[i]]
                idy = yi.reshape(len(yi), 1) == ys.reshape(1, len(ys))
                ycp[i, :] = np.sum(idy, 0)/len(yi)

        # print output
        xctb = pt.PrettyTable()
        yctb = pt.PrettyTable()
        xctb.field_names = ['x_value', *xs, 'sum']
        yctb.field_names = ['y_value', *ys, 'sum']
        for i in range(max(len(xs), len(ys))):
            if i < len(ys):
                xctb.add_row([ys[i], *xcp[i], np.sum(xcp[i])])
            if i < len(xs):
                yctb.add_row([xs[i], *ycp[i], np.sum(ycp[i])])
        print("\nconditional distribution for x\n", xctb)
        print("\nconditional distribution for y\n", yctb)

        self.xcp = xcp
        self.xyp = ycp
```

Let's apply our code to some examples.

Example 1

```
# joint
d = discrete_bijoint(f, xs, ys)
d.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+-----+--------------------+--------------------+
|  x_value/y_value   |  10 |         20         | marginal sum for x |
+--------------------+-----+--------------------+--------------------+
|         0          | 0.3 |        0.2         |        0.5         |
|         1          | 0.1 |        0.4         |        0.5         |
| marginal_sum for y | 0.4 | 0.6000000000000001 |        1.0         |
+--------------------+-----+--------------------+--------------------+
```

```
# sample marginal
d.draw(1_000_000)
d.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.499354 |
|    1    | 0.500646 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.399926 |
|    20   | 0.600074 |
|   sum   |   1.0    |
+---------+----------+
```

```
# sample conditional
d.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+---------------------+-----+
| x_value |          0         |          1          | sum |
+---------+--------------------+---------------------+-----+
|    10   |  0.748493471292189 | 0.25150652870781093 | 1.0 |
|    20   | 0.33331222482560485 |  0.6666877751743951 | 1.0 |
+---------+--------------------+---------------------+-----+

conditional distribution for y
 +---------+--------------------+--------------------+-----+
| y_value |         10         |         20         | sum |
+---------+--------------------+--------------------+-----+
|    0    |  0.5994585003824942 | 0.4005414996175058 | 1.0 |
|    1    | 0.20090842631320335 | 0.7990915736867966 | 1.0 |
+---------+--------------------+--------------------+-----+
```

Example 2

```
xs_new = np.array([10, 20, 30])
ys_new = np.array([1, 2])
f_new = np.array([[0.2, 0.1], [0.1, 0.3], [0.15, 0.15]])
d_new = discrete_bijoint(f_new, xs_new, ys_new)
d_new.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+---------------------+------+---------------------+
|  x_value/y_value   |          1          |  2   |  marginal sum for x |
+--------------------+---------------------+------+---------------------+
|         10         |         0.2         | 0.1  | 0.30000000000000004 |
|         20         |         0.1         | 0.3  |         0.4         |
|         30         |         0.15        | 0.15 |         0.3         |
| marginal_sum for y | 0.45000000000000007 | 0.55 |         1.0         |
+--------------------+---------------------+------+---------------------+
```

```
d_new.draw(1_000_000)
d_new.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    10   | 0.29986  |
|    20   | 0.399805 |
|    30   | 0.300335 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+---------+
| y_value |  y_prob |
+---------+---------+
|    1    | 0.45002 |
|    2    | 0.54998 |
|   sum   |   1.0   |
+---------+---------+
```

```
d_new.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+--------------------+--------------------+-----+
| x_value |         10         |         20         |         30         | sum |
+---------+--------------------+--------------------+--------------------+-----+
|    1    | 0.444689124927781  | 0.2221256833029643 | 0.3331851917692547 | 1.0 |
|    2    | 0.1813538674133605 | 0.5451907342085167 | 0.2734553983781228 | 1.0 |
+---------+--------------------+--------------------+--------------------+-----+

conditional distribution for y
 +---------+---------------------+---------------------+-----+
| y_value |          1          |          2          | sum |
+---------+---------------------+---------------------+-----+
```

(continues on next page)

(continued from previous page)

```
|    10    |  0.667374774894951   | 0.33262522510504905 | 1.0 |
|    20    |  0.2500243868886082  |  0.7499756131113918 | 1.0 |
|    30    | 0.49924251252767743  |  0.5007574874723226 | 1.0 |
+----------+----------------------+---------------------+-----+
```

4.16 A Continuous Bivariate Random Vector

A two-dimensional Gaussian distribution has joint density

$$f(x,y) = (2\pi\sigma_1\sigma_2\sqrt{1-\rho^2})^{-1} \exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

$$\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} 0 \\ 5 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 5 & .2 \\ .2 & 1 \end{bmatrix}$$

```
# define the joint probability density function
def func(x, y, μ1=0, μ2=5, σ1=np.sqrt(5), σ2=np.sqrt(1), ρ=.2/np.sqrt(5*1)):
    A = (2 * np.pi * σ1 * σ2 * np.sqrt(1 - ρ**2))**(-1)
    B = -1 / 2 / (1 - ρ**2)
    C1 = (x - μ1)**2 / σ1**2
    C2 = 2 * ρ * (x - μ1) * (y - μ2) / σ1 / σ2
    C3 = (y - μ2)**2 / σ2**2
    return A * np.exp(B * (C1 - C2 + C3))
```

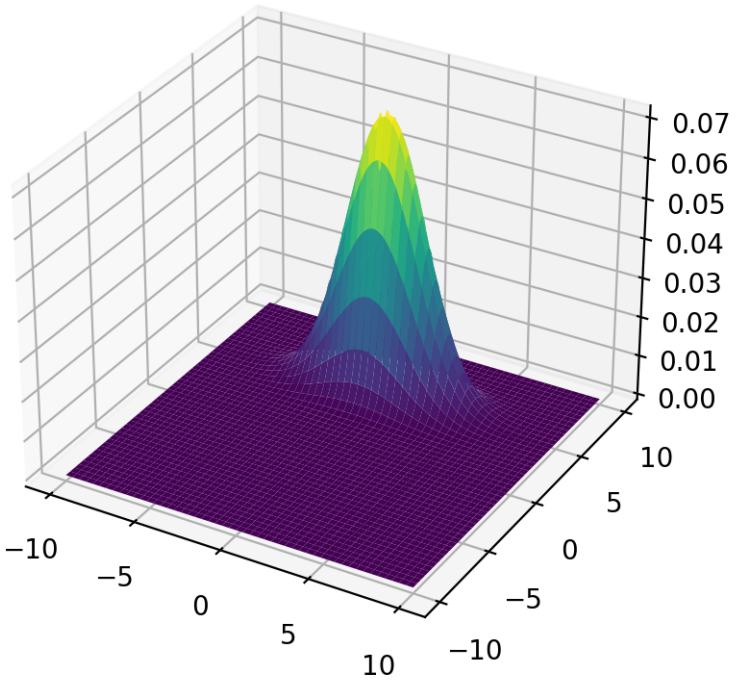
```
μ1 = 0
μ2 = 5
σ1 = np.sqrt(5)
σ2 = np.sqrt(1)
ρ = .2 / np.sqrt(5 * 1)
```

```
x = np.linspace(-10, 10, 1_000)
y = np.linspace(-10, 10, 1_000)
x_mesh, y_mesh = np.meshgrid(x, y, indexing="ij")
```

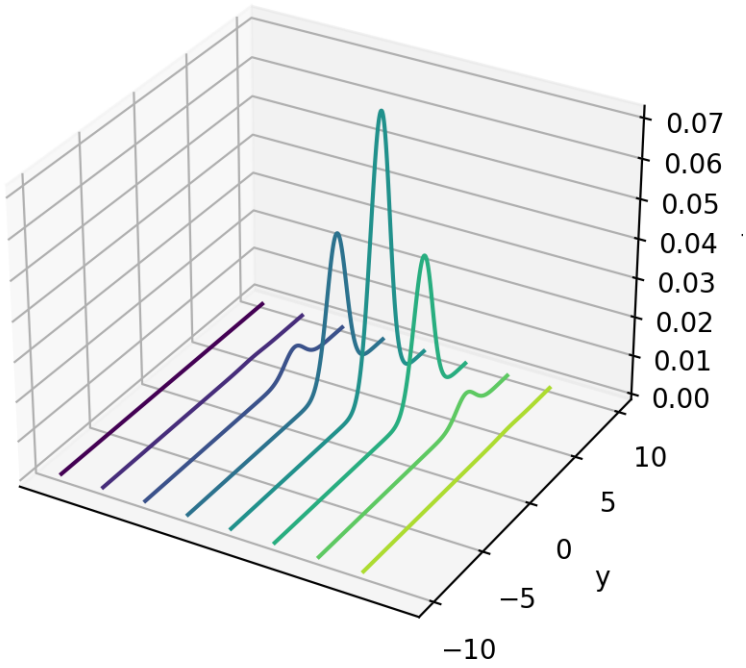
Joint Distribution

Let's plot the **population** joint density.

```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

surf = ax.plot_surface(x_mesh, y_mesh, func(x_mesh, y_mesh), cmap='viridis')
plt.show()
```




```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

curve = ax.contour(x_mesh, y_mesh, func(x_mesh, y_mesh), zdir='x')
plt.ylabel('y')
ax.set_zlabel('f')
ax.set_xticks([])
plt.show()
```



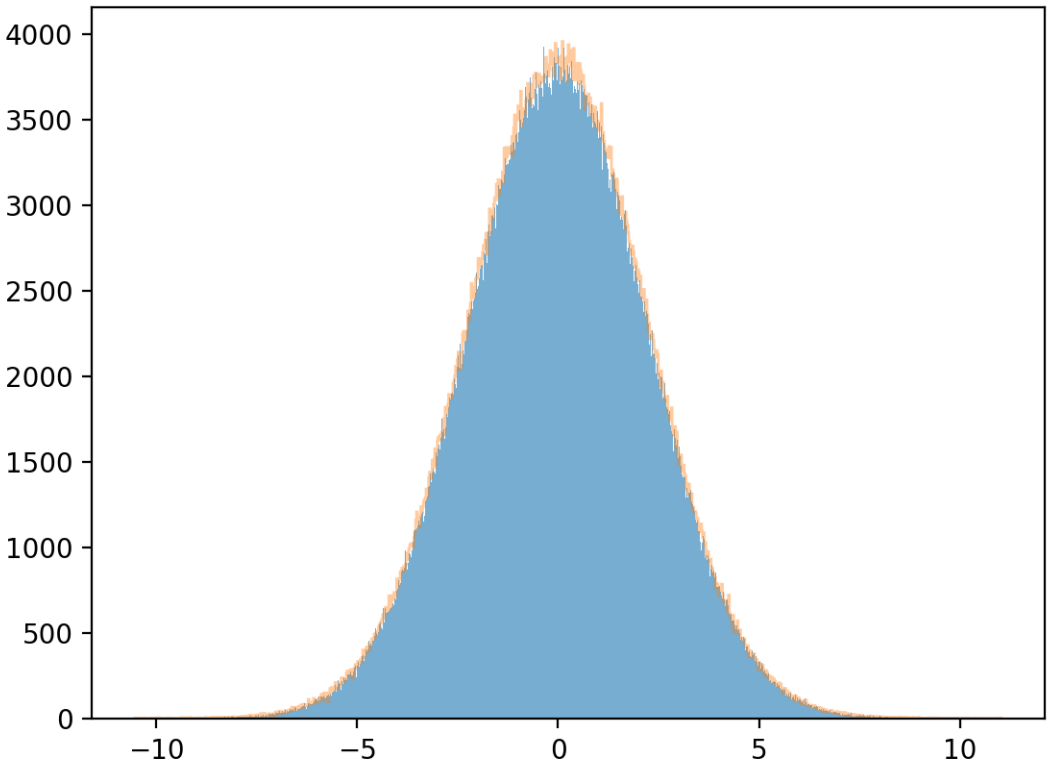
Next we can simulate from a built-in `numpy` function and calculate a **sample** marginal distribution from the sample mean and variance.

```
μ= np.array([0, 5])
σ= np.array([[5, .2], [.2, 1]])
n = 1_000_000
data = np.random.multivariate_normal(μ, σ, n)
x = data[:, 0]
y = data[:, 1]
```

Marginal distribution

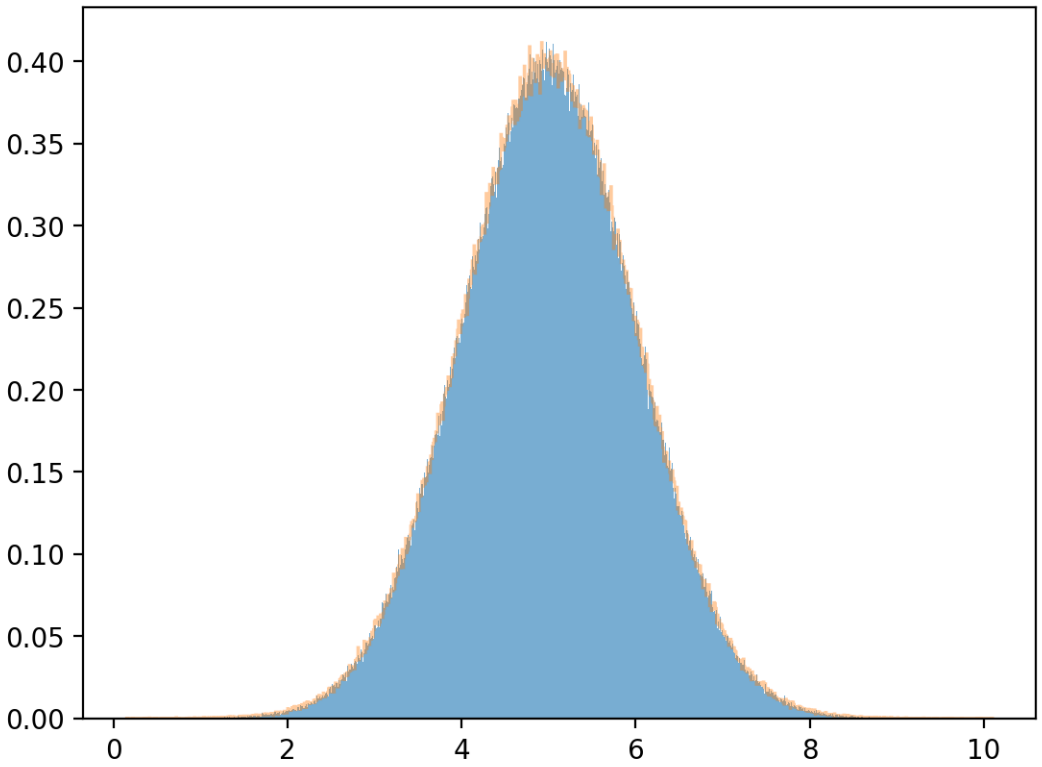
```
plt.hist(x, bins=1_000, alpha=0.6)
μx_hat, σx_hat = np.mean(x), np.std(x)
print(μx_hat, σx_hat)
x_sim = np.random.normal(μx_hat, σx_hat, 1_000_000)
plt.hist(x_sim, bins=1_000, alpha=0.4, histtype="step")
plt.show()
```

```
0.0004958360623134676 2.2355510116371384
```




```
plt.hist(y, bins=1_000, density=True, alpha=0.6)
μy_hat, σy_hat = np.mean(y), np.std(y)
print(μy_hat, σy_hat)
y_sim = np.random.normal(μy_hat, σy_hat, 1_000_000)
plt.hist(y_sim, bins=1_000, density=True, alpha=0.4, histtype="step")
plt.show()
```

```
4.99953178052401 0.9994150311616088
```



Conditional distribution

The population conditional distribution is

$$[X|Y=y] \sim \mathbb{N}\left[\mu_X + \rho\sigma_X \frac{y-\mu_Y}{\sigma_Y}, \sigma_X^2(1-\rho^2)\right]$$
$$[Y|X=x] \sim \mathbb{N}\left[\mu_Y + \rho\sigma_Y \frac{x-\mu_X}{\sigma_X}, \sigma_Y^2(1-\rho^2)\right]$$

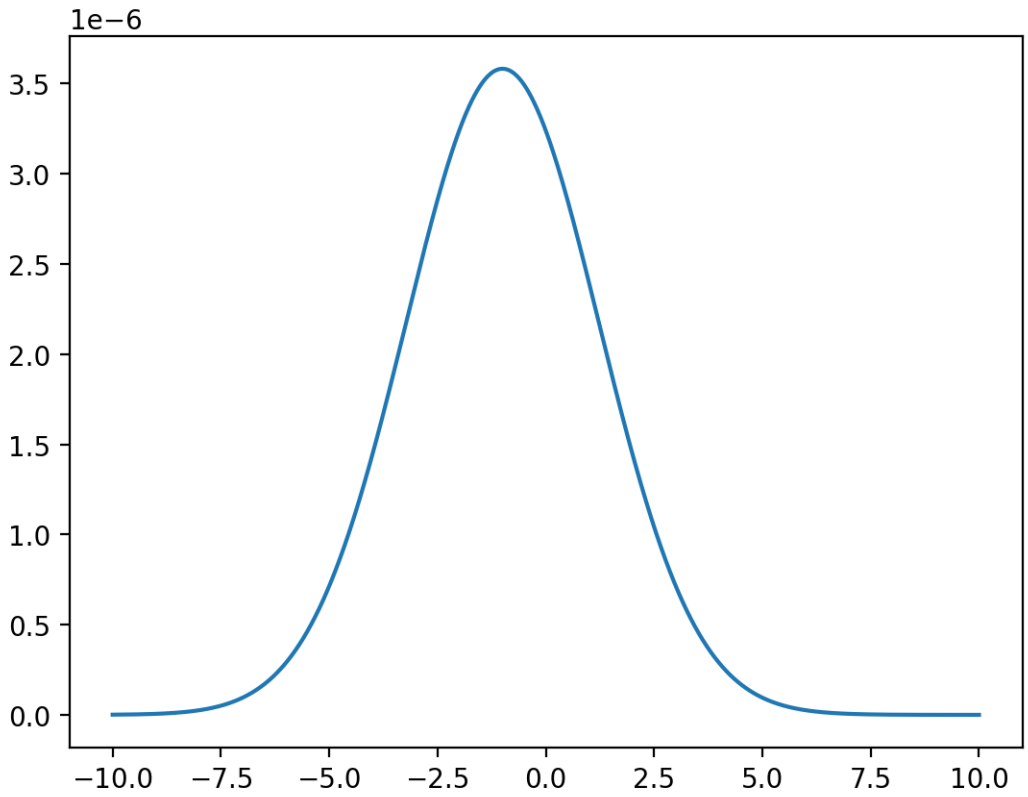
Let's approximate the joint density by discretizing and mapping the approximating joint density into a matrix.

We can compute the discretized marginal density by just using matrix algebra and noting that

$$\text{Prob}\{X=i|Y=j\} = \frac{f_{ij}}{\sum_i f_{ij}}$$

Fix $y=0$.

```
# discretized marginal density
x = np.linspace(-10, 10, 1_000_000)
z = func(x, y=0) / np.sum(func(x, y=0))
plt.plot(x, z)
plt.show()
```



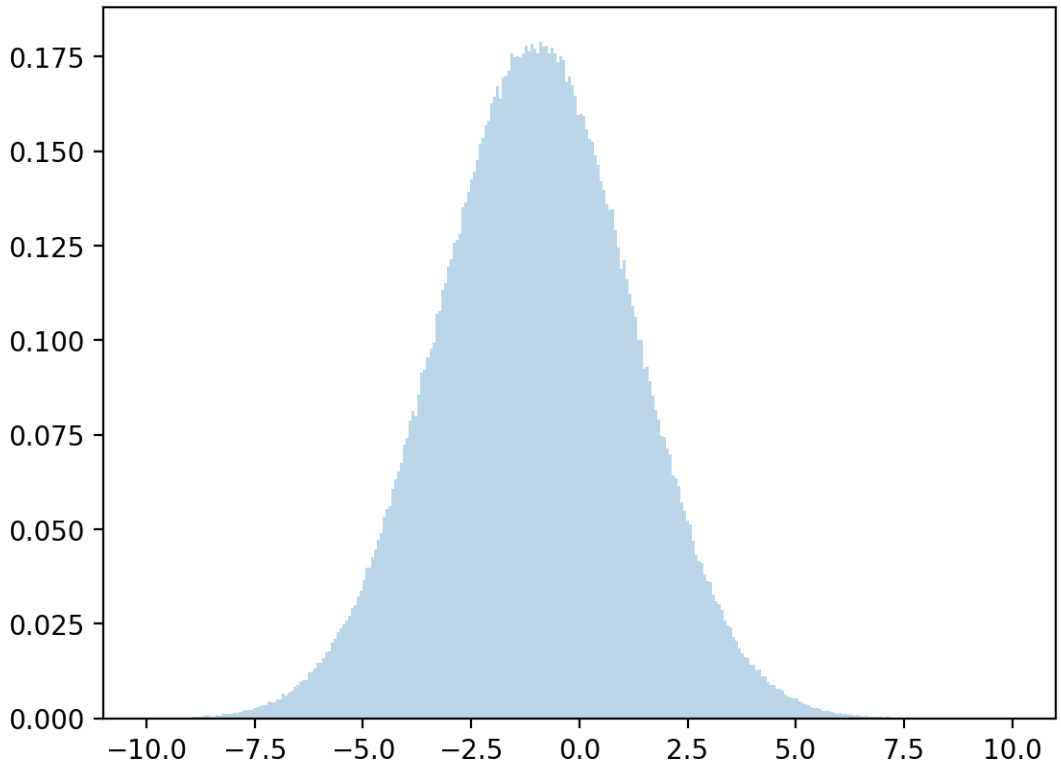
The mean and variance are computed by

$$\mathbb{E}\left[X|Y=j\right] = \sum_i iProb\{X=i|Y=j\} = \sum_i i\frac{f_{ij}}{\sum_i f_{ij}}$$

$$\mathbb{D}\left[X|Y=j\right] = \sum_i \left(i-\mu_{X|Y=j}\right)^2 \frac{f_{ij}}{\sum_i f_{ij}}$$

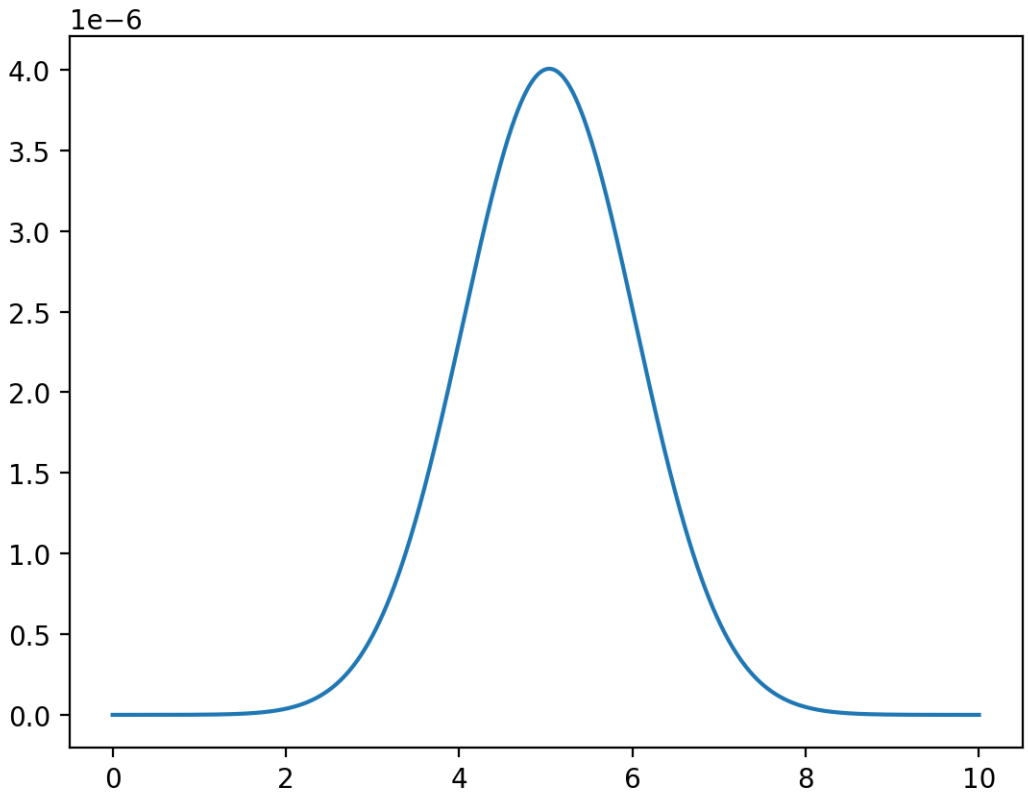
Let's draw from a normal distribution with above mean and variance and check how accurate our approximation is.

```
# discretized mean
μx = np.dot(x, z)

# discretized standard deviation
σx = np.sqrt(np.dot((x - μx)**2, z))

# sample
zz = np.random.normal(μx, σx, 1_000_000)
plt.hist(zz, bins=300, density=True, alpha=0.3, range=[-10, 10])
plt.show()
```

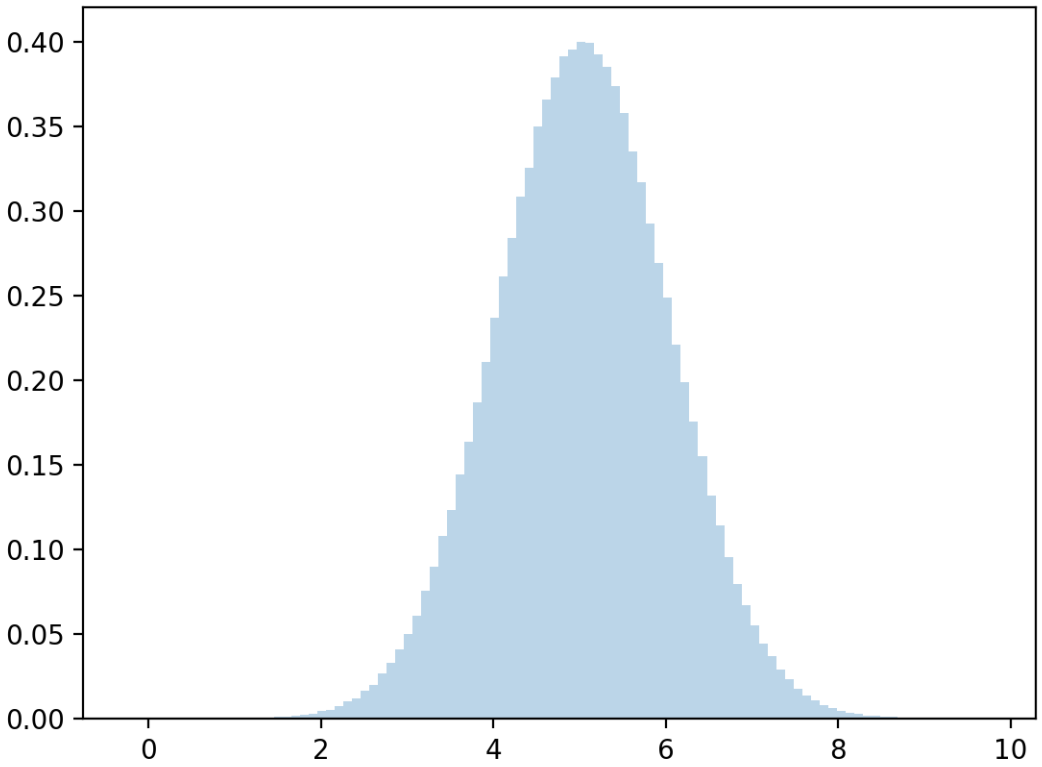


Fix $x = 1$.

```
y = np.linspace(0, 10, 1_000_000)
z = func(x=1, y=y) / np.sum(func(x=1, y=y))
plt.plot(y,z)
plt.show()
```




```
# discretized mean and standard deviation
μy = np.dot(y,z)
σy = np.sqrt(np.dot((y - μy)**2, z))

# sample
zz = np.random.normal(μy,σy,1_000_000)
plt.hist(zz, bins=100, density=True, alpha=0.3)
plt.show()
```



We compare with the analytically computed parameters and note that they are close.

```
print(μx, σx)
print(μ1 + ρ * σ1 * (0 - μ2) / σ2, np.sqrt(σ1**2 * (1 - ρ**2)))

print(μy, σy)
print(μ2 + ρ * σ2 * (1 - μ1) / σ1, np.sqrt(σ2**2 * (1 - ρ**2)))
```

```
-0.9997518414498433 2.22658413316977
-1.0 2.227105745132009
5.039999456960768 0.9959851265795597
5.04 0.9959919678390986
```

4.17 Sum of Two Independently Distributed Random Variables

Let X, Y be two independent discrete random variables that take values in $\bar{X}, \bar{Y}$, respectively.

Define a new random variable $Z = X + Y$.

Evidently, Z takes values from $\bar{Z}$ defined as follows:

$$
\begin{aligned}
\bar{X} &= \{0, 1, \ldots, I-1\}; \qquad & f_i &= \text{Prob}\{X = i\} \\
\bar{Y} &= \{0, 1, \ldots, J-1\}; \qquad & g_j &= \text{Prob}\{Y = j\} \\
\bar{Z} &= \{0, 1, \ldots, I+J-2\}; \qquad & h_k &= \text{Prob}\{X + Y = k\}
\end{aligned}
$$

Independence of X and Y implies that

$$h_k = \text{Prob}\{X = 0, Y = k\} + \text{Prob}\{X = 1, Y = k-1\} + ... + \text{Prob}\{X = k, Y = 0\}$$
$$h_k = f_0 g_k + f_1 g_{k-1} + ... + f_{k-1} g_1 + f_k g_0 \qquad \text{for} \quad k = 0, 1, ..., I + J - 2$$

Thus, we have:

$$h_k = \sum_{i=0}^{k} f_i g_{k-i} \equiv f * g$$

where $f * g$ denotes the **convolution** of the f and g sequences.

Similarly, for two random variables X, Y with densities f_X, g_Y, the density of $Z = X + Y$ is

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(z - x) dx \equiv f_X * g_Y$$

where $f_X * g_Y$ denotes the **convolution** of the f_X and g_Y functions.

4.18 Transition Probability Matrix

Consider the following joint probability distribution of two random variables.

Let X, Y be discrete random variables with joint distribution

$$\text{Prob}\{X = i, Y = j\} = \rho_{ij}$$

where $i = 0, ..., I - 1; j = 0, ..., J - 1$ and

$$\sum_i \sum_j \rho_{ij} = 1, \quad \rho_{ij} \geqslant 0.$$

An associated conditional distribution is

$$\text{Prob}\{Y = i | X = j\} = \frac{\rho_{ij}}{\sum_i \rho_{ij}} = \frac{\text{Prob}\{Y = j, X = i\}}{\text{Prob}\{X = i\}}$$

We can define a transition probability matrix

$$p_{ij} = \text{Prob}\{Y = j | X = i\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}}$$

where

$$\begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

The first row is the probability of $Y = j, j = 0, 1$ conditional on $X = 0$.

The second row is the probability of $Y = j, j = 0, 1$ conditional on $X = 1$.

Note that

- $\sum_j \rho_{ij} = \frac{\sum_j \rho_{ij}}{\sum_j \rho_{ij}} = 1$, so each row of ρ is a probability distribution (not so for each column.

4.19 Coupling

Start with a joint distribution

$$\begin{aligned} f_{ij} &= \text{Prob}\{X = i, Y = j\} \\ i &= 0, \cdots I-1 \\ j &= 0, \cdots J-1 \\ &\text{stacked to an } I \times J \text{ matrix} \\ &e.g. \quad I = 1, J = 1 \end{aligned}$$

where

$$\begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}$$

From the joint distribution, we have shown above that we obtain **unique** marginal distributions.

Now we'll try to go in a reverse direction.

We'll find that from two marginal distributions, can we usually construct more than one joint distribution that verifies these marginals.

Each of these joint distributions is called a **coupling** of the two martingal distributions.

Let's start with marginal distributions

$$\begin{aligned} \text{Prob}\{X = i\} &= \sum_j f_{ij} = \mu_i, i = 0, \cdots, I-1 \\ \text{Prob}\{Y = j\} &= \sum_j f_{ij} = \nu_j, j = 0, \cdots, J-1 \end{aligned}$$

Given two marginal distribution, μ for X and ν for Y, a joint distribution f_{ij} is said to be a **coupling** of μ and ν.

Example:

Consider the following bivariate example.

$$\begin{aligned} \text{Prob}\{X = 0\} &= 1 - q = \mu_0 \\ \text{Prob}\{X = 1\} &= q = \mu_1 \\ \text{Prob}\{Y = 0\} &= 1 - r = \nu_0 \\ \text{Prob}\{Y = 1\} &= r = \nu_1 \\ \text{where } 0 \leq q < r \leq 1 & \end{aligned}$$

We construct two couplings.

The first coupling if our two marginal distributions is the joint distribution

$$f_{ij} = \begin{bmatrix} (1-q)(1-r) & (1-q)r \\ q(1-r) & qr \end{bmatrix}$$

To verify that it is a coupling, we check that

$$\begin{aligned} (1-q)(1-r) + (1-q)r + q(1-r) + qr &= 1 \\ \mu_0 = (1-q)(1-r) + (1-q)r &= 1 - q \\ \mu_1 = q(1-r) + qr &= q \\ \nu_0 = (1-q)(1-r) + (1-r)q &= 1 - r \\ \mu_1 = r(1-q) + qr &= r \end{aligned}$$

A second coupling of our two marginal distributions is the joint distribution

$$f_{ij} = \begin{bmatrix} (1-r) & r-q \\ 0 & q \end{bmatrix}$$

The verify that this is a coupling, note that

$$\begin{aligned} 1 - r + r - q + q &= 1 \\ \mu_0 &= 1 - q \\ \mu_1 &= q \\ \nu_0 &= 1 - r \\ \nu_1 &= r \end{aligned}$$

Thus, our two proposed joint distributions have the same marginal distributions.

But the joint distributions differ.

Thus, multiple joint distributions $[f_{ij}]$ can have the same marginals.

Remark:

- Couplings are important in optimal transport problems and in Markov processes.

4.20 Copula Functions

Suppose that $X_1, X_2, \ldots, X_n$ are N random variables and that

- their marginal distributions are $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$, and
- their joint distribution is $H(x_1, x_2, \ldots, x_N)$

Then there exists a **copula function** $C(\cdot)$ that verifies

$$H(x_1, x_2, \ldots, x_N) = C(F_1(x_1), F_2(x_2), \ldots, F_N(x_N)).$$

We can obtain

$$C(u_1, u_2, \ldots, u_n) = H[F_1^{-1}(u_1), F_2^{-1}(u_2), \ldots, F_N^{-1}(u_N)]$$

In a reverse direction of logic, given univariate **marginal distributions** $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$ and a copula function $C(\cdot)$, the function $H(x_1, x_2, \ldots, x_N) = C(F_1(x_1), F_2(x_2), \ldots, F_N(x_N))$ is a **coupling** of $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$.

Thus, for given marginal distributions, we can use a copula function to determine a joint distribution when the associated univariate random variables are not independent.

Copula functions are often used to characterize **dependence** of random variables.

Discrete marginal distribution

As mentioned above, for two given marginal distributions there can be more than one coupling.

For example, consider two random variables X, Y with distributions

$$\begin{aligned} \text{Prob}(X = 0) &= 0.6, \\ \text{Prob}(X = 1) &= 0.4, \\ \text{Prob}(Y = 0) &= 0.3, \\ \text{Prob}(Y = 1) &= 0.7, \end{aligned}$$

For these two random variables there can be more than one coupling.

Let's first generate X and Y.

```
# define parameters
mu = np.array([0.6, 0.4])
nu = np.array([0.3, 0.7])

# number of draws
draws = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws)

# generate draws of X and Y via uniform distribution
x = np.ones(draws)
y = np.ones(draws)
x[p <= mu[0]] = 0
x[p > mu[0]] = 1
y[p <= nu[0]] = 0
y[p > nu[0]] = 1
```

```
# calculate parameters from draws
q_hat = sum(x[x == 1])/draws
r_hat = sum(y[y == 1])/draws

# print output
print("distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([0, 1-q_hat])
xmtb.add_row([1, q_hat])
print(xmtb)

print("distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([0, 1-r_hat])
ymtb.add_row([1, r_hat])
print(ymtb)
```

```
distribution for x
+---------+--------------------+
| x_value |       x_prob       |
+---------+--------------------+
|    0    | 0.6000639999999999 |
|    1    |      0.399936      |
+---------+--------------------+
distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    0    | 0.299975 |
|    1    | 0.700025 |
+---------+----------+
```

Let's now take our two marginal distributions, one for X, the other for Y, and construct two distinct couplings.

For the first joint distribution:

$$\text{Prob}(X = i, Y = j) = f_{ij}$$

where

$$[f_{ij}] = \begin{bmatrix} 0.18 & 0.42 \\ 0.12 & 0.28 \end{bmatrix}$$

Let's use Python to construct this joint distribution and then verify that its marginal distributions are what we want.

```
# define parameters
f1 = np.array([[0.18, 0.42], [0.12, 0.28]])
f1_cum = np.cumsum(f1)

# number of draws
draws1 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws1)

# generate draws of first copuling via uniform distribution
c1 = np.vstack([np.ones(draws1), np.ones(draws1)])
# X=0, Y=0
c1[0, p <= f1_cum[0]] = 0
c1[1, p <= f1_cum[0]] = 0
# X=0, Y=1
c1[0, (p > f1_cum[0])*(p <= f1_cum[1])] = 0
c1[1, (p > f1_cum[0])*(p <= f1_cum[1])] = 1
# X=1, Y=0
c1[0, (p > f1_cum[1])*(p <= f1_cum[2])] = 1
c1[1, (p > f1_cum[1])*(p <= f1_cum[2])] = 0
# X=1, Y=1
c1[0, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
c1[1, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
```

```
# calculate parameters from draws
f1_00 = sum((c1[0, :] == 0)*(c1[1, :] == 0))/draws1
f1_01 = sum((c1[0, :] == 0)*(c1[1, :] == 1))/draws1
f1_10 = sum((c1[0, :] == 1)*(c1[1, :] == 0))/draws1
f1_11 = sum((c1[0, :] == 1)*(c1[1, :] == 1))/draws1

# print output of first joint distribution
print("first joint distribution for c1")
c1_mtb = pt.PrettyTable()
c1_mtb.field_names = ['c1_x_value', 'c1_y_value', 'c1_prob']
c1_mtb.add_row([0, 0, f1_00])
c1_mtb.add_row([0, 1, f1_01])
c1_mtb.add_row([1, 0, f1_10])
c1_mtb.add_row([1, 1, f1_11])
print(c1_mtb)
```

```
first joint distribution for c1
+------------+------------+----------+
| c1_x_value | c1_y_value | c1_prob  |
+------------+------------+----------+
|     0      |     0      |  0.1804  |
|     0      |     1      | 0.420325 |
|     1      |     0      | 0.119486 |
|     1      |     1      | 0.279789 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c1_q_hat = sum(c1[0, :] == 1)/draws1
c1_r_hat = sum(c1[1, :] == 1)/draws1

# print output
print("marginal distribution for x")
c1_x_mtb = pt.PrettyTable()
c1_x_mtb.field_names = ['c1_x_value', 'c1_x_prob']
c1_x_mtb.add_row([0, 1-c1_q_hat])
c1_x_mtb.add_row([1, c1_q_hat])
print(c1_x_mtb)

print("marginal distribution for y")
c1_ymtb = pt.PrettyTable()
c1_ymtb.field_names = ['c1_y_value', 'c1_y_prob']
c1_ymtb.add_row([0, 1-c1_r_hat])
c1_ymtb.add_row([1, c1_r_hat])
print(c1_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c1_x_value | c1_x_prob |
+------------+-----------+
|     0      |  0.600725 |
|     1      |  0.399275 |
+------------+-----------+
marginal distribution for y
+------------+-----------+
| c1_y_value | c1_y_prob |
+------------+-----------+
|     0      |  0.299886 |
|     1      |  0.700114 |
+------------+-----------+
```

Now, let's construct another joint distribution that is also a coupling of X and Y

$$[f_{ij}] = \begin{bmatrix} 0.3 & 0.3 \\ 0 & 0.4 \end{bmatrix}$$

```
# define parameters
f2 = np.array([[0.3, 0.3], [0, 0.4]])
f2_cum = np.cumsum(f2)

# number of draws
draws2 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws2)

# generate draws of first coupling via uniform distribution
c2 = np.vstack([np.ones(draws2), np.ones(draws2)])
# X=0, Y=0
c2[0, p <= f2_cum[0]] = 0
c2[1, p <= f2_cum[0]] = 0
# X=0, Y=1
c2[0, (p > f2_cum[0])*(p <= f2_cum[1])] = 0
```

(continues on next page)

(continued from previous page)

```
c2[1, (p > f2_cum[0])*(p <= f2_cum[1])] = 1
# X=1, Y=0
c2[0, (p > f2_cum[1])*(p <= f2_cum[2])] = 1
c2[1, (p > f2_cum[1])*(p <= f2_cum[2])] = 0
# X=1, Y=1
c2[0, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
c2[1, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
```

```
# calculate parameters from draws
f2_00 = sum((c2[0, :] == 0)*(c2[1, :] == 0))/draws2
f2_01 = sum((c2[0, :] == 0)*(c2[1, :] == 1))/draws2
f2_10 = sum((c2[0, :] == 1)*(c2[1, :] == 0))/draws2
f2_11 = sum((c2[0, :] == 1)*(c2[1, :] == 1))/draws2

# print output of second joint distribution
print("first joint distribution for c2")
c2_mtb = pt.PrettyTable()
c2_mtb.field_names = ['c2_x_value', 'c2_y_value', 'c2_prob']
c2_mtb.add_row([0, 0, f2_00])
c2_mtb.add_row([0, 1, f2_01])
c2_mtb.add_row([1, 0, f2_10])
c2_mtb.add_row([1, 1, f2_11])
print(c2_mtb)
```

```
first joint distribution for c2
+------------+------------+----------+
| c2_x_value | c2_y_value | c2_prob  |
+------------+------------+----------+
|     0      |     0      | 0.29999  |
|     0      |     1      | 0.300061 |
|     1      |     0      |   0.0    |
|     1      |     1      | 0.399949 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c2_q_hat = sum(c2[0, :] == 1)/draws2
c2_r_hat = sum(c2[1, :] == 1)/draws2

# print output
print("marginal distribution for x")
c2_x_mtb = pt.PrettyTable()
c2_x_mtb.field_names = ['c2_x_value', 'c2_x_prob']
c2_x_mtb.add_row([0, 1-c2_q_hat])
c2_x_mtb.add_row([1, c2_q_hat])
print(c2_x_mtb)

print("marginal distribution for y")
c2_ymtb = pt.PrettyTable()
c2_ymtb.field_names = ['c2_y_value', 'c2_y_prob']
c2_ymtb.add_row([0, 1-c2_r_hat])
c2_ymtb.add_row([1, c2_r_hat])
print(c2_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c2_x_value | c2_x_prob |
+------------+-----------+
|     0      |  0.600051 |
|     1      |  0.399949 |
+------------+-----------+
marginal distribution for y
+------------+-----------+
| c2_y_value | c2_y_prob |
+------------+-----------+
|     0      |  0.29999  |
|     1      |  0.70001  |
+------------+-----------+
```

We have verified that both joint distributions, c_1 and c_2, have identical marginal distributions of X and Y, respectively.

So they are both couplings of X and Y.

4.21 Time Series

Suppose that there are two time periods.

- $t = 0$ "today"
- $t = 1$ "tomorrow"

Let $X(0)$ be a random variable to be realized at $t = 0$, $X(1)$ be a random variable to be realized at $t = 1$.

Suppose that

$$\text{Prob}\{X(0) = i, X(1) = j\} = f_{ij} \geq 0 \; i = 0, \cdots, I-1$$
$$\sum_i \sum_j f_{ij} = 1$$

f_{ij} is a joint distribution over $[X(0), X(1)]$.

A conditional distribution is

$$\text{Prob}\{X(1) = j | X(0) = i\} = \frac{f_{ij}}{\sum_j f_{ij}}$$

Remark:

- This is a key formula for a theory of optimally predicting a time series.

CHAPTER

FIVE

LLN AND CLT

Contents

5.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables, and
- the multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import random
import numpy as np
from scipy.stats import t, beta, lognorm, expon, gamma, uniform
from scipy.stats import gaussian_kde, poisson, binom, norm, chi2
from mpl_toolkits.mplot3d import Axes3D
```

(continues on next page)

(continued from previous page)

```
from matplotlib.collections import PolyCollection
from scipy.linalg import inv, sqrtm
```

5.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

5.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

5.3.1 The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as *Kolmogorov's strong law*.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution F.

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{5.1}$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (5.1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

5.3.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dud02].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If $X_1, \ldots, X_n$ is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{5.2}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i.

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{5.3}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{\left[\frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)\right]^2\right\} \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence.

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out.

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n.

Combining our last result with (5.3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{5.4}$$

The claim in (5.2) is now clear.

Of course, if the sequence $X_1, \ldots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms.

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large.

In other words, the LLN can still work if the sequence $X_1, \ldots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

5.3.3 Illustration

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases.

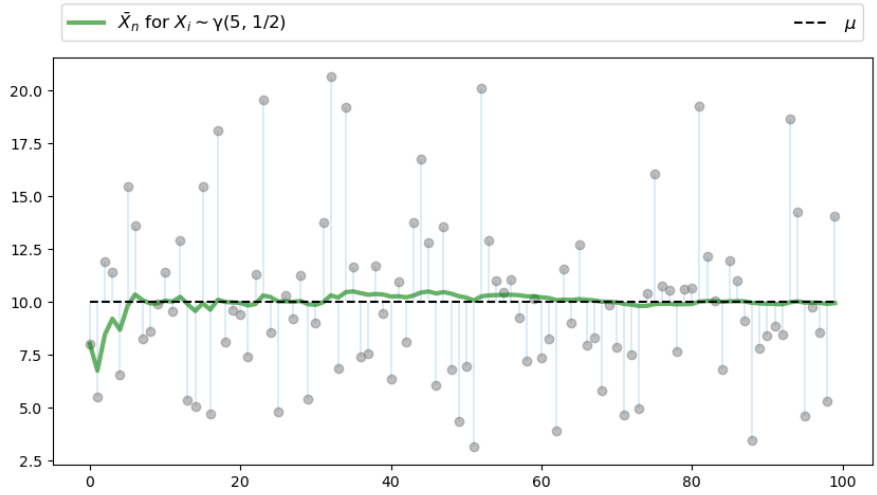
Below is a figure that does just this (as usual, you can click on it to expand it).

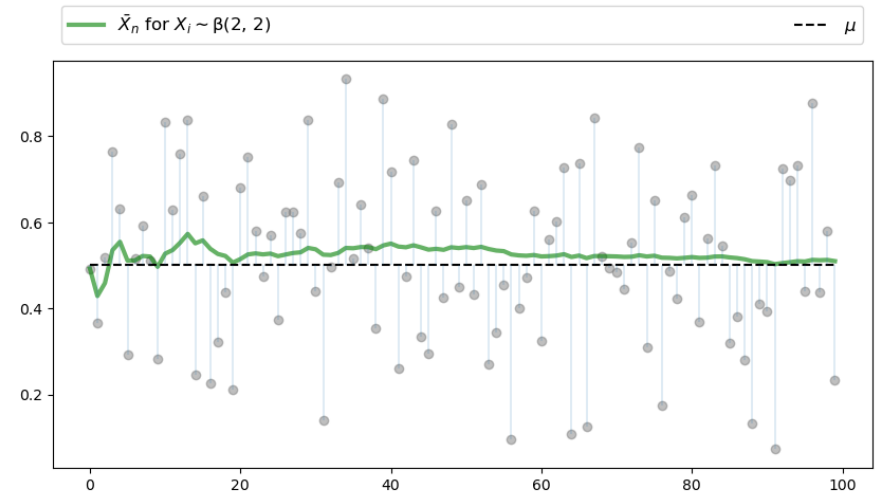
It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case.

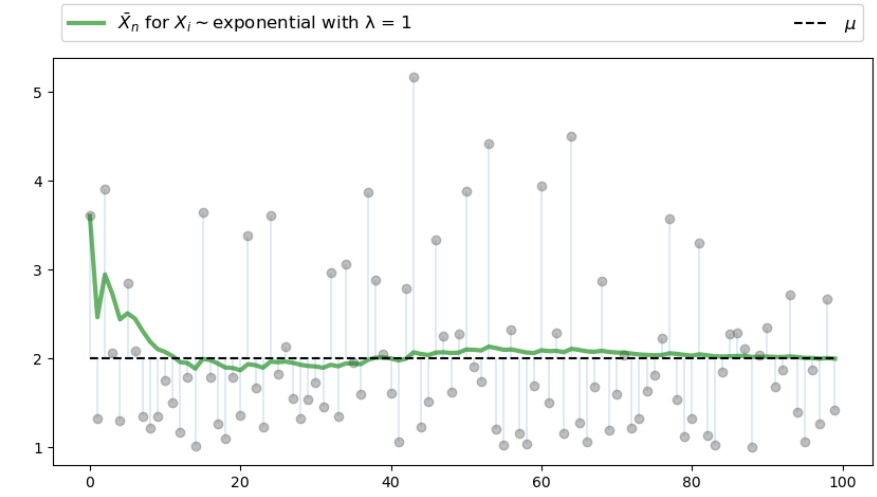
The dots represent the underlying observations X_i for $i = 1, \ldots, 100$.

In each of the three cases, convergence of $\bar{X}_n$ to μ occurs as predicted

```
n = 100

# Arbitrary collection of distributions
distributions = {"student's t with 10 degrees of freedom": t(10),
                 "β(2, 2)": beta(2, 2),
                 "lognormal LN(0, 1/2)": lognorm(0.5),
                 "γ(5, 1/2)": gamma(5, scale=2),
                 "poisson(4)": poisson(4),
                 "exponential with λ = 1": expon(1)}

# Create a figure and some axes
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 20))

# Set some plotting parameters to improve layout
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # Choose a randomly selected distribution
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

    # Generate n draws from the distribution
    data = distribution.rvs(n)

    # Compute sample mean at each n
    sample_mean = np.empty(n)
    for i in range(n):
        sample_mean[i] = np.mean(data[:i+1])

    # Plot
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = '$\\bar{X}_n$ for $X_i \sim$' + name
    ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6, label=axlabel)
    m = distribution.mean()
    ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label='$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```


$\bar{X}_n$ for $X_i \sim \gamma(5, 1/2)$
μ
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim \beta(2, 2)$
μ
0.8
0.6
0.4
0.2
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim$ exponential with $\lambda = 1$
μ
5
4
3
2
1
0
20
40
60
80
100

The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

5.4 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

5.4.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the classical IID setting, it tells us the following:

If the sequence $X_1, \ldots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{5.5}$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ.

5.4.2 Intuition

The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve.

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dud02]).

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition.

In fact, all of the proofs of the CLT that we know are similar in this respect.

Why does adding independent copies produce a bell-shaped distribution?

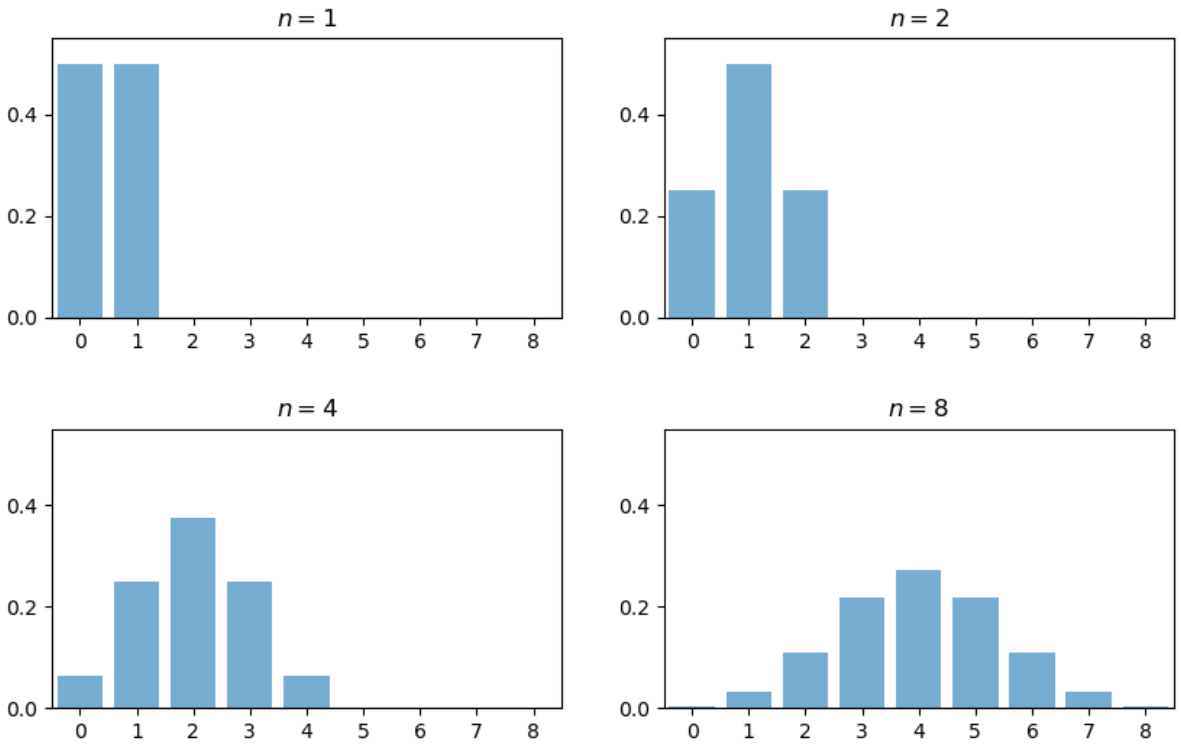
Part of the answer can be obtained by investigating the addition of independent Bernoulli random variables.

In particular, let X_i be binary, with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, and let $X_1, \ldots, X_n$ be independent.

Think of $X_i = 1$ as a "success", so that $Y_n = \sum_{i=1}^n X_i$ is the number of successes in n trials.

The next figure plots the probability mass function of Y_n for $n = 1, 2, 4, 8$

```
fig, axes = plt.subplots(2, 2, figsize=(10, 6))
plt.subplots_adjust(hspace=0.4)
axes = axes.flatten()
ns = [1, 2, 4, 8]
dom = list(range(9))

for ax, n in zip(axes, ns):
    b = binom(n, 0.5)
    ax.bar(dom, b.pmf(dom), alpha=0.6, align='center')
    ax.set(xlim=(-0.5, 8.5), ylim=(0, 0.55),
           xticks=list(range(9)), yticks=(0, 0.2, 0.4),
           title=f'$n = {n}$')

plt.show()
```



When $n = 1$, the distribution is flat — one success or no successes have the same probability.

When $n = 2$ we can either have 0, 1 or 2 successes.

Notice the peak in probability mass at the mid-point $k = 1$.

The reason is that there are more ways to get 1 success ("fail then succeed" or "succeed then fail") than to get zero or two successes.

Moreover, the two trials are independent, so the outcomes "fail then succeed" and "succeed then fail" are just as likely as the outcomes "fail then fail" and "succeed then succeed".

(If there was positive correlation, say, then "succeed then fail" would be less likely than "succeed then succeed")

Here, already we have the essence of the CLT: addition under independence leads probability mass to pile up in the middle and thin out at the tails.

For $n = 4$ and $n = 8$ we again get a peak at the "middle" value (halfway between the minimum and the maximum possible value).

The intuition is the same — there are simply more ways to get these middle outcomes.

If we continue, the bell-shaped curve becomes even more pronounced.

We are witnessing the binomial approximation of the normal distribution.

5.4.3 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build intuition.

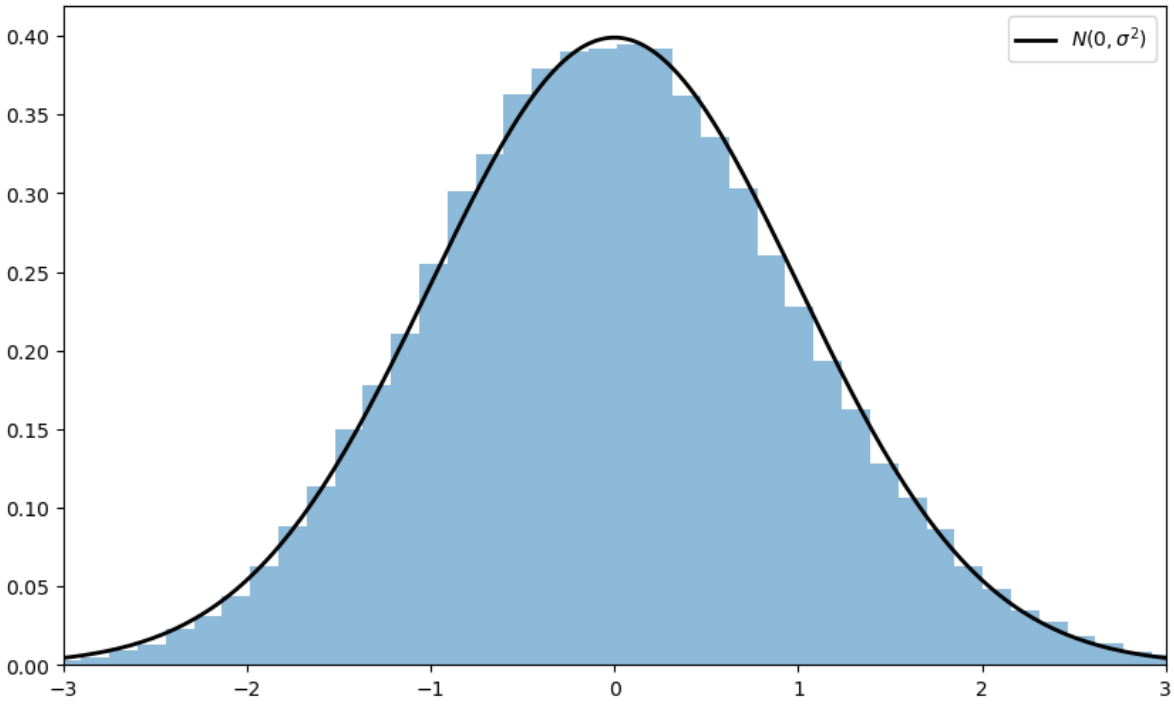
To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.
2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.
3. Use these draws to compute some measure of their distribution — such as a histogram.
4. Compare the latter to $N(0, \sigma^2)$.

Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment.)

```
# Set parameters
n = 250                  # Choice of n
k = 100000               # Number of draws of Y_n
distribution = expon(2)  # Exponential distribution, λ = 1/2
μ, s = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * s, 3 * s
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, norm.pdf(xgrid, scale=s), 'k-', lw=2, label='$N(0, \sigma^2)$')
ax.legend()

plt.show()
```



Notice the absence of for loops — every operation is vectorized, meaning that the major calculations are all shifted to highly optimized C code.

The fit to the normal density is already tight and can be further improved by increasing `n`.

You can also experiment with other specifications of F.

5.4.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases.

In the simulation, we'll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell-shaped curve.

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for f.)

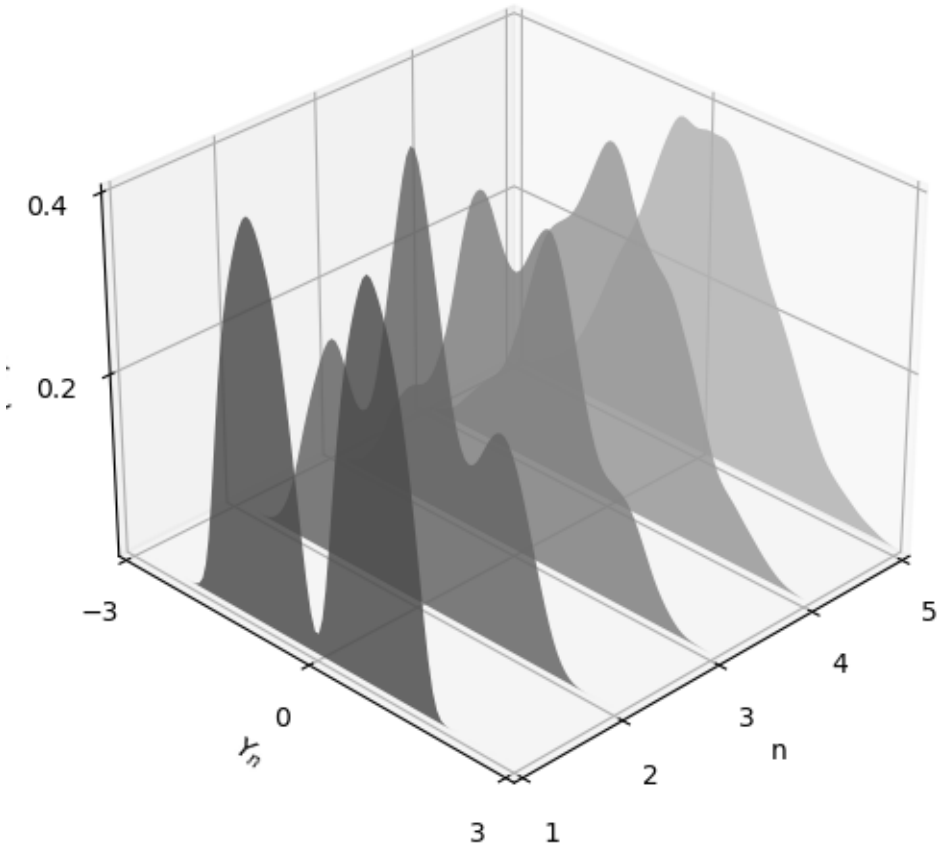
In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

```
beta_dist = beta(2, 2)

def gen_x_draws(k):
    """
    Returns a flat array containing k independent draws from the
    distribution of X, the underlying random variable.  This distribution
```

(continues on next page)

(continued from previous page)

```
    is itself a convex combination of three beta distributions.
    """
    bdraws = beta_dist.rvs((3, k))
    # Transform rows, so each represents a different distribution
    bdraws[0, :] -= 0.5
    bdraws[1, :] += 0.6
    bdraws[2, :] -= 1.1
    # Set X[i] = bdraws[j, i], where j is a random draw from {0, 1, 2}
    js = np.random.randint(0, 2, size=k)
    X = bdraws[js, np.arange(k)]
    # Rescale, so that the random variable is zero mean
    m, sigma = X.mean(), X.std()
    return (X - m) / sigma

nmax = 5
reps = 100000
ns = list(range(1, nmax + 1))

# Form a matrix Z such that each column is reps independent draws of X
Z = np.empty((reps, nmax))
for i in range(nmax):
    Z[:, i] = gen_x_draws(reps)
# Take cumulative sum across columns
S = Z.cumsum(axis=1)
# Multiply j-th column by sqrt j
Y = (1 / np.sqrt(ns)) * S

# Plot
ax = plt.figure(figsize = (10, 6)).add_subplot(projection='3d')

a, b = -3, 3
gs = 100
xs = np.linspace(a, b, gs)

# Build verts
greys = np.linspace(0.3, 0.7, nmax)
verts = []
for n in ns:
    density = gaussian_kde(Y[:, n-1])
    ys = density(xs)
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors=[str(g) for g in greys])
poly.set_alpha(0.85)
ax.add_collection3d(poly, zs=ns, zdir='x')

ax.set(xlim3d=(1, nmax), xticks=(ns), ylabel='$Y_n$', zlabel='$p(y_n)$',
       xlabel=("n"), yticks=((-3, 0, 3)), ylim3d=(a, b),
       zlim3d=(0, 0.4), zticks=((0.2, 0.4)))
ax.invert_xaxis()
# Rotates the plot 30 deg on z axis and 45 deg on x axis
ax.view_init(30, 45)
plt.show()
```



As expected, the distribution smooths out into a bell curve as n increases.

We leave you to investigate its contents if you wish to know more.

If you run the file from the ordinary IPython shell, the figure should pop up in a window that you can rotate with your mouse, giving different views on the density sequence.

5.4.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let's recall some elementary facts about random vectors.

A random vector $\mathbf{X}$ is just a sequence of k random variables $(X_1, \ldots, X_k)$.

Each realization of $\mathbf{X}$ is an element of $\mathbb{R}^k$.

A collection of random vectors $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in $\mathbb{R}^k$, we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \ldots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \ldots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \ldots, k$.

The expectation $\mathbb{E}[\mathbf{X}]$ of $\mathbf{X}$ is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu$$

The *variance-covariance matrix* of random vector $\mathbf{X}$ is defined as

$$\text{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']$$

Expanding this out, we get

$$\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The j, k-th term is the scalar covariance between X_j and X_k.

With this notation, we can proceed to the multivariate LLN and CLT.

Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$.

Let μ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$.

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{5.6}$$

Here $\bar{\mathbf{X}}_n \to \mu$ means that $\|\bar{\mathbf{X}}_n - \mu\| \to 0$, where $\|\cdot\|$ is the standard Euclidean norm.

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{5.7}$$

5.5 Exercises

Exercise 5.5.1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as *stated above*.

If $g \colon \mathbb{R} \to \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as} \quad n \to \infty \tag{5.8}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means.

(These kinds of results are often said to use the "delta method".)

The proof is based on a Taylor expansion of g around the point μ.

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$.

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program discussed *above*.

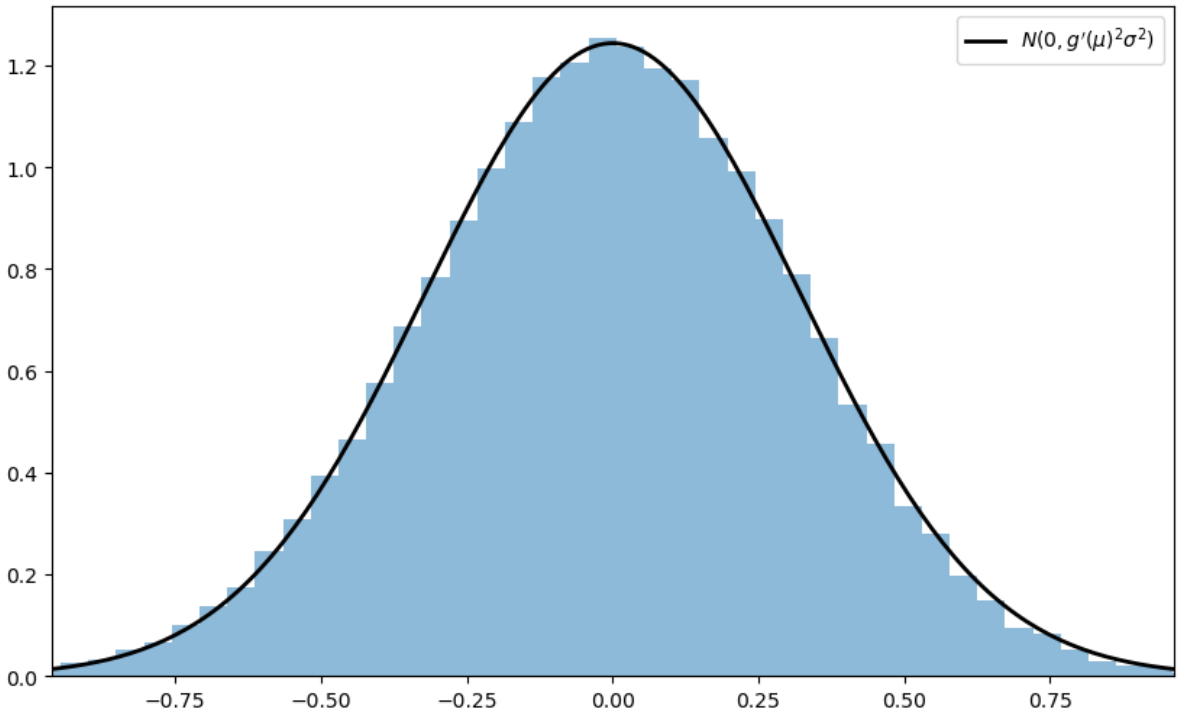
What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

Solution to Exercise 5.5.1

Here is one solution

```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

# Set parameters
n = 250
replications = 100000
distribution = uniform(loc=0, scale=(np.pi / 2))
μ, s = distribution.mean(), distribution.std()

g = np.sin
g_prime = np.cos

# Generate obs of sqrt{n} (g(X_n) - g(μ))
data = distribution.rvs((replications, n))
sample_means = data.mean(axis=1)  # Compute mean of each row
error_obs = np.sqrt(n) * (g(sample_means) - g(μ))

# Plot
asymptotic_sd = g_prime(μ) * s
fig, ax = plt.subplots(figsize=(10, 6))
xmin = -3 * g_prime(μ) * s
xmax = -xmin
ax.set_xlim(xmin, xmax)
ax.hist(error_obs, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
lb = "$N(0, g'(\mu)^2  \sigma^2)$"
ax.plot(xgrid, norm.pdf(xgrid, scale=asymptotic_sd), 'k-', lw=2, label=lb)
ax.legend()
plt.show()
```



What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

In this case, the mean μ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$.

Hence the conditions of the delta theorem are not satisfied.

Exercise 5.5.2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem.

If you study econometric theory, you will see this result used again and again.

Assume the setting of the multivariate CLT *discussed above*, so that

1. $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$.
2. $\mu := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$.
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{5.9}$$

is valid.

In a statistical setting, one often wants the right-hand side to be **standard** normal so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations.

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\operatorname{Var}[\mathbf{AX}] = \mathbf{A} \operatorname{Var}[\mathbf{X}] \mathbf{A}'$$

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{AZ}_n \xrightarrow{d} \mathbf{AZ}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of [?], then

$$\mathbf{Z}_n := \sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \xrightarrow{d} \|\mathbf{Z}\|^2$$

Given the distribution of $\mathbf{Z}$, we conclude that

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \tag{5.10}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom.

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors.)

Your second exercise is to illustrate the convergence in (5.10) with a simulation.

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$.
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$.
- U_i and W_i are independent of each other.

Hint:

1. `scipy.linalg.sqrtm(A)` computes the square root of `A`. You still need to invert it.
2. You should be able to work out Σ from the preceding information.

Solution to Exercise 5.5.2

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \mathbf{I})$$

This is straightforward given the facts presented in the exercise.

Let

$$\mathbf{Y}_n := \sqrt{n}(\bar{\mathbf{X}}_n - \mu) \quad \text{and} \quad \mathbf{Y} \sim N(\mathbf{0}, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y}$$

Since linear combinations of normal random variables are normal, the vector $\mathbf{QY}$ is also normal.

Its mean is clearly $\mathbf{0}$, and its variance-covariance matrix is

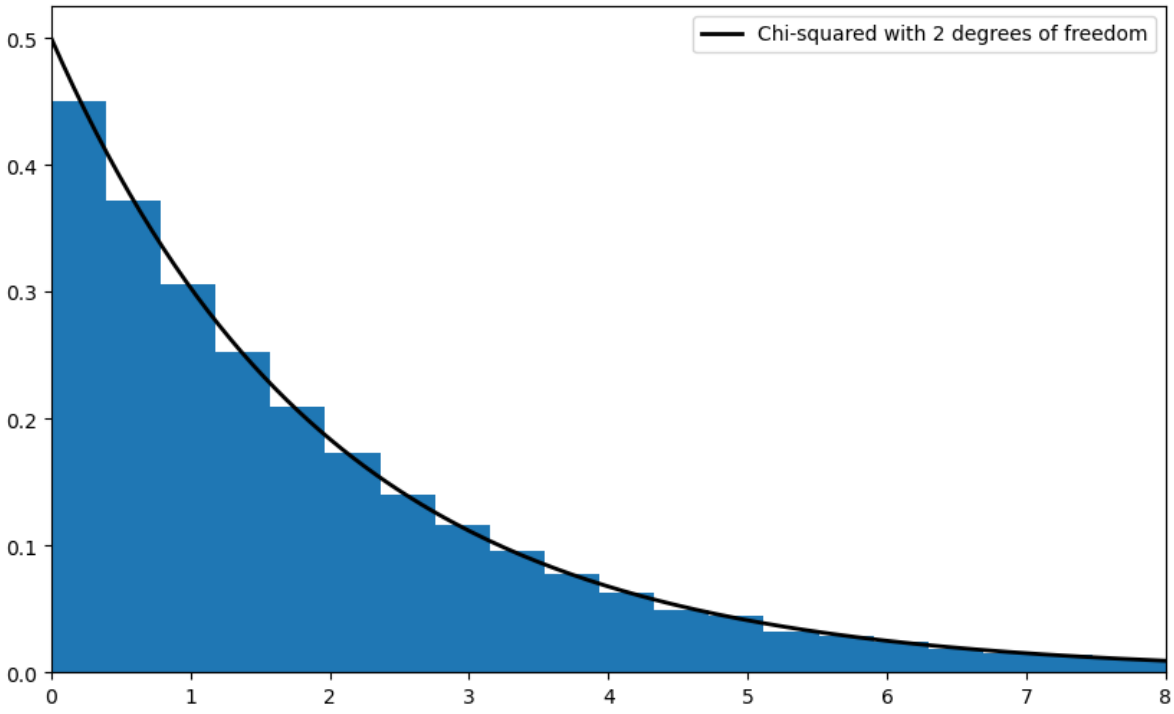
$$\text{Var}[\mathbf{QY}] = \mathbf{Q}\text{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{QY}_n \xrightarrow{d} \mathbf{QY} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

```
# Set parameters
n = 250
replications = 50000
dw = uniform(loc=-1, scale=2)  # Uniform(-1, 1)
du = uniform(loc=-2, scale=4)  # Uniform(-2, 2)
sw, su = dw.std(), du.std()
vw, vu = sw**2, su**2
Σ = ((vw, vw), (vw, vw + vu))
Σ = np.array(Σ)

# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# Generate observations of the normalized sample mean
error_obs = np.empty((2, replications))
for i in range(replications):
    # Generate one sequence of bivariate shocks
    X = np.empty((2, n))
    W = dw.rvs(n)
    U = du.rvs(n)
    # Construct the n observations of the random vector
    X[0, :] = W
    X[1, :] = W + U
    # Construct the i-th observation of Y_n
    error_obs[:, i] = np.sqrt(n) * X.mean(axis=1)

# Premultiply by Q and then take the squared norm
temp = Q @ error_obs
chisq_obs = np.sum(temp**2, axis=0)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmax = 8
ax.set_xlim(0, xmax)
xgrid = np.linspace(0, xmax, 200)
lb = "Chi-squared with 2 degrees of freedom"
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=lb)
ax.legend()
ax.hist(chisq_obs, bins=50, density=True)
plt.show()
```


Chi-squared with 2 degrees of freedom
0.5
0.4
0.3
0.2
0.1
0.0
0
1
2
3
4
5
6
7
8

CHAPTER

SIX

MULTIVARIATE NORMAL DISTRIBUTION

Contents

6.1 Overview

This lecture describes a workhorse in probability theory, statistics, and economics, namely, the **multivariate normal distribution**.

In this lecture, you will learn formulas for

- the joint distribution of a random vector x of length N
- marginal distributions for all subvectors of x
- conditional distributions for subvectors of x conditional on other subvectors of x

We will use the multivariate normal distribution to formulate some useful models:

- a factor analytic model of an intelligence quotient, i.e., IQ
- a factor analytic model of two independent inherent abilities, say, mathematical and verbal.
- a more general factor analytic model
- Principal Components Analysis (PCA) as an approximation to a factor analytic model
- time series generated by linear stochastic difference equations
- optimal linear filtering theory

6.2 The Multivariate Normal Distribution

This lecture defines a Python class `MultivariateNormal` to be used to generate **marginal** and **conditional** distributions associated with a multivariate normal distribution.

For a multivariate normal distribution it is very convenient that

- conditional expectations equal linear least squares projections
- conditional distributions are characterized by multivariate linear regressions

We apply our Python class to some examples.

We use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit
import statsmodels.api as sm
```

Assume that an $N \times 1$ random vector z has a multivariate normal probability density.

This means that the probability density takes the form

$$f(z;\mu,\Sigma) = (2\pi)^{-\left(\frac{N}{2}\right)} \det(\Sigma)^{-\frac{1}{2}} \exp\left(-.5\,(z-\mu)'\,\Sigma^{-1}\,(z-\mu)\right)$$

where $\mu = Ez$ is the mean of the random vector z and $\Sigma = E\,(z-\mu)\,(z-\mu)'$ is the covariance matrix of z.

The covariance matrix Σ is symmetric and positive definite.

```
@njit
def f(z, μ, Σ):
    """
    The density function of multivariate normal distribution.

    Parameters
    ---------------
    z: ndarray(float, dim=2)
        random vector, N by 1
    μ: ndarray(float, dim=1 or 2)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1
    """

    z = np.atleast_2d(z)
```

(continues on next page)

(continued from previous page)

```
    μ = np.atleast_2d(μ)
    Σ = np.atleast_2d(Σ)

    N = z.size

    temp1 = np.linalg.det(Σ) ** (-1/2)
    temp2 = np.exp(-.5 * (z - μ).T @ np.linalg.inv(Σ) @ (z - μ))

    return (2 * np.pi) ** (-N/2) * temp1 * temp2
```

For some integer $k \in \{1, \ldots, N-1\}$, partition z as

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix},$$

where z_1 is an $(N-k) \times 1$ vector and z_2 is a $k \times 1$ vector.

Let

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

be corresponding partitions of μ and Σ.

The **marginal** distribution of z_1 is

- multivariate normal with mean μ_1 and covariance matrix Σ_{11}.

The **marginal** distribution of z_2 is

- multivariate normal with mean μ_2 and covariance matrix Σ_{22}.

The distribution of z_1 **conditional** on z_2 is

- multivariate normal with mean

$$\hat{\mu}_1 = \mu_1 + \beta (z_2 - \mu_2)$$

and covariance matrix

$$\hat{\Sigma}_{11} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} = \Sigma_{11} - \beta\Sigma_{22}\beta'$$

where

$$\beta = \Sigma_{12}\Sigma_{22}^{-1}$$

is an $(N-k) \times k$ matrix of **population regression coefficients** of the $(N-k) \times 1$ random vector $z_1 - \mu_1$ on the $k \times 1$ random vector $z_2 - \mu_2$.

The following class constructs a multivariate normal distribution instance with two methods.

- a method `partition` computes β, taking k as an input
- a method `cond_dist` computes either the distribution of z_1 conditional on z_2 or the distribution of z_2 conditional on z_1

```
class MultivariateNormal:
    """
    Class of multivariate normal distribution.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    μ: ndarray(float, dim=1)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1

    Arguments
    ---------
    μ, Σ:
        see parameters
    μs: list(ndarray(float, dim=1))
        list of mean vectors μ1 and μ2 in order
    Σs: list(list(ndarray(float, dim=2)))
        2 dimensional list of covariance matrices
        Σ11, Σ12, Σ21, Σ22 in order
    βs: list(ndarray(float, dim=1))
        list of regression coefficients β1 and β2 in order
    """

    def __init__(self, μ, Σ):
        "initialization"
        self.μ = np.array(μ)
        self.Σ = np.atleast_2d(Σ)

    def partition(self, k):
        """
        Given k, partition the random vector z into a size k vector z1
        and a size N-k vector z2. Partition the mean vector μ into
        μ1 and μ2, and the covariance matrix Σ into Σ11, Σ12, Σ21, Σ22
        correspondingly. Compute the regression coefficients β1 and β2
        using the partitioned arrays.
        """
        μ = self.μ
        Σ = self.Σ

        self.μs = [μ[:k], μ[k:]]
        self.Σs = [[Σ[:k, :k], Σ[:k, k:]],
                   [Σ[k:, :k], Σ[k:, k:]]]

        self.βs = [self.Σs[0][1] @ np.linalg.inv(self.Σs[1][1]),
                   self.Σs[1][0] @ np.linalg.inv(self.Σs[0][0])]

    def cond_dist(self, ind, z):
        """
        Compute the conditional distribution of z1 given z2, or reversely.
        Argument ind determines whether we compute the conditional
        distribution of z1 (ind=0) or z2 (ind=1).

        Returns
        ---------
        μ_hat: ndarray(float, ndim=1)
            The conditional mean of z1 or z2.
        Σ_hat: ndarray(float, ndim=2)
            The conditional covariance matrix of z1 or z2.
        """
```

(continues on next page)

(continued from previous page)

```
        β = self.βs[ind]
        μs = self.μs
        Σs = self.Σs

        μ_hat = μs[ind] + β @ (z - μs[1-ind])
        Σ_hat = Σs[ind][ind] - β @ Σs[1-ind][1-ind] @ β.T

        return μ_hat, Σ_hat
```

Let's put this code to work on a suite of examples.

We begin with a simple bivariate example; after that we'll turn to a trivariate example.

We'll compute population moments of some conditional distributions using our `MultivariateNormal` class.

For fun we'll also compute sample analogs of the associated population regressions by generating simulations and then computing linear least squares regressions.

We'll compare those linear least squares regressions for the simulated data to their population counterparts.

6.3 Bivariate Example

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} .5 \\ 1.0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & .5 \\ .5 & 1 \end{bmatrix}$$

```
μ = np.array([.5, 1.])
Σ = np.array([[1., .5], [.5 ,1.]])

# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
k = 1 # choose partition

# partition and compute regression coefficients
multi_normal.partition(k)
multi_normal.βs[0],multi_normal.βs[1]
```

```
(array([[0.5]]), array([[0.5]]))
```

Let's illustrate the fact that you *can regress anything on anything else*.

We have computed everything we need to compute two regression lines, one of z_2 on z_1, the other of z_1 on z_2.

We'll represent these regressions as

$$z_1 = a_1 + b_1 z_2 + \epsilon_1$$

and

$$z_2 = a_2 + b_2 z_1 + \epsilon_2$$

where we have the population least squares orthogonality conditions

$$E \epsilon_1 z_2 = 0$$

and

$$E\epsilon_2 z_1 = 0$$

Let's compute a_1, a_2, b_1, b_2.

```
beta = multi_normal.βs

a1 = μ[0] - beta[0]*μ[1]
b1 = beta[0]

a2 = μ[1] - beta[1]*μ[0]
b2 = beta[1]
```

Let's print out the intercepts and slopes.

For the regression of z_1 on z_2 we have

```
print ("a1 = ", a1)
print ("b1 = ", b1)
```

```
a1 =  [[0.]]
b1 =  [[0.5]]
```

For the regression of z_2 on z_1 we have

```
print ("a2 = ", a2)
print ("b2 = ", b2)
```

```
a2 =  [[0.75]]
b2 =  [[0.5]]
```

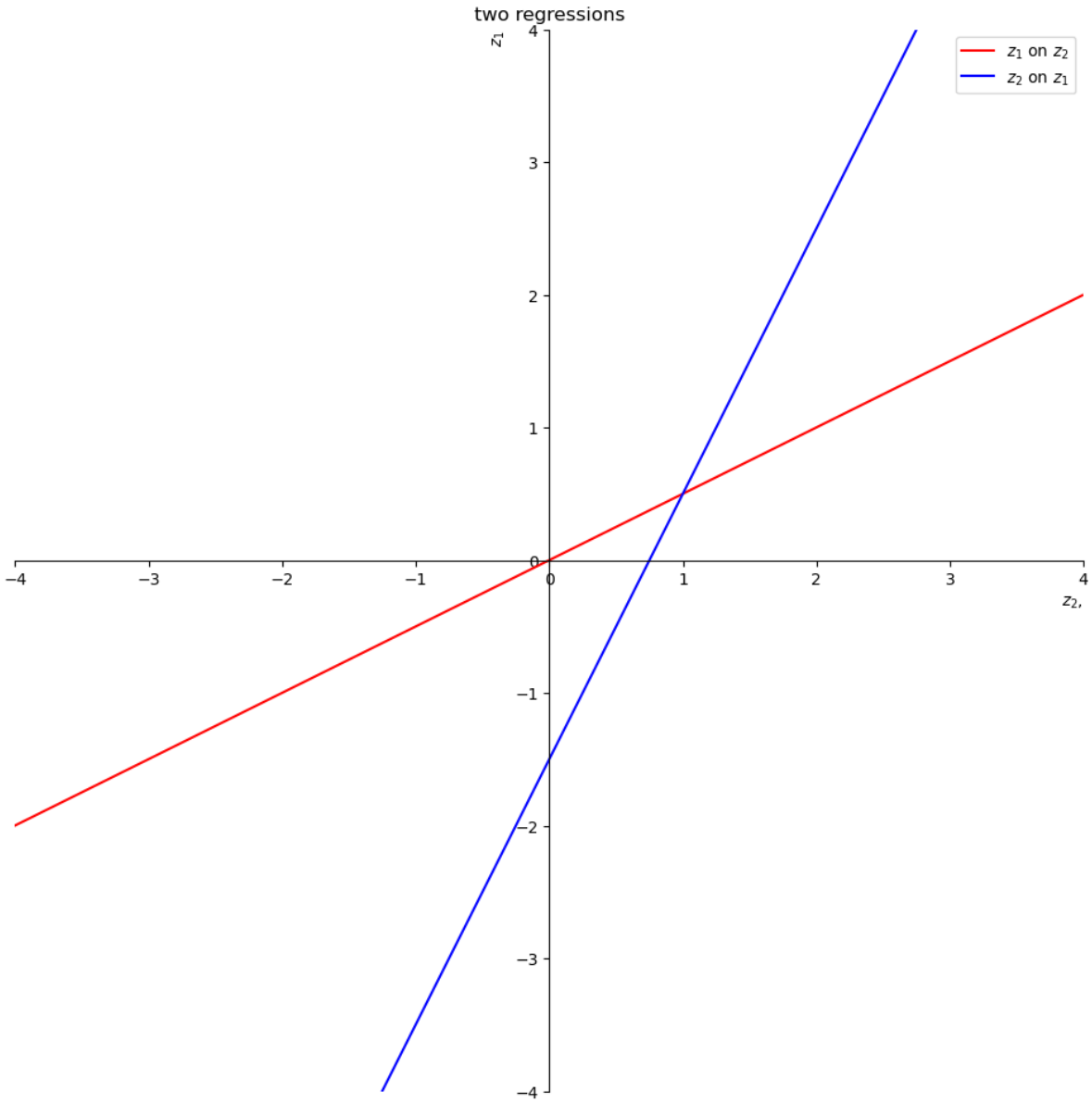
Now let's plot the two regression lines and stare at them.

```
z2 = np.linspace(-4,4,100)


a1 = np.squeeze(a1)
b1 = np.squeeze(b1)

a2 = np.squeeze(a2)
b2 = np.squeeze(b2)

z1  = b1*z2 + a1


z1h = z2/b2 - a2/b2


fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(1, 1, 1)
ax.set(xlim=(-4, 4), ylim=(-4, 4))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
```

(continues on next page)

(continued from previous page)

```
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
plt.ylabel('$z_1$', loc = 'top')
plt.xlabel('$z_2$,', loc = 'right')
plt.title('two regressions')
plt.plot(z2,z1, 'r', label = "$z_1$ on $z_2$")
plt.plot(z2,z1h, 'b', label = "$z_2$ on $z_1$")
plt.legend()
plt.show()
```



The red line is the expectation of z_1 conditional on z_2.

The intercept and slope of the red line are

```
print("a1 = ", a1)
print("b1 = ", b1)
```

```
a1 =  0.0
b1 =  0.5
```

The blue line is the expectation of z_2 conditional on z_1.

The intercept and slope of the blue line are

```
print("-a2/b2 = ", - a2/b2)
print("1/b2 = ", 1/b2)
```

```
-a2/b2 =  -1.5
1/b2 =  2.0
```

We can use these regression lines or our code to compute conditional expectations.

Let's compute the mean and variance of the distribution of z_2 conditional on $z_1 = 5$.

After that we'll reverse what are on the left and right sides of the regression.

```
# compute the cond. dist. of z1
ind = 1
z1 = np.array([5.]) # given z1

μ2_hat, Σ2_hat = multi_normal.cond_dist(ind, z1)
print('μ2_hat, Σ2_hat = ', μ2_hat, Σ2_hat)
```

```
μ2_hat, Σ2_hat =  [3.25] [[0.75]]
```

Now let's compute the mean and variance of the distribution of z_1 conditional on $z_2 = 5$.

```
# compute the cond. dist. of z1
ind = 0
z2 = np.array([5.]) # given z2

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
print('μ1_hat, Σ1_hat = ', μ1_hat, Σ1_hat)
```

```
μ1_hat, Σ1_hat =  [2.5] [[0.75]]
```

Let's compare the preceding population mean and variance with outcomes from drawing a large sample and then regressing $z_1 - \mu_1$ on $z_2 - \mu_2$.

We know that

$$Ez_1|z_2 = (\mu_1 - \beta\mu_2) + \beta z_2$$

which can be arranged to

$$z_1 - \mu_1 = \beta\left(z_2 - \mu_2\right) + \epsilon,$$

We anticipate that for larger and larger sample sizes, estimated OLS coefficients will converge to β and the estimated variance of ϵ will converge to $\hat{\Sigma}_1$.

```
n = 1_000_000 # sample size

# simulate multivariate normal random vectors
data = np.random.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, 0]
z2_data = data[:, 1]

# OLS regression
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

Let's compare the preceding population β with the OLS sample estimate on $z_2 - \mu_2$

```
multi_normal.βs[0], results.params
```

```
(array([[0.5]]), array([0.49922311]))
```

Let's compare our population $\hat{\Sigma}_1$ with the degrees-of-freedom adjusted estimate of the variance of ϵ

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.75]]), 0.7498096649038325)
```

Lastly, let's compute the estimate of $\hat{Ez_1|z_2}$ and compare it with $\hat{\mu}_1$

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.5]), array([2.49689244]))
```

Thus, in each case, for our very large sample size, the sample analogues closely approximate their population counterparts.

A Law of Large Numbers explains why sample analogues approximate population objects.

6.4 Trivariate Example

Let's apply our code to a trivariate example.

We'll specify the mean vector and the covariance matrix as follows.

```
μ = np.random.random(3)
C = np.random.random((3, 3))
Σ = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(μ, Σ)
```

```
μ, Σ
```

```
(array([0.59342983, 0.45695387, 0.78249021]),
 array([[0.80236993, 0.77482758, 0.83014953],
        [0.77482758, 0.88669328, 0.78034355],
        [0.83014953, 0.78034355, 1.32023878]]))
```

```
k = 1
multi_normal.partition(k)
```

Let's compute the distribution of z_1 conditional on $z_2 = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$.

```
ind = 0
z2 = np.array([2., 5.])

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
```

```
n = 1_000_000
data = np.random.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, :k]
z2_data = data[:, k:]
```

```
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

As above, we compare population and sample regression coefficients, the conditional covariance matrix, and the conditional mean vector in that order.

```
multi_normal.βs[0], results.params
```

```
(array([[0.66788031, 0.23402845]]), array([0.66746885, 0.23422764]))
```

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.09059923]]), 0.09067354818597988)
```

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.61101726]), array([2.61122244]))
```

Once again, sample analogues do a good job of approximating their populations counterparts.

6.5 One Dimensional Intelligence (IQ)

Let's move closer to a real-life example, namely, inferring a one-dimensional measure of intelligence called IQ from a list of test scores.

The ith test score y_i equals the sum of an unknown scalar IQ θ and a random variable w_i.

$$y_i = \theta + \sigma_y w_i, \quad i = 1, \ldots, n$$

The distribution of IQ's for a cross-section of people is a normal random variable described by

$$\theta = \mu_\theta + \sigma_\theta w_{n+1}.$$

We assume that the noises $\{w_i\}_{i=1}^N$ in the test scores are IID and not correlated with IQ.

We also assume that $\{w_i\}_{i=1}^{n+1}$ are i.i.d. standard normal:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} \sim N\left(0, I_{n+1}\right)$$

The following system describes the $(n+1) \times 1$ random vector X that interests us:

$$X = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ \theta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \vdots \\ \mu_\theta \\ \mu_\theta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & \cdots & 0 & \sigma_\theta \\ 0 & \sigma_y & \cdots & 0 & \sigma_\theta \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_y & \sigma_\theta \\ 0 & 0 & \cdots & 0 & \sigma_\theta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix},$$

or equivalently,

$$X = \mu_\theta 1_{n+1} + Dw$$

where $X = \begin{bmatrix} y \\ \theta \end{bmatrix}$, 1_{n+1} is a vector of 1s of size $n+1$, and D is an $n+1$ by $n+1$ matrix.

Let's define a Python function that constructs the mean μ and covariance matrix Σ of the random vector X that we know is governed by a multivariate normal distribution.

As arguments, the function takes the number of tests n, the mean μ_θ and the standard deviation σ_θ of the IQ distribution, and the standard deviation of the randomness in test scores σ_y.

```
def construct_moments_IQ(n, μθ, σθ, σy):

    μ_IQ = np.full(n+1, μθ)

    D_IQ = np.zeros((n+1, n+1))
    D_IQ[range(n), range(n)] = σy
    D_IQ[:, n] = σθ

    Σ_IQ = D_IQ @ D_IQ.T

    return μ_IQ, Σ_IQ, D_IQ
```

Now let's consider a specific instance of this model.

Assume we have recorded 50 test scores and we know that $\mu_\theta = 100$, $\sigma_\theta = 10$, and $\sigma_y = 10$.

We can compute the mean vector and covariance matrix of X easily with our `construct_moments_IQ` function as follows.

```
n = 50
μθ, σθ, σy = 100., 10., 10.

μ_IQ, Σ_IQ, D_IQ = construct_moments_IQ(n, μθ, σθ, σy)
μ_IQ, Σ_IQ, D_IQ
```

```
(array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
```

(continues on next page)

(continued from previous page)

```
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100.]),
 array([[200., 100., 100., ..., 100., 100., 100.],
        [100., 200., 100., ..., 100., 100., 100.],
        [100., 100., 200., ..., 100., 100., 100.],
        ...,
        [100., 100., 100., ..., 200., 100., 100.],
        [100., 100., 100., ..., 100., 200., 100.],
        [100., 100., 100., ..., 100., 100., 100.]]),
 array([[10.,  0.,  0., ...,  0.,  0., 10.],
        [ 0., 10.,  0., ...,  0.,  0., 10.],
        [ 0.,  0., 10., ...,  0.,  0., 10.],
        ...,
        [ 0.,  0.,  0., ..., 10.,  0., 10.],
        [ 0.,  0.,  0., ...,  0., 10., 10.],
        [ 0.,  0.,  0., ...,  0.,  0., 10.]]))
```

We can now use our `MultivariateNormal` class to construct an instance, then partition the mean vector and covariance matrix as we wish.

We want to regress IQ, the random variable θ (*what we don't know*), on the vector y of test scores (*what we do know*).

We choose `k=n` so that $z_1 = y$ and $z_2 = \theta$.

```
multi_normal_IQ = MultivariateNormal(μ_IQ, Σ_IQ)

k = n
multi_normal_IQ.partition(k)
```

Using the generator `multivariate_normal`, we can make one draw of the random vector from our distribution and then compute the distribution of θ conditional on our test scores.

Let's do that and then print out some pertinent quantities.

```
x = np.random.multivariate_normal(μ_IQ, Σ_IQ)
y = x[:-1] # test scores
θ = x[-1]  # IQ
```

```
# the true value
θ
```

```
119.26413202086053
```

The method `cond_dist` takes test scores y as input and returns the conditional normal distribution of the IQ θ.

In the following code, `ind` sets the variables on the right side of the regression.

Given the way we have defined the vector X, we want to set `ind=1` in order to make θ the left side variable in the population regression.

```
ind = 1
multi_normal_IQ.cond_dist(ind, y)
```

```
(array([118.35606013]), array([[1.96078431]]))
```

The first number is the conditional mean $\hat{\mu}_\theta$ and the second is the conditional variance $\hat{\Sigma}_\theta$.

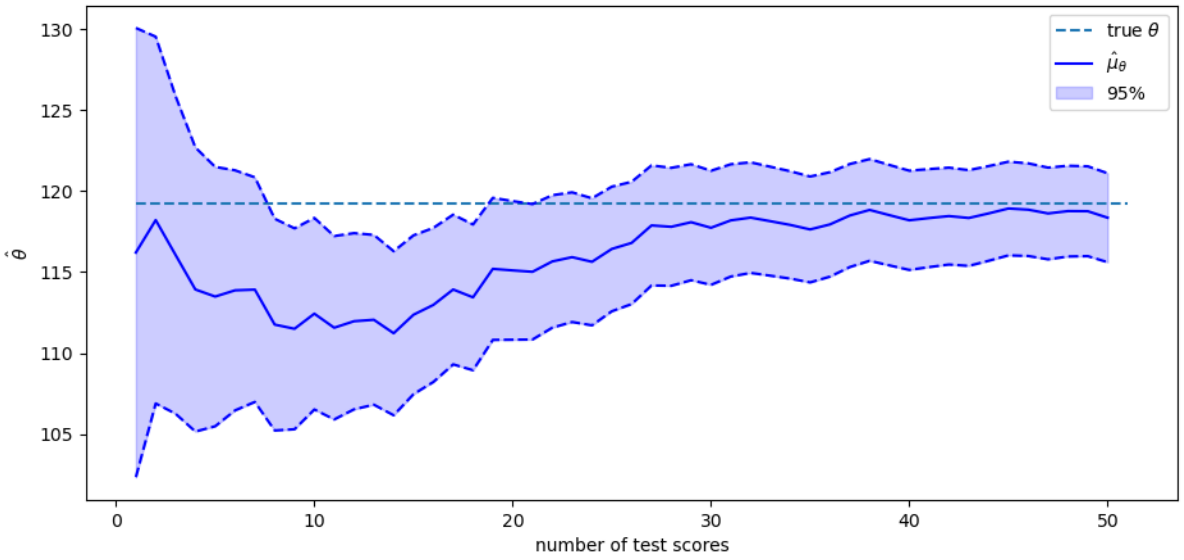
How do additional test scores affect our inferences?

To shed light on this, we compute a sequence of conditional distributions of θ by varying the number of test scores in the conditioning set from 1 to n.

We'll make a pretty graph showing how our judgment of the person's IQ change as more test results come in.

```
# array for containing moments
μθ_hat_arr = np.empty(n)
Σθ_hat_arr = np.empty(n)

# loop over number of test scores
for i in range(1, n+1):
    # construction of multivariate normal distribution instance
    μ_IQ_i, Σ_IQ_i, D_IQ_i = construct_moments_IQ(i, μθ, σθ, σy)
    multi_normal_IQ_i = MultivariateNormal(μ_IQ_i, Σ_IQ_i)

    # partition and compute conditional distribution
    multi_normal_IQ_i.partition(i)
    scores_i = y[:i]
    μθ_hat_i, Σθ_hat_i = multi_normal_IQ_i.cond_dist(1, scores_i)

    # store the results
    μθ_hat_arr[i-1] = μθ_hat_i[0]
    Σθ_hat_arr[i-1] = Σθ_hat_i[0, 0]

# transform variance to standard deviation
σθ_hat_arr = np.sqrt(Σθ_hat_arr)
```

```
μθ_hat_lower = μθ_hat_arr - 1.96 * σθ_hat_arr
μθ_hat_higher = μθ_hat_arr + 1.96 * σθ_hat_arr

plt.hlines(θ, 1, n+1, ls='--', label='true $θ$')
plt.plot(range(1, n+1), μθ_hat_arr, color='b', label='$\hat{μ}_{θ}$')
plt.plot(range(1, n+1), μθ_hat_lower, color='b', ls='--')
plt.plot(range(1, n+1), μθ_hat_higher, color='b', ls='--')
plt.fill_between(range(1, n+1), μθ_hat_lower, μθ_hat_higher,
                 color='b', alpha=0.2, label='95%')

plt.xlabel('number of test scores')
plt.ylabel('$\hat{θ}$')
plt.legend()

plt.show()
```



The solid blue line in the plot above shows $\hat{\mu}_\theta$ as a function of the number of test scores that we have recorded and conditioned on.

The blue area shows the span that comes from adding or subtracting $1.96\hat{\sigma}_\theta$ from $\hat{\mu}_\theta$.

Therefore, 95% of the probability mass of the conditional distribution falls in this range.

The value of the random θ that we drew is shown by the black dotted line.

As more and more test scores come in, our estimate of the person's θ become more and more reliable.

By staring at the changes in the conditional distributions, we see that adding more test scores makes $\hat{\theta}$ settle down and approach θ.

Thus, each y_i adds information about θ.

If we were to drive the number of tests $n \to +\infty$, the conditional standard deviation $\hat{\sigma}_\theta$ would converge to 0 at rate $\frac{1}{n^{.5}}$.

6.6 Information as Surprise

By using a different representation, let's look at things from a different perspective.

We can represent the random vector X defined above as

$$X = \mu_\theta 1_{n+1} + C\epsilon, \quad \epsilon \sim N(0, I)$$

where C is a lower triangular **Cholesky factor** of Σ so that

$$\Sigma \equiv DD' = CC'$$

and

$$E\epsilon\epsilon' = I.$$

It follows that

$$\epsilon \sim N(0, I).$$

Let $G = C^{-1}$

G is also lower triangular.

We can compute ϵ from the formula

$$\epsilon = G\left(X - \mu_\theta 1_{n+1}\right)$$

This formula confirms that the orthonormal vector ϵ contains the same information as the non-orthogonal vector $(X - \mu_\theta 1_{n+1})$.

We can say that ϵ is an orthogonal basis for $(X - \mu_\theta 1_{n+1})$.

Let c_i be the ith element in the last row of C.

Then we can write

$$\theta = \mu_\theta + c_1\epsilon_1 + c_2\epsilon_2 + \cdots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \tag{6.1}$$

The mutual orthogonality of the ϵ_i's provides us with an informative way to interpret them in light of equation (6.1).

Thus, relative to what is known from tests $i = 1, \ldots, n-1$, $c_i\epsilon_i$ is the amount of **new information** about θ brought by the test number i.

Here **new information** means **surprise** or what could not be predicted from earlier information.

Formula (6.1) also provides us with an enlightening way to express conditional means and conditional variances that we computed earlier.

In particular,

$$E\left[\theta \mid y_1, \ldots, y_k\right] = \mu_\theta + c_1\epsilon_1 + \cdots + c_k\epsilon_k$$

and

$$Var\left(\theta \mid y_1, \ldots, y_k\right) = c_{k+1}^2 + c_{k+2}^2 + \cdots + c_{n+1}^2.$$

```
C = np.linalg.cholesky(Σ_IQ)
G = np.linalg.inv(C)

ε = G @ (x - μθ)
```

```
cε = C[n, :] * ε

# compute the sequence of μ9 and Σ9 conditional on y1, y2, ..., yk
μθ_hat_arr_C = np.array([np.sum(cε[:k+1]) for k in range(n)]) + μθ
Σθ_hat_arr_C = np.array([np.sum(C[n, i+1:n+1] ** 2) for i in range(n)])
```

To confirm that these formulas give the same answers that we computed earlier, we can compare the means and variances of θ conditional on $\{y_i\}_{i=1}^k$ with what we obtained above using the formulas implemented in the class `MultivariateNormal` built on our original representation of conditional distributions for multivariate normal distributions.

```
# conditional mean
np.max(np.abs(μθ_hat_arr - μθ_hat_arr_C)) < 1e-10
```

```
True
```

```
# conditional variance
np.max(np.abs(Σθ_hat_arr - Σθ_hat_arr_C)) < 1e-10
```

```
True
```

6.7 Cholesky Factor Magic

Evidently, the Cholesky factorizations automatically computes the population **regression coefficients** and associated statistics that are produced by our `MultivariateNormal` class.

The Cholesky factorization computes these things **recursively**.

Indeed, in formula (6.1),

- the random variable $c_i \epsilon_i$ is information about θ that is not contained by the information in $\epsilon_1, \epsilon_2, \ldots, \epsilon_{i-1}$
- the coefficient c_i is the simple population regression coefficient of $\theta - \mu_\theta$ on ϵ_i

6.8 Math and Verbal Intelligence

We can alter the preceding example to be more realistic.

There is ample evidence that IQ is not a scalar.

Some people are good in math skills but poor in language skills.

Other people are good in language skills but poor in math skills.

So now we shall assume that there are two dimensions of IQ, θ and η.

These determine average performances in math and language tests, respectively.

We observe math scores $\{y_i\}_{i=1}^n$ and language scores $\{y_i\}_{i=n+1}^{2n}$.

When $n = 2$, we assume that outcomes are draws from a multivariate normal distribution with representation

$$X = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \theta \\ \eta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \mu_\eta \\ \mu_\eta \\ \mu_\theta \\ \mu_\eta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & \sigma_y & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & \sigma_y & 0 & 0 & \sigma_\eta \\ 0 & 0 & 0 & \sigma_y & 0 & \sigma_\eta \\ 0 & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_\eta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}$$

where $w \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_6 \end{bmatrix}$ is a standard normal random vector.

We construct a Python function `construct_moments_IQ2d` to construct the mean vector and covariance matrix of the joint normal distribution.

```
def construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy):

    μ_IQ2d = np.empty(2*(n+1))
    μ_IQ2d[:n] = μθ
    μ_IQ2d[2*n] = μθ
    μ_IQ2d[n:2*n] = μη
    μ_IQ2d[2*n+1] = μη
```

(continues on next page)

(continued from previous page)

```
    D_IQ2d = np.zeros((2*(n+1), 2*(n+1)))
    D_IQ2d[range(2*n), range(2*n)] = σy
    D_IQ2d[:n, 2*n] = σθ
    D_IQ2d[2*n, 2*n] = σθ
    D_IQ2d[n:2*n, 2*n+1] = ση
    D_IQ2d[2*n+1, 2*n+1] = ση

    Σ_IQ2d = D_IQ2d @ D_IQ2d.T

    return μ_IQ2d, Σ_IQ2d, D_IQ2d
```

Let's put the function to work.

```
n = 2
# mean and variance of θ, η, and y
μθ, σθ, μη, ση, σy = 100., 10., 100., 10, 10

μ_IQ2d, Σ_IQ2d, D_IQ2d = construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy)
μ_IQ2d, Σ_IQ2d, D_IQ2d
```

```
(array([100., 100., 100., 100., 100., 100.]),
 array([[200., 100.,   0.,   0., 100.,   0.],
        [100., 200.,   0.,   0., 100.,   0.],
        [  0.,   0., 200., 100.,   0., 100.],
        [  0.,   0., 100., 200.,   0., 100.],
        [100., 100.,   0.,   0., 100.,   0.],
        [  0.,   0., 100., 100.,   0., 100.]]),
 array([[10.,  0.,  0.,  0., 10.,  0.],
        [ 0., 10.,  0.,  0., 10.,  0.],
        [ 0.,  0., 10.,  0.,  0., 10.],
        [ 0.,  0.,  0., 10.,  0., 10.],
        [ 0.,  0.,  0.,  0., 10.,  0.],
        [ 0.,  0.,  0.,  0.,  0., 10.]]))
```

```
# take one draw
x = np.random.multivariate_normal(μ_IQ2d, Σ_IQ2d)
y1 = x[:n]
y2 = x[n:2*n]
θ = x[2*n]
η = x[2*n+1]

# the true values
θ, η
```

```
(95.65332814847206, 74.75924045741095)
```

We first compute the joint normal distribution of (θ, η).

```
multi_normal_IQ2d = MultivariateNormal(μ_IQ2d, Σ_IQ2d)

k = 2*n # the length of data vector
```

(continues on next page)

(continued from previous page)

```
multi_normal_IQ2d.partition(k)

multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

```
(array([94.40352908, 73.37649136]),
 array([[33.33333333,  0.        ],
        [ 0.        , 33.33333333]]))
```

Now let's compute distributions of θ and μ separately conditional on various subsets of test scores.

It will be fun to compare outcomes with the help of an auxiliary function `cond_dist_IQ2d` that we now construct.

```
def cond_dist_IQ2d(μ, Σ, data):

    n = len(μ)

    multi_normal = MultivariateNormal(μ, Σ)
    multi_normal.partition(n-1)
    μ_hat, Σ_hat = multi_normal.cond_dist(1, data)

    return μ_hat, Σ_hat
```

Let's see how things work for an example.

```
for indices, IQ, conditions in [([*range(2*n), 2*n], 'θ', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n], 'θ', 'y1, y2'),
                                ([*range(n, 2*n), 2*n], 'θ', 'y3, y4'),
                                ([*range(2*n), 2*n+1], 'η', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n+1], 'η', 'y1, y2'),
                                ([*range(n, 2*n), 2*n+1], 'η', 'y3, y4')]:

    μ_hat, Σ_hat = cond_dist_IQ2d(μ_IQ2d[indices], Σ_IQ2d[indices][:, indices],␣
 ↪x[indices[:-1]])
    print(f'The mean and variance of {IQ} conditional on {conditions: <15} are ' +
          f'{μ_hat[0]:1.2f} and {Σ_hat[0, 0]:1.2f} respectively')
```

```
The mean and variance of θ conditional on y1, y2, y3, y4  are 94.40 and 33.33␣
 ↪respectively
The mean and variance of θ conditional on y1, y2          are 94.40 and 33.33␣
 ↪respectively
The mean and variance of θ conditional on y3, y4          are 100.00 and 100.00␣
 ↪respectively
The mean and variance of η conditional on y1, y2, y3, y4  are 73.38 and 33.33␣
 ↪respectively
The mean and variance of η conditional on y1, y2          are 100.00 and 100.00␣
 ↪respectively
The mean and variance of η conditional on y3, y4          are 73.38 and 33.33␣
 ↪respectively
```

Evidently, math tests provide no information about μ and language tests provide no information about η.

6.9 Univariate Time Series Analysis

We can use the multivariate normal distribution and a little matrix algebra to present foundations of univariate linear time series analysis.

Let x_t, y_t, v_t, w_{t+1} each be scalars for $t \geq 0$.

Consider the following model:

$$\begin{aligned} x_0 &\sim N\left(0, \sigma_0^2\right) \\ x_{t+1} &= ax_t + bw_{t+1}, \quad w_{t+1} \sim N\left(0,1\right), t \geq 0 \\ y_t &= cx_t + dv_t, \quad v_t \sim N\left(0,1\right), t \geq 0 \end{aligned}$$

We can compute the moments of x_t

1. $Ex_{t+1}^2 = a^2 Ex_t^2 + b^2, t \geq 0$, where $Ex_0^2 = \sigma_0^2$
2. $Ex_{t+j}x_t = a^j Ex_t^2, \forall t \; \forall j$

Given some T, we can formulate the sequence $\{x_t\}_{t=0}^T$ as a random vector

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_T \end{bmatrix}$$

and the covariance matrix Σ_x can be constructed using the moments we have computed above.

Similarly, we can define

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_T \end{bmatrix}, \quad v = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_T \end{bmatrix}$$

and therefore

$$Y = CX + DV$$

where C and D are both diagonal matrices with constant c and d as diagonal respectively.

Consequently, the covariance matrix of Y is

$$\Sigma_y = EYY' = C\Sigma_x C' + DD'$$

By stacking X and Y, we can write

$$Z = \begin{bmatrix} X \\ Y \end{bmatrix}$$

and

$$\Sigma_z = EZZ' = \begin{bmatrix} \Sigma_x & \Sigma_x C' \\ C\Sigma_x & \Sigma_y \end{bmatrix}$$

Thus, the stacked sequences $\{x_t\}_{t=0}^T$ and $\{y_t\}_{t=0}^T$ jointly follow the multivariate normal distribution $N\left(0, \Sigma_z\right)$.

```
# as an example, consider the case where T = 3
T = 3
```

```
# variance of the initial distribution x_0
σ0 = 1.

# parameters of the equation system
a = .9
b = 1.
c = 1.0
d = .05
```

```
# construct the covariance matrix of X
Σx = np.empty((T+1, T+1))

Σx[0, 0] = σ0 ** 2
for i in range(T):
    Σx[i, i+1:] = Σx[i, i] * a ** np.arange(1, T+1-i)
    Σx[i+1:, i] = Σx[i, i+1:]

    Σx[i+1, i+1] = a ** 2 * Σx[i, i] + b ** 2
```

```
Σx
```

```
array([[1.      , 0.9     , 0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541]])
```

```
# construct the covariance matrix of Y
C = np.eye(T+1) * c
D = np.eye(T+1) * d

Σy = C @ Σx @ C.T + D @ D.T
```

```
# construct the covariance matrix of Z
Σz = np.empty((2*(T+1), 2*(T+1)))

Σz[:T+1, :T+1] = Σx
Σz[:T+1, T+1:] = Σx @ C.T
Σz[T+1:, :T+1] = C @ Σx
Σz[T+1:, T+1:] = Σy
```

```
Σz
```

```
array([[1.      , 0.9     , 0.81    , 0.729   , 1.      , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.81    ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
        2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 2.997541],
       [1.      , 0.9     , 0.81    , 0.729   , 1.0025  , 0.9     ,
        0.81    , 0.729   ],
```

(continues on next page)

(continued from previous page)

```
        [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.8125  ,
         1.629   , 1.4661  ],
        [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
         2.4686  , 2.21949 ],
        [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
         2.21949 , 3.000041]])
```

```
# construct the mean vector of Z
μz = np.zeros(2*(T+1))
```

The following Python code lets us sample random vectors X and Y.

This is going to be very useful for doing the conditioning to be used in the fun exercises below.

```
z = np.random.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

6.9.1 Smoothing Example

This is an instance of a classic `smoothing` calculation whose purpose is to compute $EX \mid Y$.

An interpretation of this example is

- X is a random sequence of hidden Markov state variables x_t
- Y is a sequence of observed signals y_t bearing information about the hidden state

```
# construct a MultivariateNormal instance
multi_normal_ex1 = MultivariateNormal(μz, Σz)
x = z[:T+1]
y = z[T+1:]
```

```
# partition Z into X and Y
multi_normal_ex1.partition(T+1)
```

```
# compute the conditional mean and covariance matrix of X given Y=y

print("X = ", x)
print("Y = ", y)
print(" E [ X | Y] = ", )

multi_normal_ex1.cond_dist(0, y)
```

```
X =  [0.95443932 2.27517229 2.21527392 2.31479003]
Y =  [0.98713274 2.35304452 2.197711   2.29038948]
 E [ X | Y] =
```

```
(array([0.98794883, 2.3495815 , 2.19820231, 2.28961141]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861729e-08, 2.80236945e-11],
```

(continues on next page)

(continued from previous page)

```
        [5.57449314e-06, 2.48876343e-03, 5.57452116e-06, 1.25113944e-08],
        [1.24861728e-08, 5.57452116e-06, 2.48876346e-03, 5.58575339e-06],
        [2.80236945e-11, 1.25113941e-08, 5.58575339e-06, 2.49377812e-03]]))
```

6.9.2 Filtering Exercise

Compute $E\left[x_t \mid y_{t-1}, y_{t-2}, \dots, y_0\right]$.

To do so, we need to first construct the mean vector and the covariance matrix of the subvector $[x_t, y_0, \dots, y_{t-2}, y_{t-1}]$.

For example, let's say that we want the conditional distribution of x_3.

```
t = 3
```

```
# mean of the subvector
sub_μz = np.zeros(t+1)

# covariance matrix of the subvector
sub_Σz = np.empty((t+1, t+1))

sub_Σz[0, 0] = Σz[t, t] # x_t
sub_Σz[0, 1:] = Σz[t, T+1:T+t+1]
sub_Σz[1:, 0] = Σz[T+1:T+t+1, t]
sub_Σz[1:, 1:] = Σz[T+1:T+t+1, T+1:T+t+1]
```

```
sub_Σz
```

```
array([[2.997541, 0.729   , 1.4661  , 2.21949 ],
       [0.729   , 1.0025  , 0.9     , 0.81    ],
       [1.4661  , 0.9     , 1.8125  , 1.629   ],
       [2.21949 , 0.81    , 1.629   , 2.4686  ]])
```

```
multi_normal_ex2 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex2.partition(1)
```

```
sub_y = y[:t]

multi_normal_ex2.cond_dist(0, sub_y)
```

```
(array([1.97775341]), array([[1.00201996]]))
```

6.9.3 Prediction Exercise

Compute $E\left[y_t \mid y_{t-j}, \ldots, y_0\right]$.

As what we did in exercise 2, we will construct the mean vector and covariance matrix of the subvector $\left[y_t, y_0, \ldots, y_{t-j-1}, y_{t-j}\right]$.

For example, we take a case in which $t = 3$ and $j = 2$.

```
t = 3
j = 2
```

```
sub_μz = np.zeros(t-j+2)
sub_Σz = np.empty((t-j+2, t-j+2))

sub_Σz[0, 0] = Σz[T+t+1, T+t+1]
sub_Σz[0, 1:] = Σz[T+t+1, T+1:T+t-j+2]
sub_Σz[1:, 0] = Σz[T+1:T+t-j+2, T+t+1]
sub_Σz[1:, 1:] = Σz[T+1:T+t-j+2, T+1:T+t-j+2]
```

```
sub_Σz
```

```
array([[3.000041, 0.729   , 1.4661  ],
       [0.729   , 1.0025  , 0.9     ],
       [1.4661  , 0.9     , 1.8125  ]])
```

```
multi_normal_ex3 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex3.partition(1)
```

```
sub_y = y[:t-j+1]

multi_normal_ex3.cond_dist(0, sub_y)
```

```
(array([1.90300907]), array([[1.81413617]]))
```

6.9.4 Constructing a Wold Representation

Now we'll apply Cholesky decomposition to decompose $\Sigma_y = HH'$ and form

$$\epsilon = H^{-1}Y.$$

Then we can represent y_t as

$$y_t = h_{t,t}\epsilon_t + h_{t,t-1}\epsilon_{t-1} + \cdots + h_{t,0}\epsilon_0.$$

```
H = np.linalg.cholesky(Σy)

H
```

```
array([[1.00124922, 0.        , 0.        , 0.        ],
       [0.8988771 , 1.00225743, 0.        , 0.        ],
       [0.80898939, 0.89978675, 1.00225743, 0.        ],
       [0.72809046, 0.80980808, 0.89978676, 1.00225743]])
```

```
ε = np.linalg.inv(H) @ y

ε
```

```
array([0.98590114, 1.46353673, 0.08306895, 0.3119319 ])
```

```
y
```

```
array([0.98713274, 2.35304452, 2.197711  , 2.29038948])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

6.10 Stochastic Difference Equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{y-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and

$$\begin{bmatrix} y_{-1} \\ y_0 \end{bmatrix} \sim N\left(\mu_{\tilde{y}}, \Sigma_{\tilde{y}}\right)$$

It can be written as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b}$$

We can compute y by solving the system

$$y = A^{-1}\left(b + u\right)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1} E\left[\left(b - \mu_b + u\right)\left(b - \mu_b + u\right)'\right]\left(A^{-1}\right)' \\ &= A^{-1}\left(\Sigma_b + \Sigma_u\right)\left(A^{-1}\right)' \end{aligned}$$

where

$$\mu_b = \begin{bmatrix} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}$$

$$\Sigma_b = \begin{bmatrix} C\Sigma_{\tilde{y}}C' & 0_{N-2\times N-2} \\ 0_{N-2\times 2} & 0_{N-2\times N-2} \end{bmatrix}, \quad C = \begin{bmatrix} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{bmatrix}$$

$$\Sigma_u = \begin{bmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{bmatrix}$$

```
# set parameters
T = 80
T = 160
# coefficients of the second order difference equation
α0 = 10
α1 = 1.53
α2 = -.9

# variance of u
σu = 1.
σu = 10.

# distribution of y_{-1} and y_{0}
μy_tilde = np.array([1., 0.5])
Σy_tilde = np.array([[2., 1.], [1., 0.5]])
```

```
# construct A and A^{\prime}
A = np.zeros((T, T))

for i in range(T):
    A[i, i] = 1

    if i-1 >= 0:
        A[i, i-1] = -α1

    if i-2 >= 0:
        A[i, i-2] = -α2

A_inv = np.linalg.inv(A)
```

```
# compute the mean vectors of b and y
μb = np.full(T, α0)
μb[0] += α1 * μy_tilde[1] + α2 * μy_tilde[0]
μb[1] += α2 * μy_tilde[1]

μy = A_inv @ μb
```

```
# compute the covariance matrices of b and y
Σu = np.eye(T) * σu ** 2

Σb = np.zeros((T, T))

C = np.array([[α2, α1], [0, α2]])
Σb[:2, :2] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

6.11 Application to Stock Price Model

Let

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}$$

Form

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

we have

$$\mu_p = B\mu_y$$
$$\Sigma_p = B\Sigma_y B'$$

```
β = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = β ** np.arange(0, T-i)
```

Denote

$$z = \begin{bmatrix} y \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} I \\ B \end{bmatrix}}_{\equiv D} y$$

Thus, $\{y_t\}_{t=1}^T$ and $\{p_t\}_{t=1}^T$ jointly follow the multivariate normal distribution $N(\mu_z, \Sigma_z)$, where

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D'$$

```
D = np.vstack([np.eye(T), B])
```

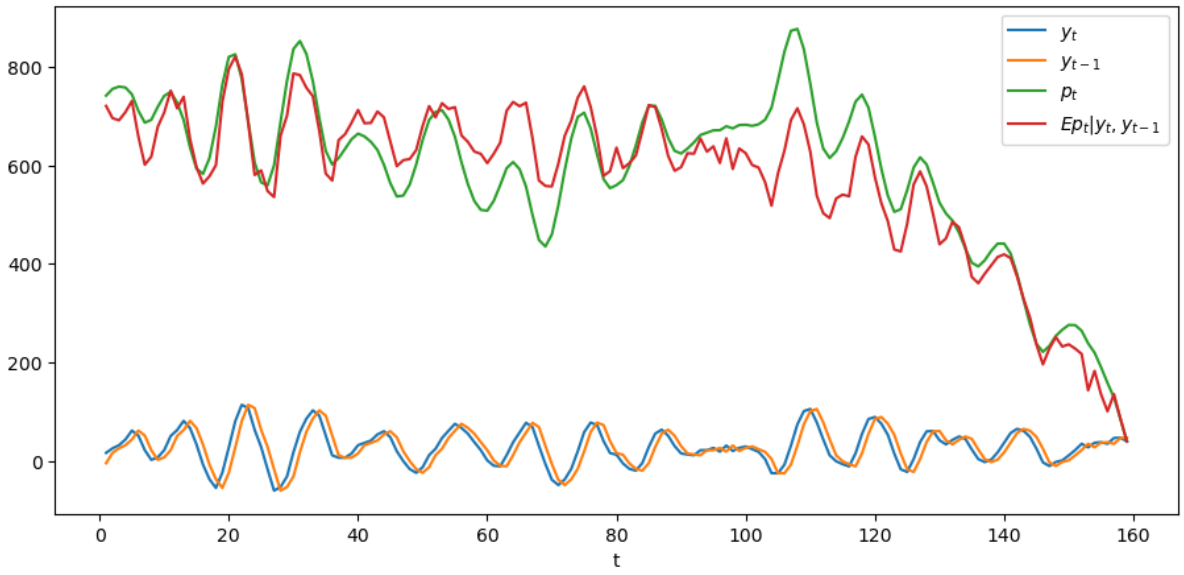
```
μz = D @ μy
Σz = D @ Σy @ D.T
```

We can simulate paths of y_t and p_t and compute the conditional mean $E[p_t \mid y_{t-1}, y_t]$ using the `MultivariateNormal` class.

```
z = np.random.multivariate_normal(μz, Σz)
y, p = z[:T], z[T:]
```

```
cond_Ep = np.empty(T-1)

sub_μ = np.empty(3)
sub_Σ = np.empty((3, 3))
for t in range(2, T+1):
    sub_μ[:] = μz[[t-2, t-1, T-1+t]]
    sub_Σ[:, :] = Σz[[t-2, t-1, T-1+t], :][:, [t-2, t-1, T-1+t]]

    multi_normal = MultivariateNormal(sub_μ, sub_Σ)
    multi_normal.partition(2)

    cond_Ep[t-2] = multi_normal.cond_dist(1, y[t-2:t])[0][0]
```

```
plt.plot(range(1, T), y[1:], label='$y_{t}$')
plt.plot(range(1, T), y[:-1], label='$y_{t-1}$')
plt.plot(range(1, T), p[1:], label='$p_{t}$')
plt.plot(range(1, T), cond_Ep, label='$Ep_{t}|y_{t}, y_{t-1}$')

plt.xlabel('t')
plt.legend(loc=1)
plt.show()
```



In the above graph, the green line is what the price of the stock would be if people had perfect foresight about the path of dividends while the green line is the conditional expectation $Ep_t|y_t, y_{t-1}$, which is what the price would be if people did not have perfect foresight but were optimally predicting future dividends on the basis of the information y_t, y_{t-1} at time t.

6.12 Filtering Foundations

Assume that x_0 is an $n \times 1$ random vector and that y_0 is a $p \times 1$ random vector determined by the *observation equation*

$$y_0 = Gx_0 + v_0, \quad x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0), \quad v_0 \sim \mathcal{N}(0, R)$$

where v_0 is orthogonal to x_0, G is a $p \times n$ matrix, and R is a $p \times p$ positive definite matrix.

We consider the problem of someone who

- *observes* y_0
- does not observe x_0,
- knows $\hat{x}_0, \Sigma_0, G, R$ and therefore the joint probability distribution of the vector $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$
- wants to infer x_0 from y_0 in light of what he knows about that joint probability distribution.

Therefore, the person wants to construct the probability distribution of x_0 conditional on the random vector y_0.

The joint distribution of $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ is multivariate normal $\mathcal{N}(\mu, \Sigma)$ with

$$\mu = \begin{bmatrix} \hat{x}_0 \\ G\hat{x}_0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_0 & \Sigma_0 G' \\ G\Sigma_0 & G\Sigma_0 G' + R \end{bmatrix}$$

By applying an appropriate instance of the above formulas for the mean vector $\hat{\mu}_1$ and covariance matrix $\hat{\Sigma}_{11}$ of z_1 conditional on z_2, we find that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ where

$$\begin{aligned} \beta_0 &= \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} \\ \tilde{x}_0 &= \hat{x}_0 + \beta_0(y_0 - G\hat{x}_0) \\ \tilde{\Sigma}_0 &= \Sigma_0 - \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 \end{aligned}$$

We can express our finding that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ by representing x_0 as

$$x_0 = \tilde{x}_0 + \zeta_0 \tag{6.2}$$

where ζ_0 is a Gaussian random vector that is orthogonal to $\tilde{x}_0$ and y_0 and that has mean vector 0 and conditional covariance matrix $E[\zeta_0 \zeta_0' | y_0] = \tilde{\Sigma}_0$.

6.12.1 Step toward dynamics

Now suppose that we are in a time series setting and that we have the one-step state transition equation

$$x_1 = Ax_0 + Cw_1, \quad w_1 \sim \mathcal{N}(0, I)$$

where A is an $n \times n$ matrix and C is an $n \times m$ matrix.

Using equation (6.2), we can also represent x_1 as

$$x_1 = A(\tilde{x}_0 + \zeta_0) + Cw_1$$

It follows that

$$Ex_1 | y_0 = A\tilde{x}_0$$

and that the corresponding conditional covariance matrix $E(x_1 - Ex_1|y_0)(x_1 - Ex_1|y_0)' \equiv \Sigma_1$ is

$$\Sigma_1 = A\tilde{\Sigma}_0 A' + CC'$$

or

$$\Sigma_1 = A\Sigma_0 A' - A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 A'$$

We can write the mean of x_1 conditional on y_0 as

$$\hat{x}_1 = A\hat{x}_0 + A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}(y_0 - G\hat{x}_0)$$

or

$$\hat{x}_1 = A\hat{x}_0 + K_0(y_0 - G\hat{x}_0)$$

where

$$K_0 = A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}$$

6.12.2 Dynamic version

Suppose now that for $t \geq 0$, $\{x_{t+1}, y_t\}_{t=0}^{\infty}$ are governed by the equations

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \end{aligned}$$

where as before $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$, w_{t+1} is the $t+1$th component of an i.i.d. stochastic process distributed as $w_{t+1} \sim \mathcal{N}(0, I)$, and v_t is the tth component of an i.i.d. process distributed as $v_t \sim \mathcal{N}(0, R)$ and the $\{w_{t+1}\}_{t=0}^{\infty}$ and $\{v_t\}_{t=0}^{\infty}$ processes are orthogonal at all pairs of dates.

The logic and formulas that we applied above imply that the probability distribution of x_t conditional on $y_0, y_1, \ldots, y_{t-1} = y^{t-1}$ is

$$x_t|y^{t-1} \sim \mathcal{N}(A\tilde{x}_t, A\tilde{\Sigma}_t A' + CC')$$

where $\{\tilde{x}_t, \tilde{\Sigma}_t\}_{t=1}^{\infty}$ can be computed by iterating on the following equations starting from $t = 1$ and initial conditions for $\tilde{x}_0, \tilde{\Sigma}_0$ computed as we have above:

$$\begin{aligned} \Sigma_t &= A\tilde{\Sigma}_{t-1} A' + CC' \\ \hat{x}_t &= A\tilde{x}_{t-1} \\ \beta_t &= \Sigma_t G'(G\Sigma_t G' + R)^{-1} \\ \tilde{x}_t &= \hat{x}_t + \beta_t(y_t - G\hat{x}_t) \\ \tilde{\Sigma}_t &= \Sigma_t - \Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t \end{aligned}$$

If we shift the first equation forward one period and then substitute the expression for $\tilde{\Sigma}_t$ on the right side of the fifth equation into it we obtain

$$\Sigma_{t+1} = CC' + A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A'.$$

This is a matrix Riccati difference equation that is closely related to another matrix Riccati difference equation that appears in a quantecon lecture on the basics of linear quadratic control theory.

That equation has the form

$$P_{t-1} = R + A'P_tA - A'P_tB(B'P_tB + Q)^{-1}B'P_tA.$$

Stare at the two preceding equations for a moment or two, the first being a matrix difference equation for a conditional covariance matrix, the second being a matrix difference equation in the matrix appearing in a quadratic form for an intertemporal cost of value function.

Although the two equations are not identical, they display striking family resemblences.

- the first equation tells dynamics that work **forward** in time
- the second equation tells dynamics that work **backward** in time
- while many of the terms are similar, one equation seems to apply matrix transformations to some matrices that play similar roles in the other equation

The family resemblences of these two equations reflects a transcendent **duality** that prevails between control theory and filtering theory.

6.12.3 An example

We can use the Python class *MultivariateNormal* to construct examples.

Here is an example for a single period problem at time 0

```
G = np.array([[1., 3.]])
R = np.array([[1.]])

x0_hat = np.array([0., 1.])
Σ0 = np.array([[1., .5], [.3, 2.]])

μ = np.hstack([x0_hat, G @ x0_hat])
Σ = np.block([[Σ0, Σ0 @ G.T], [G @ Σ0, G @ Σ0 @ G.T + R]])
```

```
# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
multi_normal.partition(2)
```

```
# the observation of y
y0 = 2.3

# conditional distribution of x0
μ1_hat, Σ11 = multi_normal.cond_dist(0, y0)
μ1_hat, Σ11
```

```
(array([-0.078125,  0.803125]),
 array([[ 0.72098214, -0.203125  ],
        [-0.403125  ,  0.228125  ]]))
```

```
A = np.array([[0.5, 0.2], [-0.1, 0.3]])
C = np.array([[2.], [1.]])
```

(continues on next page)

(continued from previous page)

```
# conditional distribution of x1
x1_cond = A @ μ1_hat
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
(array([0.1215625, 0.24875  ]),
 array([[4.12874554, 1.95523214],
        [1.92123214, 1.04592857]]))
```

6.12.4 Code for Iterating

Here is code for solving a dynamic filtering problem by iterating on our equations, followed by an example.

```
def iterate(x0_hat, Σ0, A, C, G, R, y_seq):

    p, n = G.shape

    T = len(y_seq)
    x_hat_seq = np.empty((T+1, n))
    Σ_hat_seq = np.empty((T+1, n, n))

    x_hat_seq[0] = x0_hat
    Σ_hat_seq[0] = Σ0

    for t in range(T):
        xt_hat = x_hat_seq[t]
        Σt = Σ_hat_seq[t]
        μ = np.hstack([xt_hat, G @ xt_hat])
        Σ = np.block([[Σt, Σt @ G.T], [G @ Σt, G @ Σt @ G.T + R]])

        # filtering
        multi_normal = MultivariateNormal(μ, Σ)
        multi_normal.partition(n)
        x_tilde, Σ_tilde = multi_normal.cond_dist(0, y_seq[t])

        # forecasting
        x_hat_seq[t+1] = A @ x_tilde
        Σ_hat_seq[t+1] = C @ C.T + A @ Σ_tilde @ A.T

    return x_hat_seq, Σ_hat_seq
```

```
iterate(x0_hat, Σ0, A, C, G, R, [2.3, 1.2, 3.2])
```

```
(array([[0.        , 1.        ],
        [0.1215625 , 0.24875   ],
        [0.18680212, 0.06904689],
        [0.75576875, 0.05558463]]),
 array([[[1.        , 0.5       ],
         [0.3       , 2.        ]],

        [[4.12874554, 1.95523214],
         [1.92123214, 1.04592857]],
```

(continues on next page)

(continued from previous page)

```
        [[4.08198663, 1.99218488],
         [1.98640488, 1.00886423]],

        [[4.06457628, 2.00041999],
         [1.99943739, 1.00275526]]]))
```

The iterative algorithm just described is a version of the celebrated **Kalman filter**.

We describe the Kalman filter and some applications of it in A First Look at the Kalman Filter

6.13 Classic Factor Analysis Model

The factor analysis model widely used in psychology and other fields can be represented as

$$Y = \Lambda f + U$$

where

1. Y is $n \times 1$ random vector, $EUU' = D$ is a diagonal matrix,
2. Λ is $n \times k$ coefficient matrix,
3. f is $k \times 1$ random vector, $Eff' = I$,
4. U is $n \times 1$ random vector, and $U \perp f$ (i.e., $EUf' = 0$)
5. It is presumed that k is small relative to n; often k is only 1 or 2, as in our IQ examples.

This implies that

$$\begin{aligned} \Sigma_y = EYY' &= \Lambda\Lambda' + D \\ EYf' &= \Lambda \\ EfY' &= \Lambda' \end{aligned}$$

Thus, the covariance matrix Σ_Y is the sum of a diagonal matrix D and a positive semi-definite matrix $\Lambda\Lambda'$ of rank k.

This means that all covariances among the n components of the Y vector are intermediated by their common dependencies on the $k <$ factors.

Form

$$Z = \begin{pmatrix} f \\ Y \end{pmatrix}$$

the covariance matrix of the expanded random vector Z can be computed as

$$\Sigma_z = EZZ' = \begin{pmatrix} I & \Lambda' \\ \Lambda & \Lambda\Lambda' + D \end{pmatrix}$$

In the following, we first construct the mean vector and the covariance matrix for the case where $N = 10$ and $k = 2$.

```
N = 10
k = 2
```

We set the coefficient matrix Λ and the covariance matrix of U to be

$$\Lambda = \begin{pmatrix} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{pmatrix}, \quad D = \begin{pmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{pmatrix}$$

where the first half of the first column of Λ is filled with 1s and 0s for the rest half, and symmetrically for the second column.

D is a diagonal matrix with parameter σ_u^2 on the diagonal.

```
Λ = np.zeros((N, k))
Λ[:N//2, 0] = 1
Λ[N//2:, 1] = 1

σu = .5
D = np.eye(N) * σu ** 2
```

```
# compute Σy
Σy = Λ @ Λ.T + D
```

We can now construct the mean vector and the covariance matrix for Z.

```
μz = np.zeros(k+N)

Σz = np.empty((k+N, k+N))

Σz[:k, :k] = np.eye(k)
Σz[:k, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
z = np.random.multivariate_normal(μz, Σz)

f = z[:k]
y = z[k:]
```

```
multi_normal_factor = MultivariateNormal(μz, Σz)
multi_normal_factor.partition(k)
```

Let's compute the conditional distribution of the hidden factor f on the observations Y, namely, $f \mid Y = y$.

```
multi_normal_factor.cond_dist(0, y)
```

```
(array([ 0.7823104 , -1.56360865]),
 array([[0.04761905, 0.        ],
        [0.        , 0.04761905]]))
```

We can verify that the conditional mean $E[f \mid Y = y] = BY$ where $B = \Lambda' \Sigma_y^{-1}$.

```
B = Λ.T @ np.linalg.inv(Σy)

B @ y
```

```
array([ 0.7823104 , -1.56360865])
```

Similarly, we can compute the conditional distribution $Y \mid f$.

```
multi_normal_factor.cond_dist(1, f)
```

```
(array([ 0.75102765,  0.75102765,  0.75102765,  0.75102765,  0.75102765,
        -1.76633527, -1.76633527, -1.76633527, -1.76633527, -1.76633527]),
 array([[0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25]]))
```

It can be verified that the mean is $\Lambda I^{-1} f = \Lambda f$.

```
Λ @ f
```

```
array([ 0.75102765,  0.75102765,  0.75102765,  0.75102765,  0.75102765,
       -1.76633527, -1.76633527, -1.76633527, -1.76633527, -1.76633527])
```

6.14 PCA and Factor Analysis

To learn about Principal Components Analysis (PCA), please see this lecture Singular Value Decompositions.

For fun, let's apply a PCA decomposition to a covariance matrix Σ_y that in fact is governed by our factor-analytic model.

Technically, this means that the PCA model is misspecified. (Can you explain why?)

Nevertheless, this exercise will let us study how well the first two principal components from a PCA can approximate the conditional expectations $Ef_i|Y$ for our two factors $f_i, i = 1, 2$ for the factor analytic model that we have assumed truly governs the data on Y we have generated.

So we compute the PCA decomposition

$$\Sigma_y = P\tilde{\Lambda}P'$$

where $\tilde{\Lambda}$ is a diagonal matrix.

We have

$$Y = P\epsilon$$

and

$$\epsilon = P'Y$$

Note that we will arrange the eigenvectors in P in the *descending* order of eigenvalues.

```
λ_tilde, P = np.linalg.eigh(Σy)

# arrange the eigenvectors by eigenvalues
ind = sorted(range(N), key=lambda x: λ_tilde[x], reverse=True)

P = P[:, ind]
λ_tilde = λ_tilde[ind]
Λ_tilde = np.diag(λ_tilde)

print('λ_tilde =', λ_tilde)
```

```
λ_tilde = [5.25 5.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
```

```
# verify the orthogonality of eigenvectors
np.abs(P @ P.T - np.eye(N)).max()
```

```
4.440892098500626e-16
```

```
# verify the eigenvalue decomposition is correct
P @ Λ_tilde @ P.T
```

```
array([[1.25, 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.25, 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.25, 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.  , 1.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.25, 1.  , 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.25, 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.25, 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.25, 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.  , 1.25]])
```

```
ε = P.T @ y

print("ε = ", ε)
```

```
ε =  [-3.67115199  1.8367642   0.25818451 -0.17288947 -0.37080011  0.10197771
  0.42231308 -0.16379674 -0.13284216  0.83610909]
```

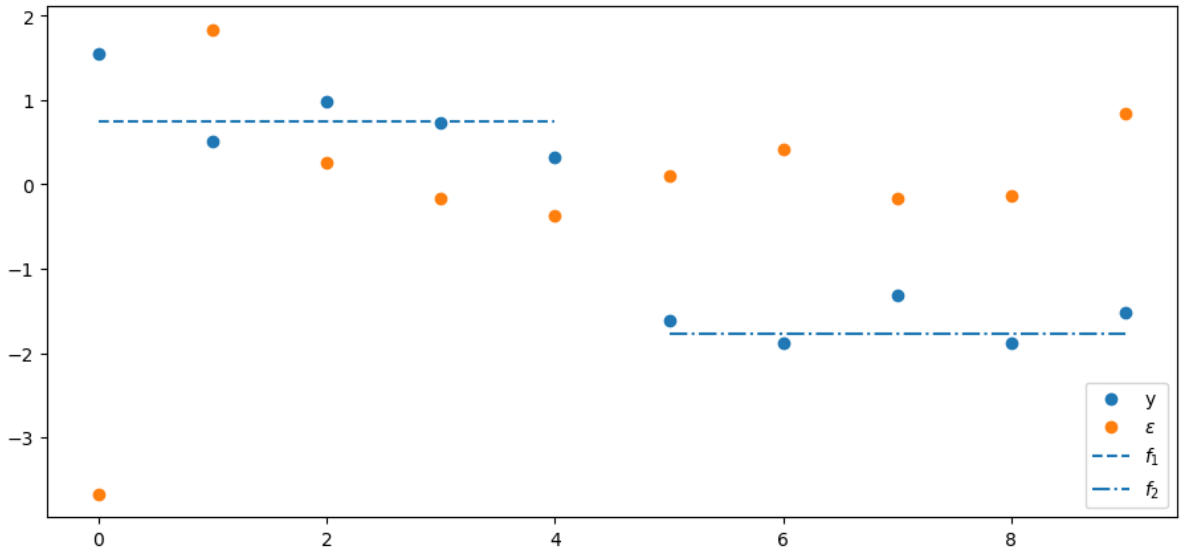
```
# print the values of the two factors

print('f = ', f)
```

```
f =  [ 0.75102765 -1.76633527]
```

Below we'll plot several things

- the N values of y
- the N values of the principal components ϵ
- the value of the first factor f_1 plotted only for the first $N/2$ observations of y for which it receives a non-zero loading in Λ
- the value of the second factor f_2 plotted only for the final $N/2$ observations for which it receives a non-zero loading in Λ

```
plt.scatter(range(N), y, label='y')
plt.scatter(range(N), ε, label='$\epsilon$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')
plt.legend()

plt.show()
```



Consequently, the first two ϵ_j correspond to the largest two eigenvalues.

Let's look at them, after which we'll look at $Ef|y = By$

```
ε[:2]
```

```
array([-3.67115199,  1.8367642 ])
```

```
# compare with Ef|y
B @ y
```

```
array([ 0.7823104 , -1.56360865])
```

The fraction of variance in y_t explained by the first two principal components can be computed as below.

```
□_tilde[:2].sum() / □_tilde.sum()
```

```
0.84
```

Compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

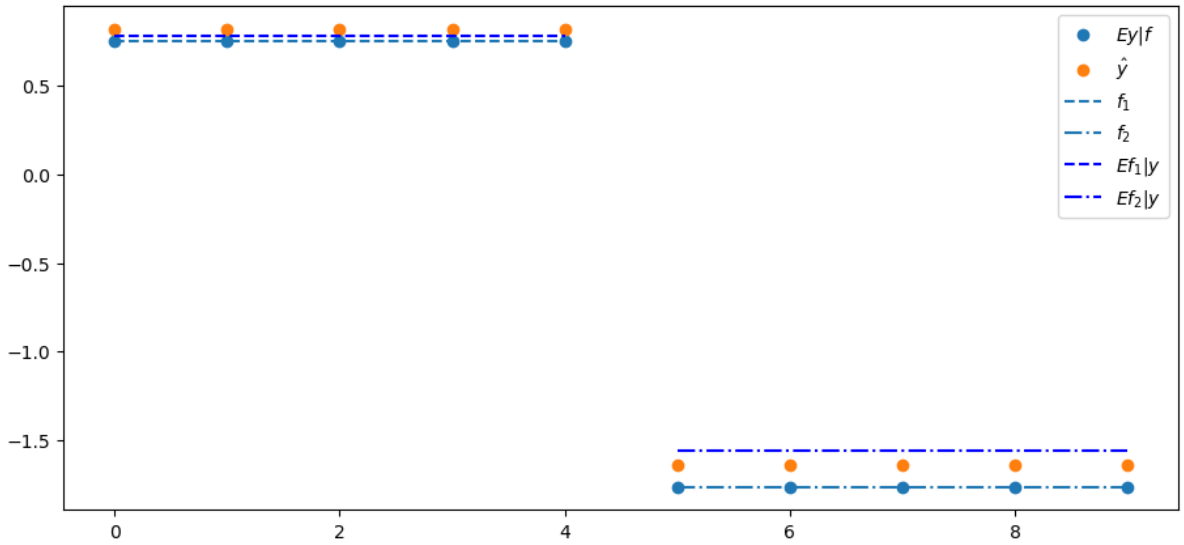
where P_j and P_k correspond to the largest two eigenvalues.

```
y_hat = P[:, :2] @ ε[:2]
```

In this example, it turns out that the projection $\hat{Y}$ of Y on the first two principal components does a good job of approximating $Ef \mid y$.

We confirm this in the following plot of f, $Ey \mid f$, $Ef \mid y$, and $\hat{y}$ on the coordinate axis versus y on the ordinate axis.

```
plt.scatter(range(N), Λ @ f, label='$Ey|f$')
plt.scatter(range(N), y_hat, label='$\hat{y}$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')

Efy = B @ y
plt.hlines(Efy[0], 0, N//2-1, ls='--', color='b', label='$Ef_{1}|y$')
plt.hlines(Efy[1], N//2, N-1, ls='-.', color='b', label='$Ef_{2}|y$')
plt.legend()

plt.show()
```



The covariance matrix of $\hat{Y}$ can be computed by first constructing the covariance matrix of ϵ and then use the upper left block for ϵ_1 and ϵ_2.

```
Σεjk = (P.T @ Σy @ P)[:2, :2]

Pjk = P[:, :2]

Σy_hat = Pjk @ Σεjk @ Pjk.T
print('Σy_hat = \n', Σy_hat)
```

```
Σy_hat =
 [[1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]]
```

Part II

Information & Bayesian Statistics

CHAPTER

SEVEN

TWO MEANINGS OF PROBABILITY

7.1 Overview

This lecture illustrates two distinct interpretations of a **probability distribution**

- A frequentist interpretation as **relative frequencies** anticipated to occur in a large i.i.d. sample
- A Bayesian interpretation as a **personal opinion** (about a parameter or list of parameters) after seeing a collection of observations

We recommend watching this video about **hypothesis testing** within the frequentist approach

https://youtu.be/8JIe_cz6qGA

After you watch that video, please watch the following video on the Bayesian approach to constructing **coverage intervals**

https://youtu.be/Pahyv9i_X2k

After you are familiar with the material in these videos, this lecture uses the Socratic method to to help consolidate your understanding of the different questions that are answered by

- a frequentist confidence interval
- a Bayesian coverage interval

We do this by inviting you to write some Python code.

It would be especially useful if you tried doing this after each question that we pose for you, before proceeding to read the rest of the lecture.

We provide our own answers as the lecture unfolds, but you'll learn more if you try writing your own code before reading and running ours.

Code for answering questions:

In addition to what's in Anaconda, this lecture will deploy the following library:

```
pip install prettytable
```

To answer our coding questions, we'll start with some imports

```
import numpy as np
import pandas as pd
import prettytable as pt
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as st
```

Empowered with these Python tools, we'll now explore the two meanings described above.

7.2 Frequentist Interpretation

Consider the following classic example.

The random variable X takes on possible values $k = 0, 1, 2, \ldots, n$ with probabilties

$$\text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

where the fixed parameter $\theta \in (0, 1)$.

This is called the **binomial distribution**.

Here

- θ is the probability that one toss of a coin will be a head, an outcome that we encode as $Y = 1$.
- $1 - \theta$ is the probability that one toss of the coin will be a tail, an outcome that we denote $Y = 0$.
- X is the total number of heads that came up after flipping the coin n times.

Consider the following experiment:

Take I **independent** sequences of n **independent** flips of the coin

Notice the repeated use of the adjective **independent**:

- we use it once to describe that we are drawing n independent times from a **Bernoulli** distribution with parameter θ to arrive at one draw from a **Binomial** distribution with parameters θ, n.
- we use it again to describe that we are then drawing I sequences of n coin draws.

Let $y_h^i \in \{0, 1\}$ be the realized value of Y on the hth flip during the ith sequence of flips.

Let $\sum_{h=1}^n y_h^i$ denote the total number of times heads come up during the ith sequence of n independent coin flips.

Let f_k record the fraction of samples of length n for which $\sum_{h=1}^n y_h^i = k$:

$$f_k^I = \frac{\text{number of samples of length n for which } \sum_{h=1}^n y_h^i = k}{I}$$

The probability $\text{Prob}(X = k|\theta)$ answers the following question:

- As I becomes large, in what fraction of I independent draws of n coin flips should we anticipate k heads to occur?

As usual, a law of large numbers justifies this answer.

Exercise 7.2.1

1. Please write a Python class to compute f_k^I
2. Please use your code to compute $f_k^I, k = 0, \ldots, n$ and compare them to $\text{Prob}(X = k|\theta)$ for various values of θ, n and I
3. With the Law of Large numbers in mind, use your code to say something

---

Solution to Exercise 7.2.1

Here is one solution:

```
class frequentist:

    def __init__(self, θ, n, I):

        '''
        initialization
        ------------------
        parameters:
        9 : probability that one toss of a coin will be a head with Y = 1
        n : number of independent flips in each independent sequence of draws
        I : number of independent sequence of draws

        '''

        self.θ, self.n, self.I = θ, n, I

    def binomial(self, k):

        '''compute the theoretical probability for specific input k'''

        θ, n = self.θ, self.n
        self.k = k
        self.P = binom.pmf(k, n, θ)

    def draw(self):

        '''draw n independent flips for I independent sequences'''

        θ, n, I = self.θ, self.n, self.I
        sample = np.random.rand(I, n)
        Y = (sample <= θ) * 1
        self.Y = Y

    def compute_fk(self, kk):

        '''compute f_{k}^I for specific input k'''

        Y, I = self.Y, self.I
        K = np.sum(Y, 1)
        f_kI = np.sum(K == kk) / I
        self.f_kI = f_kI
        self.kk = kk

    def compare(self):

        '''compute and print the comparison'''

        n = self.n
        comp = pt.PrettyTable()
        comp.field_names = ['k', 'Theoretical', 'Frequentist']
        self.draw()
        for i in range(n):
            self.binomial(i+1)
            self.compute_fk(i+1)
            comp.add_row([i+1, self.P, self.f_kI])
        print(comp)
```

```
θ, n, k, I = 0.7, 20, 10, 1_000_000

freq = frequentist(θ, n, I)

freq.compare()
```

```
+----+------------------------+-------------+
| k  |      Theoretical       | Frequentist |
+----+------------------------+-------------+
| 1  | 1.6271660538000033e-09 |     0.0     |
| 2  | 3.606884752589999e-08  |     0.0     |
| 3  |  5.04963865362601e-07  |     0.0     |
| 4  | 5.007558331512455e-06  |    7e-06    |
| 5  | 3.7389768875293014e-05 |   3.9e-05   |
| 6  | 0.00021810698510587546 |   0.000207  |
| 7  |  0.001017832597160754  |   0.000996  |
| 8  |  0.003859281930901185  |   0.003853  |
| 9  |  0.012006654896137007  |   0.012091  |
| 10 |  0.030817080900085007  |   0.030878  |
| 11 |  0.06536956554563476   |   0.065151  |
| 12 |  0.11439673970486108   |   0.114262  |
| 13 |   0.1642619852172365   |   0.164178  |
| 14 |  0.19163898275344252   |   0.191031  |
| 15 |  0.17886305056987967   |   0.179718  |
| 16 |   0.1304209743738704   |   0.130461  |
| 17 |  0.07160367220526209   |   0.071618  |
| 18 |  0.027845872524268643  |   0.027971  |
| 19 |  0.006839337111223895  |   0.006736  |
| 20 | 0.0007979226629761189  |   0.000803  |
+----+------------------------+-------------+
```

From the table above, can you see the law of large numbers at work?

Let's do some more calculations.

Comparison with different θ

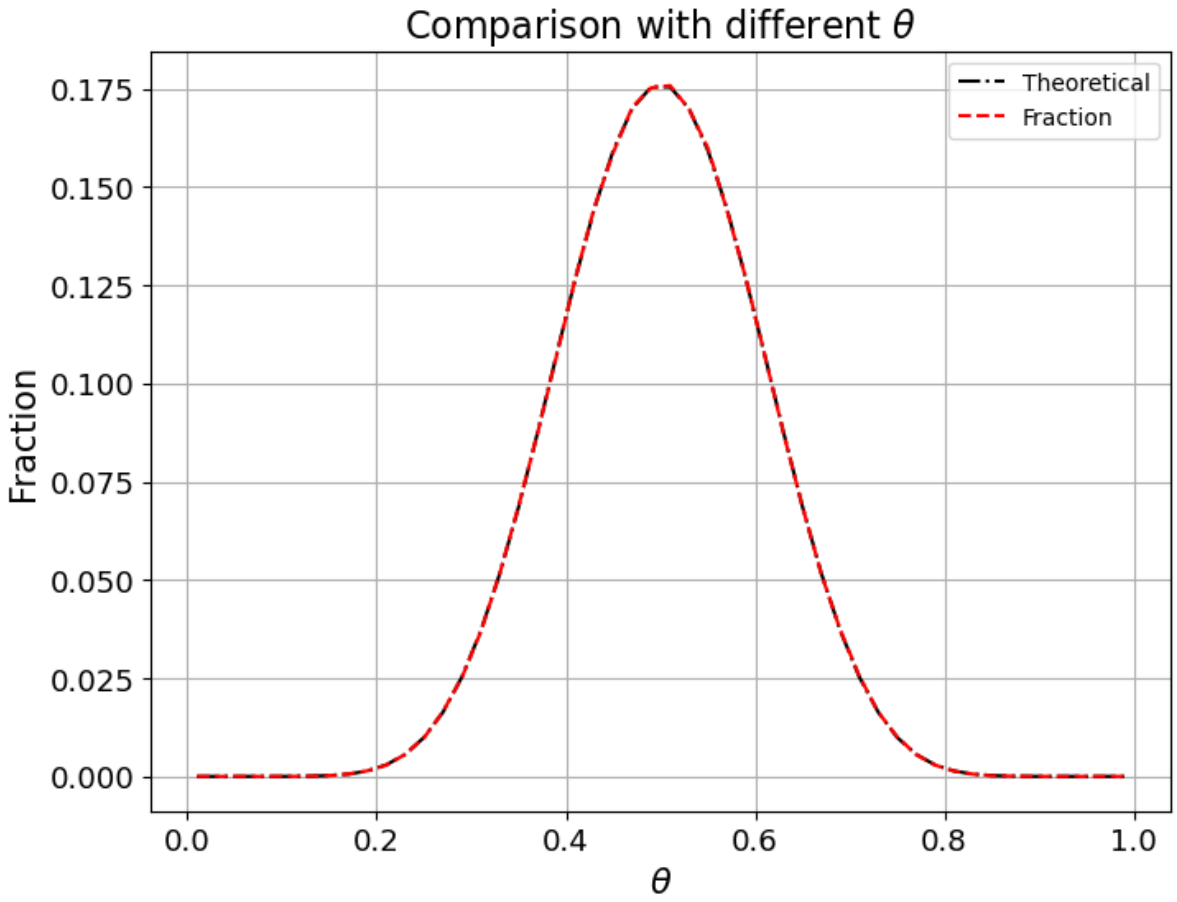
Now we fix

$$n = 20, k = 10, I = 1,000,000$$

We'll vary θ from 0.01 to 0.99 and plot outcomes against θ.

```
θ_low, θ_high, npt = 0.01, 0.99, 50
thetas = np.linspace(θ_low, θ_high, npt)
P = []
f_kI = []
for i in range(npt):
    freq = frequentist(thetas[i], n, I)
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(thetas, P, 'k-.', label='Theoretical')
ax.plot(thetas, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $\theta$', fontsize=16)
plt.xlabel(r'$\theta$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



Comparison with different n

Now we fix $\theta = 0.7, k = 10, I = 1,000,000$ and vary n from 1 to 100.

Then we'll plot outcomes.

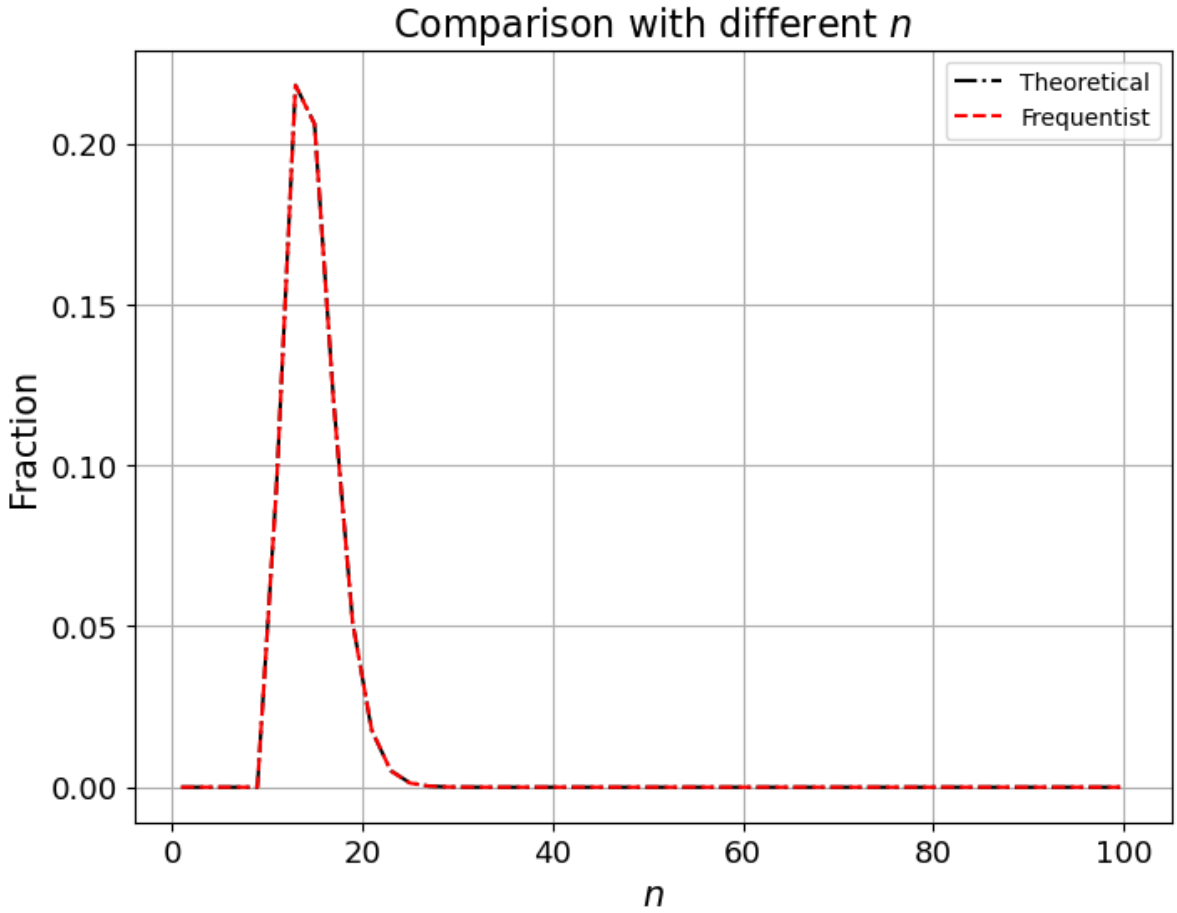
```
n_low, n_high, nn = 1, 100, 50
ns = np.linspace(n_low, n_high, nn, dtype='int')
P = []
f_kI = []
for i in range(nn):
    freq = frequentist(θ, ns[i], I)
    freq.binomial(k)
    freq.draw()
```

(continues on next page)

(continued from previous page)

```
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(ns, P, 'k-.', label='Theoretical')
ax.plot(ns, f_kI, 'r--', label='Frequentist')
plt.title(r'Comparison with different $n$', fontsize=16)
plt.xlabel(r'$n$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



Comparison with different I

Now we fix $\theta = 0.7, n = 20, k = 10$ and vary $\log(I)$ from 2 to 7.

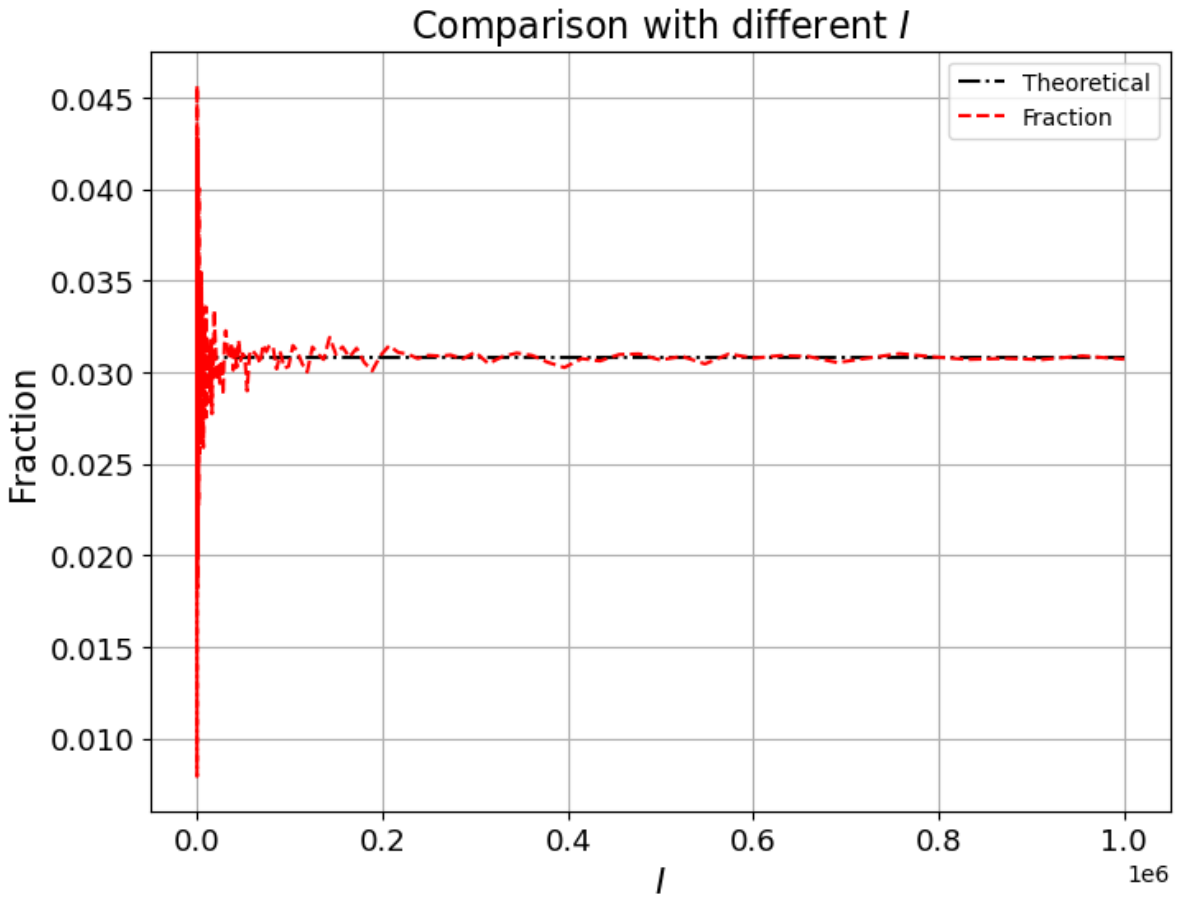
```
I_log_low, I_log_high, nI = 2, 6, 200
log_Is = np.linspace(I_log_low, I_log_high, nI)
Is = np.power(10, log_Is).astype(int)
P = []
```

(continues on next page)

(continued from previous page)

```
f_kI = []
for i in range(nI):
    freq = frequentist(θ, n, Is[i])
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(Is, P, 'k-.', label='Theoretical')
ax.plot(Is, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $I$', fontsize=16)
plt.xlabel(r'$I$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



From the above graphs, we can see that I**, the number of independent sequences,** plays an important role.

When I becomes larger, the difference between theoretical probability and frequentist estimate becomes smaller.

Also, as long as I is large enough, changing θ or n does not substantially change the accuracy of the observed fraction as

an approximation of θ.

The Law of Large Numbers is at work here.

For each draw of an independent sequence, Prob$(X_i = k|\theta)$ is the same, so aggregating all draws forms an i.i.d sequence of a binary random variable $\rho_{k,i}, i = 1, 2, ...I$, with a mean of Prob$(X = k|\theta)$ and a variance of

$$n \cdot \text{Prob}(X = k|\theta) \cdot (1 - \text{Prob}(X = k|\theta)).$$

So, by the LLN, the average of $P_{k,i}$ converges to:

$$E[\rho_{k,i}] = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

as I goes to infinity.

7.3 Bayesian Interpretation

We again use a binomial distribution.

But now we don't regard θ as being a fixed number.

Instead, we think of it as a **random variable**.

θ is described by a probability distribution.

But now this probability distribution means something different than a relative frequency that we can anticipate to occur in a large i.i.d. sample.

Instead, the probability distribution of θ is now a summary of our views about likely values of θ either

- **before** we have seen **any** data at all, or
- **before** we have seen **more** data, after we have seen **some** data

Thus, suppose that, before seeing any data, you have a personal prior probability distribution saying that

$$P(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

where $B(\alpha, \beta)$ is a **beta function** , so that $P(\theta)$ is a **beta distribution** with parameters α, β.

Exercise 7.3.1

a) Please write down the **likelihood function** for a sample of length n from a binomial distribution with parameter θ.

b) Please write down the **posterior** distribution for θ after observing one flip of the coin.

c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

d) Please plot the posterior distribution for θ as a function of θ as n grows as $1, 2, ...$

e) For various n's, please describe and compute a Bayesian coverage interval for the interval $[.45, .55]$.

f) Please tell what question a Bayesian coverage interval answers.

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Solution to Exercise 7.3.1

a) Please write down the **likelihood function** and the **posterior** distribution for θ after observing one flip of our coin.

Suppose the outcome is $\mathbf{Y}$.

The likelihood function is:

$$L(Y|\theta) = \text{Prob}(X = Y|\theta) = \theta^Y (1-\theta)^{1-Y}$$

b) Please write the **posterior** distribution for θ after observing one flip of our coin.

The prior distribution is

$$\text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

We can derive the posterior distribution for θ via

$$\begin{aligned}
\text{Prob}(\theta|Y) &= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\text{Prob}(Y)} \\
&= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(Y|\theta)\text{Prob}(\theta)d\theta} \\
&= \frac{\theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta} \\
&= \frac{\theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}}{\int_0^1 \theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}d\theta}
\end{aligned}$$

which means that

$$\text{Prob}(\theta|Y) \sim \text{Beta}(\alpha + Y, \beta + (1-Y))$$

Now please pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior with $\beta = \alpha = .5$.

c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

```
class Bayesian:

    def __init__(self, θ=0.4, n=1_000_000, α=0.5, β=0.5):
        """
        Parameters:
        -----------
        θ : float, ranging from [0,1].
           probability that one toss of a coin will be a head with Y = 1

        n : int.
           number of independent flips in an independent sequence of draws

        α&β : int or float.
```

(continues on next page)

(continued from previous page)

```
            parameters of the prior distribution on θ

        """
        self.θ, self.n, self.α, self.β = θ, n, α, β
        self.prior = st.beta(α, β)

    def draw(self):
        """
        simulate a single sequence of draws of length n, given probability θ

        """
        array = np.random.rand(self.n)
        self.draws = (array < self.θ).astype(int)

    def form_single_posterior(self, step_num):
        """
        form a posterior distribution after observing the first step_num elements of␣
↪the draws

        Parameters
        ----------
        step_num: int.
               number of steps observed to form a posterior distribution

        Returns
        ------
        the posterior distribution for sake of plotting in the subsequent steps

        """
        heads_num = self.draws[:step_num].sum()
        tails_num = step_num - heads_num

        return st.beta(self.α+heads_num, self.β+tails_num)

    def form_posterior_series(self,num_obs_list):
        """
        form a series of posterior distributions that form after observing different␣
↪number of draws.

        Parameters
        ----------
        num_obs_list: a list of int.
               a list of the number of observations used to form a series of␣
↪posterior distributions.

        """
        self.posterior_list = []
        for num in num_obs_list:
            self.posterior_list.append(self.form_single_posterior(num))
```

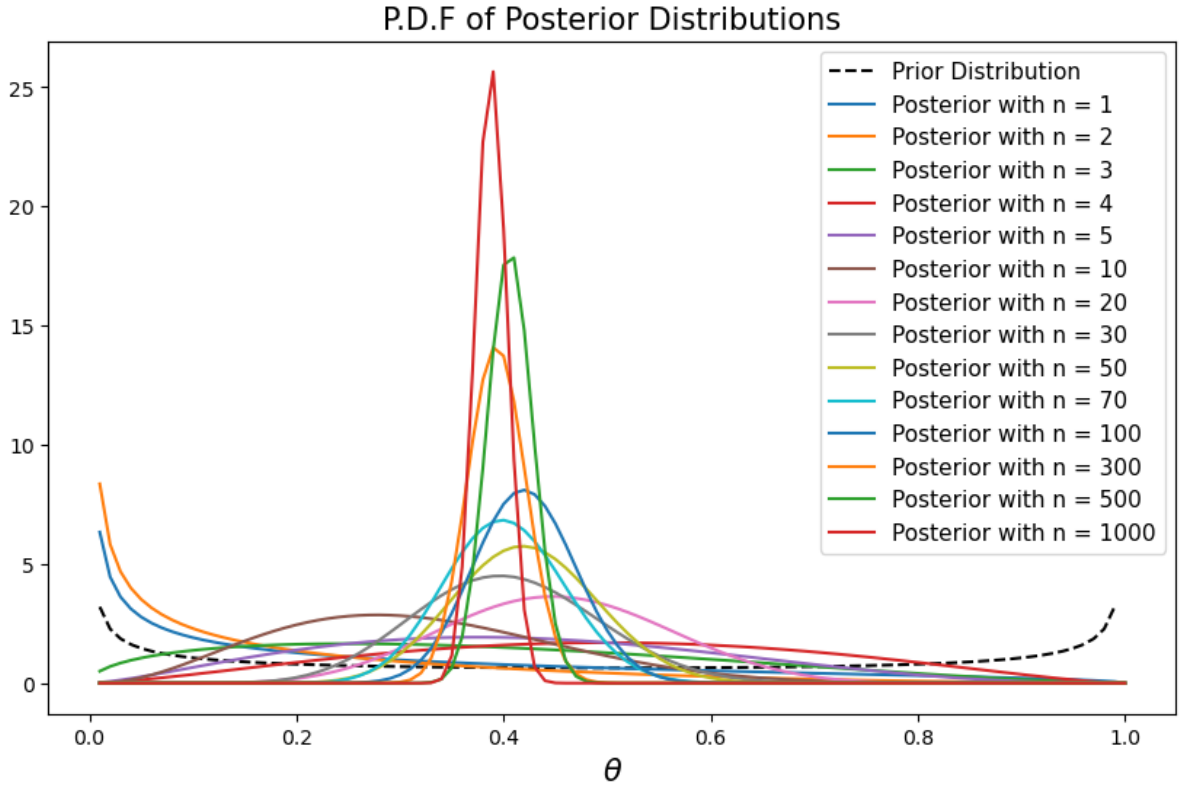
d) Please plot the posterior distribution for θ as a function of θ as n grows from $1, 2, ...$

```
Bay_stat = Bayesian()
Bay_stat.draw()

num_list = [1, 2, 3, 4, 5, 10, 20, 30, 50, 70, 100, 300, 500, 1000, # this line for␣
↪finite n
```

(continues on next page)

(continued from previous page)

```
            5000, 10_000, 50_000, 100_000, 200_000, 300_000]  # this line for␣
↪approximately infinite n

Bay_stat.form_posterior_series(num_list)

θ_values = np.linspace(0.01, 1, 100)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(θ_values, Bay_stat.prior.pdf(θ_values), label='Prior Distribution', color='k',
↪ linestyle='--')

for ii, num in enumerate(num_list[:14]):
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values), label='Posterior␣
↪with n = %d' % num)

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



e) For various n's, please describe and compute .05 and .95 quantiles for posterior probabilities.

```
upper_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list[:14]]
lower_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list[:14]]

interval_df = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
interval_df['upper'] = upper_bound
interval_df['lower'] = lower_bound
interval_df.index = num_list[:14]
interval_df = interval_df.T
interval_df
```

```
            1         2         3         4         5         10        20    \
upper  0.001543  0.000868  0.062413  0.16528  0.127776  0.117329  0.280091
lower  0.771480  0.569259  0.764466  0.83472  0.739366  0.558127  0.629953

            30        50        70        100       300       500       1000
upper  0.264015  0.310582  0.307914  0.341239  0.347839  0.370342  0.362915
lower  0.549653  0.536089  0.498084  0.502138  0.440341  0.442460  0.413563
```

As n increases, we can see that Bayesian coverage intervals narrow and move toward 0.4.

f) Please tell what question a Bayesian coverage interval answers.

The Bayesian coverage interval tells the range of θ that corresponds to the $[p_1, p_2]$ quantiles of the cumulative probability distribution (CDF) of the posterior distribution.

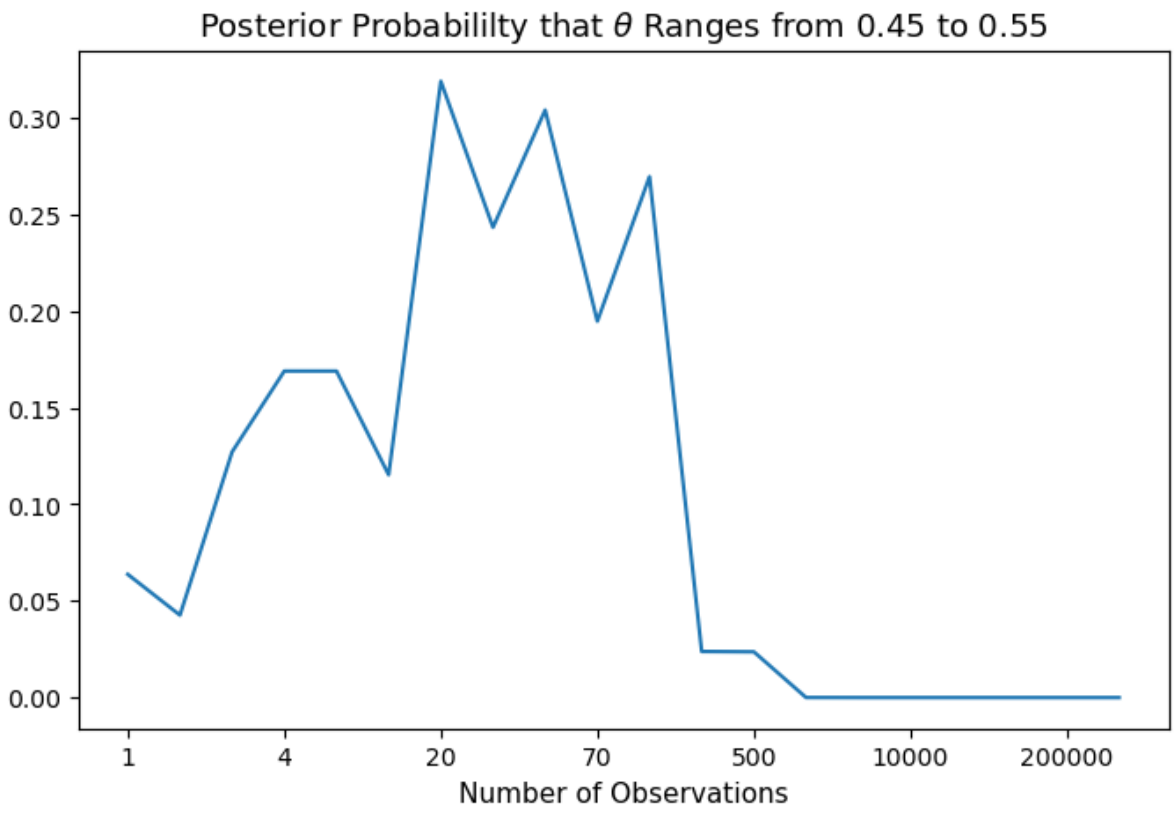
To construct the coverage interval we first compute a posterior distribution of the unknown parameter θ.

If the CDF is $F(\theta)$, then the Bayesian coverage interval $[a, b]$ for the interval $[p_1, p_2]$ is described by

$$F(a) = p_1, F(b) = p_2$$

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

```
left_value, right_value = 0.45, 0.55

posterior_prob_list=[ii.cdf(right_value)-ii.cdf(left_value) for ii in Bay_stat.
 ↪posterior_list]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(posterior_prob_list)
ax.set_title('Posterior Probabililty that '+ r"$\theta$" +' Ranges from %.2f to %.2f'
 ↪%(left_value, right_value),
             fontsize=13)
ax.set_xticks(np.arange(0, len(posterior_prob_list), 3))
ax.set_xticklabels(num_list[::3])
ax.set_xlabel('Number of Observations', fontsize=11)

plt.show()
```



Notice that in the graph above the posterior probabililty that $\theta \in [.45, .55]$ typically exhibits a hump shape as n increases.

Two opposing forces are at work.

The first force is that the individual adjusts his belief as he observes new outcomes, so his posterior probability distribution becomes more and more realistic, which explains the rise of the posterior probabililty.

However, $[.45, .55]$ actually excludes the true $\theta = .4$ that generates the data.

As a result, the posterior probabililty drops as larger and larger samples refine his posterior probability distribution of θ.

The descent seems precipitous only because of the scale of the graph that has the number of observations increasing disproportionately.

When the number of observations becomes large enough, our Bayesian becomes so confident about θ that he considers $\theta \in [.45, .55]$ very unlikely.

That is why we see a nearly horizontal line when the number of observations exceeds 500.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

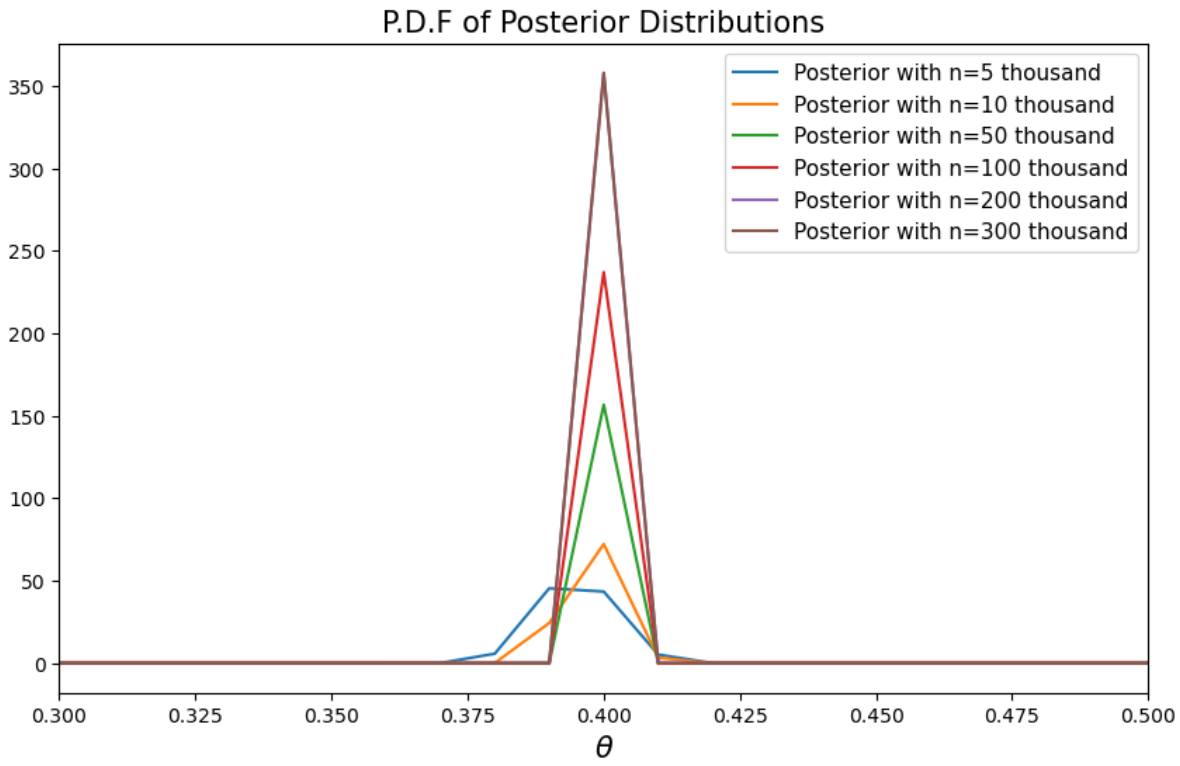
Using the Python class we made above, we can see the evolution of posterior distributions as n approaches infinity.

```
fig, ax = plt.subplots(figsize=(10, 6))

for ii, num in enumerate(num_list[14:]):
    ii += 14
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values),
            label='Posterior with n=%d thousand' % (num/1000))
```

(continues on next page)

(continued from previous page)

```
ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)
ax.set_xlim(0.3, 0.5)

ax.legend(fontsize=11)
plt.show()
```



As n increases, we can see that the probability density functions *concentrate* on 0.4, the true value of θ.

Here the posterior means converges to 0.4 while the posterior standard deviations converges to 0 from above.

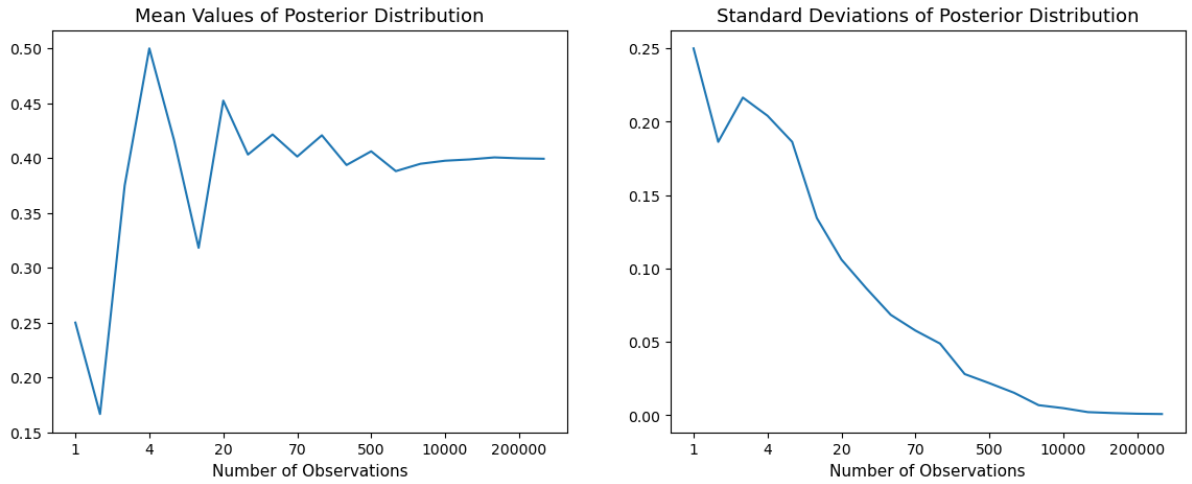
To show this, we compute the means and variances statistics of the posterior distributions.

```
mean_list = [ii.mean() for ii in Bay_stat.posterior_list]
std_list = [ii.std() for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

ax[0].plot(mean_list)
ax[0].set_title('Mean Values of Posterior Distribution', fontsize=13)
ax[0].set_xticks(np.arange(0, len(mean_list), 3))
ax[0].set_xticklabels(num_list[::3])
ax[0].set_xlabel('Number of Observations', fontsize=11)

ax[1].plot(std_list)
ax[1].set_title('Standard Deviations of Posterior Distribution', fontsize=13)
ax[1].set_xticks(np.arange(0, len(std_list), 3))
ax[1].set_xticklabels(num_list[::3])
ax[1].set_xlabel('Number of Observations', fontsize=11)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



How shall we interpret the patterns above?

The answer is encoded in the Bayesian updating formulas.

It is natural to extend the one-step Bayesian update to an n-step Bayesian update.

$$\text{Prob}(\theta|k) = \frac{\text{Prob}(\theta, k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta) * \text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta) * \text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta) * \text{Prob}(\theta) d\theta}$$

$$= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} d\theta}$$

$$= \frac{(1-\theta)^{\beta+N-k-1} * \theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1} * \theta^{\alpha+k-1} d\theta}$$

$$= Beta(\alpha + k, \beta + N - k)$$

A beta distribution with α and β has the following mean and variance.

The mean is $\frac{\alpha}{\alpha+\beta}$

The variance is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

- α can be viewed as the number of successes
- β can be viewed as the number of failures

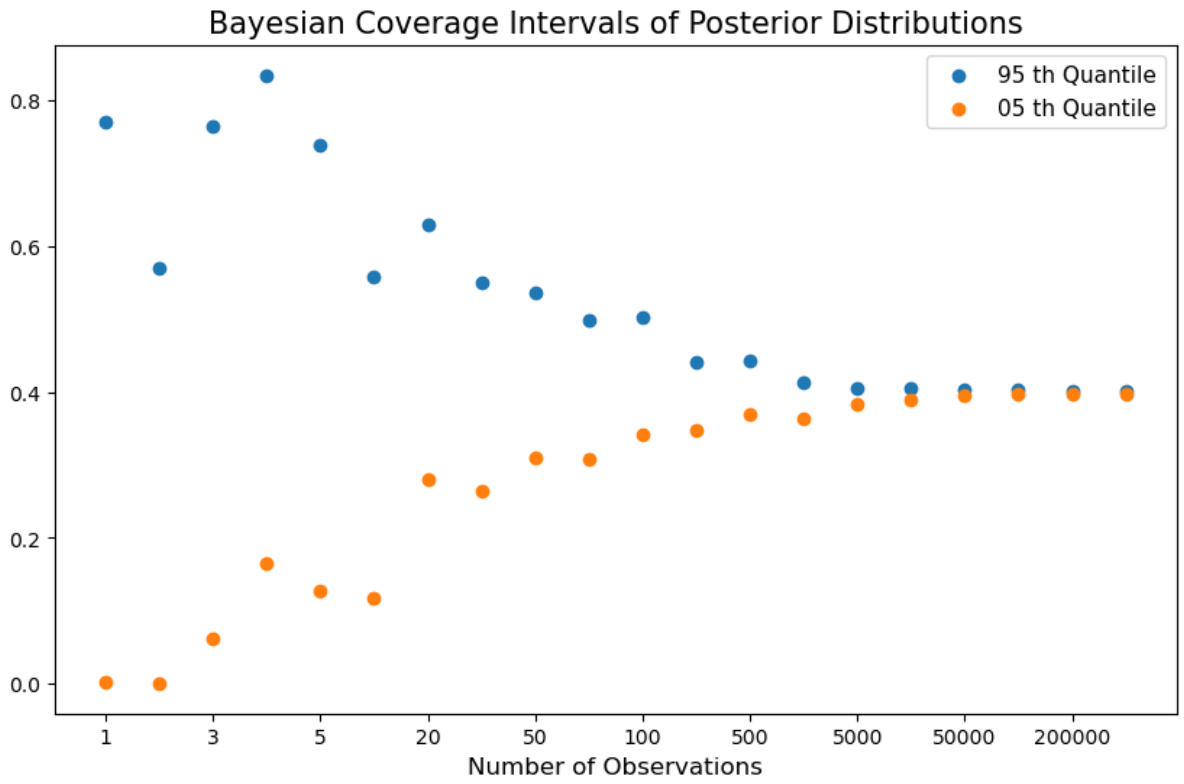
The random variables k and $N - k$ are governed by Binomial Distribution with $\theta = 0.4$.

Call this the true data generating process.

According to the Law of Large Numbers, for a large number of observations, observed frequencies of k and $N - k$ will be described by the true data generating process, i.e., the population probability distribution that we assumed when generating the observations on the computer. (See Exercise 7.2.1).

Consequently, the mean of the posterior distribution converges to 0.4 and the variance withers to zero.

```
upper_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list]
lower_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(figsize=(10, 6))
ax.scatter(np.arange(len(upper_bound)), upper_bound, label='95 th Quantile')
ax.scatter(np.arange(len(lower_bound)), lower_bound, label='05 th Quantile')

ax.set_xticks(np.arange(0, len(upper_bound), 2))
ax.set_xticklabels(num_list[::2])
ax.set_xlabel('Number of Observations', fontsize=12)
ax.set_title('Bayesian Coverage Intervals of Posterior Distributions', fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



After observing a large number of outcomes, the posterior distribution collapses around 0.4.

Thus, the Bayesian statististian comes to believe that θ is near $.4$.

As shown in the figure above, as the number of observations grows, the Bayesian coverage intervals (BCIs) become narrower and narrower around 0.4.

However, if you take a closer look, you will find that the centers of the BCIs are not exactly 0.4, due to the persistent influence of the prior distribution and the randomness of the simulation path.

7.4 Role of a Conjugate Prior

We have made assumptions that link functional forms of our likelihood function and our prior in a way that has eased our calculations considerably.

In particular, our assumptions that the likelihood function is **binomial** and that the prior distribution is a **beta distribution** have the consequence that the posterior distribution implied by Bayes' Law is also a **beta distribution**.

So posterior and prior are both beta distributions, albeit ones with different parameters.

When a likelihood function and prior fit together like hand and glove in this way, we can say that the prior and posterior are **conjugate distributions**.

In this situation, we also sometimes say that we have **conjugate prior** for the likelihood function $\text{Prob}(X|\theta)$.

Typically, the functional form of the likelihood function determines the functional form of a **conjugate prior**.

A natural question to ask is why should a person's personal prior about a parameter θ be restricted to be described by a conjugate prior?

Why not some other functional form that more sincerely describes the person's beliefs.

To be argumentative, one could ask, why should the form of the likelihood function have *anything* to say about my personal beliefs about θ?

A dignified response to that question is, well, it shouldn't, but if you want to compute a posterior easily you'll just be happier if your prior is conjugate to your likelihood.

Otherwise, your posterior won't have a convenient analytical form and you'll be in the situation of wanting to apply the Markov chain Monte Carlo techniques deployed in *this quantecon lecture*.

We also apply these powerful methods to approximating Bayesian posteriors for non-conjugate priors in *this quantecon lecture* and *this quantecon lecture*

CHAPTER

EIGHT

NON-CONJUGATE PRIORS

This lecture is a sequel to the *quantecon lecture*.

That lecture offers a Bayesian interpretation of probability in a setting in which the likelihood function and the prior distribution over parameters just happened to form a **conjugate** pair in which

- application of Bayes' Law produces a posterior distribution that has the same functional form as the prior

Having a likelihood and prior that are conjugate can simplify calculation of a posterior, faciltating analytical or nearly analytical calculations.

But in many situations the likelihood and prior need not form a conjugate pair.

- after all, a person's prior is his or her own business and would take a form conjugate to a likelihood only by remote coincidence

In these situations, computing a posterior can become very challenging.

In this lecture, we illustrate how modern Bayesians confront non-conjugate priors by using Monte Carlo techniques that involve

- first cleverly forming a Markov chain whose invariant distribution is the posterior distribution we want
- simulating the Markov chain until it has converged and then sampling from the invariant distribution to approximate the posterior

We shall illustrate the approach by deploying two powerful Python modules that implement this approach as well as another closely related one to be described below.

The two Python modules are

- `numpyro`
- `pymc4`

As usual, we begin by importing some Python code.

```
# install dependencies
!pip install numpyro pyro-ppl torch jax
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as st
import torch

# jax
```

(continues on next page)

(continued from previous page)

```
import jax.numpy as jnp
from jax import lax, random

# pyro
import pyro
from pyro import distributions as dist
import pyro.distributions.constraints as constraints
from pyro.infer import MCMC, NUTS, SVI, ELBO, Trace_ELBO
from pyro.optim import Adam

# numpyro
import numpyro
from numpyro import distributions as ndist
import numpyro.distributions.constraints as nconstraints
from numpyro.infer import MCMC as nMCMC
from numpyro.infer import NUTS as nNUTS
from numpyro.infer import SVI as nSVI
from numpyro.infer import ELBO as nELBO
from numpyro.infer import Trace_ELBO as nTrace_ELBO
from numpyro.optim import Adam as nAdam
```

8.1 Unleashing MCMC on a Binomial Likelihood

This lecture begins with the binomial example in the *quantecon lecture*.

That lecture computed a posterior

- analytically via choosing the conjugate priors,

This lecture instead computes posteriors

- numerically by sampling from the posterior distribution through MCMC methods, and
- using a variational inference (VI) approximation.

We use both the packages `pyro` and `numpyro` with assistance from `jax` to approximate a posterior distribution

We use several alternative prior distributions

We compare computed posteriors with ones associated with a conjugate prior as described in *the quantecon lecture*

8.1.1 Analytical Posterior

Assume that the random variable $X \sim Binom\left(n, \theta\right)$.

This defines a likelihood function

$$L\left(Y|\theta\right) = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

where $Y = k$ is an observed data point.

We view θ as a random variable for which we assign a prior distribution having density $f(\theta)$.

We will try alternative priors later, but for now, suppose the prior is distributed as $\theta \sim Beta\left(\alpha, \beta\right)$, i.e.,

$$f(\theta) = \text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

We choose this as our prior for now because we know that a conjugate prior for the binomial likelihood function is a beta distribution.

After observing k successes among N sample observations, the posterior probability distributionof θ is

$$\text{Prob}(\theta|k) = \frac{\text{Prob}(\theta, k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta)\text{Prob}(\theta)d\theta}$$

$$= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta}$$

$$= \frac{(1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1}d\theta}.$$

Thus,

$$\text{Prob}(\theta|k) \sim Beta(\alpha + k, \beta + N - k)$$

The analytical posterior for a given conjugate beta prior is coded in the following Python code.

```
def simulate_draw(theta, n):
    """
    Draws a Bernoulli sample of size n with probability P(Y=1) = theta
    """
    rand_draw = np.random.rand(n)
    draw = (rand_draw < theta).astype(int)
    return draw


def analytical_beta_posterior(data, alpha0, beta0):
    """
    Computes analytically the posterior distribution with beta prior parametrized by␣
 ↪(alpha, beta)
    given # num observations

    Parameters
    ---------
    num : int.
        the number of observations after which we calculate the posterior
    alpha0, beta0 : float.
        the parameters for the beta distribution as a prior

    Returns
    ---------
    The posterior beta distribution
    """
    num = len(data)
    up_num = data.sum()
    down_num = num - up_num
    return st.beta(alpha0 + up_num, beta0 + down_num)
```

8.1.2 Two Ways to Approximate Posteriors

Suppose that we don't have a conjugate prior.

Then we can't compute posteriors analytically.

Instead, we use computational tools to approximate the posterior distribution for a set of alternative prior distributions using both `Pyro` and `Numpyro` packages in Python.

We first use the **Markov Chain Monte Carlo** (MCMC) algorithm .

We implement the NUTS sampler to sample from the posterior.

In that way we construct a sampling distribution that approximates the posterior.

After doing that we deply another procedure called **Variational Inference** (VI).

In particular, we implement Stochastic Variational Inference (SVI) machinery in both `Pyro` and `Numpyro`.

The MCMC algorithm supposedly generates a more accurate approximation since in principle it directly samples from the posterior distribution.

But it can be computationally expensive, especially when dimension is large.

A VI approach can be cheaper, but it is likely to produce an inferior approximation to the posterior, for the simple reason that it requires guessing a parametric **guide functional form** that we use to approximate a posterior.

This guide function is likely at best to be an imperfect approximation.

By paying the cost of restricting the putative posterior to have a restricted functional form, the problem of approximating a posteriors is transformed to a well-posed optimization problem that seeks parameters of the putative posterior that minimize a Kullback-Leibler (KL) divergence between true posterior and the putatitive posterior distribution.

- minimizing the KL divergence is equivalent with maximizing a criterion called the **Evidence Lower Bound** (ELBO), as we shall verify soon.

8.2 Prior Distributions

In order to be able to apply MCMC sampling or VI, `Pyro` and `Numpyro` require that a prior distribution satisfy special properties:

- we must be able sample from it;
- we must be able to compute the log pdf pointwise;
- the pdf must be differentiable with respect to the parameters.

We'll want to define a distribution `class`.

We will use the following priors:

- a uniform distribution on $[\underline{\theta}, \overline{\theta}]$, where $0 \leq \underline{\theta} < \overline{\theta} \leq 1$.
- a truncated log-normal distribution with support on $[0, 1]$ with parameters (μ, σ).
  - To implement this, let $Z \sim Normal(\mu, \sigma)$ and $\tilde{Z}$ be truncated normal with support $[\log(0), \log(1)]$, then $\exp(Z)$ has a log normal distribution with bounded support $[0, 1]$. This can be easily coded since `Numpyro` has a built-in truncated normal distribution, and `Torch` provides a `TransformedDistribution` class that includes an exponential transformation.
  - Alternatively, we can use a rejection sampling strategy by assigning the probability rate to 0 outside the bounds and rescaling accepted samples, i.e., realizations that are within the bounds, by the total probability computed

via CDF of the original distribution. This can be implemented by defining a truncated distribution class with `pyro`'s `dist.Rejector` class.

  - We implement both methods in the below section and verify that they produce the same result.

- a shifted von Mises distribution that has support confined to $[0, 1]$ with parameter (μ, κ).
  - Let $X \sim vonMises(0, \kappa)$. We know that X has bounded support $[-\pi, \pi]$. We can define a shifted von Mises random variable $\tilde{X} = a + bX$ where $a = 0.5, b = 1/(2\pi)$ so that $\tilde{X}$ is supported on $[0, 1]$.
  - This can be implemented using `Torch`'s `TransformedDistribution` class with its `AffineTransform` method.
  - If instead, we want the prior to be von-Mises distributed with center $\mu = 0.5$, we can choose a high concentration level κ so that most mass is located between 0 and 1. Then we can truncate the distribution using the above strategy. This can be implemented using `pyro`'s `dist.Rejector` class. We choose $\kappa > 40$ in this case.
- a truncated Laplace distribution.
  - We also considered a truncated Laplace distribution because its density comes in a piece-wise non-smooth form and has a distinctive spiked shape.
  - The truncated Laplace can be created using `Numpyro`'s `TruncatedDistribution` class.

```
# used by Numpyro
def TruncatedLogNormal_trans(loc, scale):
    """
    Obtains the truncated log normal distribution using numpyro's TruncatedNormal and␣
 ↪ExpTransform
    """
    base_dist = ndist.TruncatedNormal(low=jnp.log(0), high=jnp.log(1), loc=loc,␣
 ↪scale=scale)
    return ndist.TransformedDistribution(
        base_dist,ndist.transforms.ExpTransform()
        )

def ShiftedVonMises(kappa):
    """
    Obtains the shifted von Mises distribution using AffineTransform
    """
    base_dist = ndist.VonMises(0, kappa)
    return ndist.TransformedDistribution(
        base_dist, ndist.transforms.AffineTransform(loc=0.5, scale=1/(2*jnp.pi))
        )

def TruncatedLaplace(loc, scale):
    """
    Obtains the truncated Laplace distribution on [0,1]
    """
    base_dist = ndist.Laplace(loc, scale)
    return ndist.TruncatedDistribution(
        base_dist, low=0.0, high=1.0
    )

# used by Pyro
class TruncatedLogNormal(dist.Rejector):
    """
    Define a TruncatedLogNormal distribution through rejection sampling in Pyro
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, loc, scale_0, upp=1):
        self.upp = upp
        propose = dist.LogNormal(loc, scale_0)

        def log_prob_accept(x):
            return (x < upp).type_as(x).log()

        log_scale = dist.LogNormal(loc, scale_0).cdf(torch.as_tensor(upp)).log()
        super(TruncatedLogNormal, self).__init__(propose, log_prob_accept, log_scale)

    @constraints.dependent_property
    def support(self):
        return constraints.interval(0, self.upp)


class TruncatedvonMises(dist.Rejector):
    """
    Define a TruncatedvonMises distribution through rejection sampling in Pyro
    """
    def __init__(self, kappa, mu=0.5, low=0.0, upp=1.0):
        self.low, self.upp = low, upp
        propose = dist.VonMises(mu, kappa)

        def log_prob_accept(x):
            return ((x > low) & (x < upp)).type_as(x).log()

        log_scale = torch.log(
            torch.tensor(
                st.vonmises(kappa=kappa, loc=mu).cdf(upp)
                - st.vonmises(kappa=kappa, loc=mu).cdf(low))
        )
        super(TruncatedvonMises, self).__init__(propose, log_prob_accept, log_scale)

    @constraints.dependent_property
    def support(self):
        return constraints.interval(self.low, self.upp)
```

8.2.1 Variational Inference

Instead of directly sampling from the posterior, the **variational inference** methodw approximates an unknown posterior distribution with a family of tractable distributions/densities.

It then seeks to minimizes a measure of statistical discrepancy between the approximating and true posteriors.

Thus variational inference (VI) approximates a posterior by solving a minimization problem.

Let the latent parameter/variable that we want to infer be θ.

Let the prior be $p(\theta)$ and the likelihood be $p\left(Y|\theta\right)$.

We want $p\left(\theta|Y\right)$.

Bayes' rule implies

$$p\left(\theta|Y\right) = \frac{p\left(Y,\theta\right)}{p\left(Y\right)} = \frac{p\left(Y|\theta\right)p\left(\theta\right)}{p\left(Y\right)}$$

where

$$p\left(Y\right)=\int d\theta p\left(Y\mid\theta\right)p\left(Y\right). \tag{8.1}$$

The integral on the right side of (8.1) is typically difficult to compute.

Consider a **guide distribution** $q_\phi(\theta)$ parameterized by ϕ that we'll use to approximate the posterior.

We choose parameters ϕ of the guide distribution to minimize a Kullback-Leibler (KL) divergence between the approximate posterior $q_\phi(\theta)$ and the posterior:

$$D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))\equiv-\int d\theta q(\theta;\phi)\log\frac{p(\theta\mid Y)}{q(\theta;\phi)}$$

Thus, we want a **variational distribution** q that solves

$$\min_{\phi}\quad D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))$$

Note that

$$\begin{aligned}
D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y)) &= -\int d\theta q(\theta;\phi)\log\frac{P(\theta\mid Y)}{q(\theta;\phi)}\\
&= -\int d\theta q(\theta)\log\frac{\frac{p(\theta,Y)}{p(Y)}}{q(\theta)}\\
&= -\int d\theta q(\theta)\log\frac{p(\theta,Y)}{p(\theta)q(Y)}\\
&= -\int d\theta q(\theta)\left[\log\frac{p(\theta,Y)}{q(\theta)}-\log p(Y)\right]\\
&= -\int d\theta q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}+\int d\theta q(\theta)\log p(Y)\\
&= -\int d\theta q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}+\log p(Y)\\
\log p(Y) &= D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))+\int d\theta q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}
\end{aligned}$$

For observed data Y, $p(\theta,Y)$ is a constant, so minimizing KL divergence is equivalent to maximizing

$$ELBO\equiv\int d\theta q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}=\mathbb{E}_{q_\phi(\theta)}\left[\log p(\theta,Y)-\log q_\phi(\theta)\right] \tag{8.2}$$

Formula (8.2) is called the evidence lower bound (ELBO).

A standard optimization routine can used to search for the optimal ϕ in our parametrized distribution $q_\phi(\theta)$.

The parameterized distribution $q_\phi(\theta)$ is called the **variational distribution**.

We can implement Stochastic Variational Inference (SVI) in Pyro and Numpyro using the `Adam` gradient descent algorithm to approximate posterior.

We use two sets of variational distributions: Beta and TruncatedNormal with support $[0,1]$

- Learnable parameters for the Beta distribution are (alpha, beta), both of which are positive.
- Learnable parameters for the Truncated Normal distribution are (loc, scale).

We restrict the truncated Normal paramter 'loc' to be in the interval $[0,1]$.

8.3 Implementation

We have constructed a Python class `BaysianInference` that requires the following arguments to be initialized:

- `param`: a tuple/scalar of parameters dependent on distribution types
- `name_dist`: a string that specifies distribution names

The (`param`, `name_dist`) pair includes:

- ('beta', alpha, beta)
- ('uniform', upper_bound, lower_bound)
- ('lognormal', loc, scale)
  - Note: This is the truncated log normal.
- ('vonMises', kappa), where kappa denotes concentration parameter, and center location is set to 0.5.
  - Note: When using `Pyro`, this is the truncated version of the original vonMises distribution;
  - Note: When using `Numpyro`, this is the **shifted** distribution.
- ('laplace', loc, scale)
  - Note: This is the truncated Laplace

The class `BaysianInference` has several key methods :

- `sample_prior`:
  - This can be used to draw a single sample from the given prior distribution.
- `show_prior`:
  - Plots the approximate prior distribution by repeatedly drawing samples and fitting a kernal density curve.
- `MCMC_sampling`:
  - INPUT: (data, num_samples, num_warmup=1000)
  - Take a `np.array` data and generate MCMC sampling of posterior of size `num_samples`.
- `SVI_run`:
  - INPUT: (data, guide_dist, n_steps=10000)
  - guide_dist = 'normal' - use a **truncated** normal distribution as the parametrized guide
  - guide_dist = 'beta' - use a beta distribution as the parametrized guide
  - RETURN: (params, losses) - the learned parameters in a `dict` and the vector of loss at each step.

```
class BayesianInference:
    def __init__(self, param, name_dist, solver):
        """
        Parameters
        ----------
        param : tuple.
            a tuple object that contains all relevant parameters for the distribution
        dist : str.
            name of the distribution - 'beta', 'uniform', 'lognormal', 'vonMises',
↪'tent'
        solver : str.
            either pyro or numpyro
```

(continues on next page)

(continued from previous page)

```
        """
        self.param = param
        self.name_dist = name_dist
        self.solver = solver

        # jax requires explicit PRNG state to be passed
        self.rng_key = random.PRNGKey(0)


    def sample_prior(self):
        """
        Define the prior distribution to sample from in Pyro/Numpyro models.
        """
        if self.name_dist=='beta':
            # unpack parameters
            alpha0, beta0 = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', dist.Beta(alpha0, beta0))
            else:
                sample = numpyro.sample('theta', ndist.Beta(alpha0, beta0), rng_
↪key=self.rng_key)

        elif self.name_dist=='uniform':
            # unpack parameters
            lb, ub = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', dist.Uniform(lb, ub))
            else:
                sample = numpyro.sample('theta', ndist.Uniform(lb, ub), rng_key=self.
↪rng_key)

        elif self.name_dist=='lognormal':
            # unpack parameters
            loc, scale = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', TruncatedLogNormal(loc, scale))
            else:
                sample = numpyro.sample('theta', TruncatedLogNormal_trans(loc, scale),
↪ rng_key=self.rng_key)

        elif self.name_dist=='vonMises':
            # unpack parameters
            kappa = self.param
            if self.solver=='pyro':
                sample = pyro.sample('theta', TruncatedvonMises(kappa))
            else:
                sample = numpyro.sample('theta', ShiftedVonMises(kappa), rng_key=self.
↪rng_key)

        elif self.name_dist=='laplace':
            # unpack parameters
            loc, scale = self.param
            if self.solver=='pyro':
                print("WARNING: Please use Numpyro for truncated Laplace.")
                sample = None
            else:
```

(continues on next page)

(continued from previous page)

```
                sample = numpyro.sample('theta', TruncatedLaplace(loc, scale), rng_
↪key=self.rng_key)

        return sample


    def show_prior(self, size=1e5, bins=20, disp_plot=1):
        """
        Visualizes prior distribution by sampling from prior and plots the␣
↪approximated sampling distribution
        """
        self.bins = bins

        if self.solver=='pyro':
            with pyro.plate('show_prior', size=size):
                sample = self.sample_prior()
            # to numpy
            sample_array = sample.numpy()

        elif self.solver=='numpyro':
            with numpyro.plate('show_prior', size=size):
                sample = self.sample_prior()
            # to numpy
            sample_array=jnp.asarray(sample)

        # plot histogram and kernel density
        if disp_plot==1:
            sns.displot(sample_array, kde=True, stat='density', bins=bins, height=5,␣
↪aspect=1.5)
            plt.xlim(0, 1)
            plt.show()
        else:
            return sample_array


    def model(self, data):
        """
        Define the probabilistic model by specifying prior, conditional likelihood,␣
↪and data conditioning
        """
        if not torch.is_tensor(data):
            data = torch.tensor(data)
        # set prior
        theta = self.sample_prior()

        # sample from conditional likelihood
        if self.solver=='pyro':
            output = pyro.sample('obs', dist.Binomial(len(data), theta), obs=torch.
↪sum(data))
        else:
            # Note: numpyro.sample() requires obs=np.ndarray
            output = numpyro.sample('obs', ndist.Binomial(len(data), theta),␣
↪obs=torch.sum(data).numpy())
        return output
```

(continues on next page)

(continued from previous page)

```
    def MCMC_sampling(self, data, num_samples, num_warmup=1000):
        """
        Computes numerically the posterior distribution with beta prior parametrized␣
↪by (alpha0, beta0)
        given data using MCMC
        """
        # tensorize
        data = torch.tensor(data)

        # use pyro
        if self.solver=='pyro':

            nuts_kernel = NUTS(self.model)
            mcmc = MCMC(nuts_kernel, num_samples=num_samples, warmup_steps=num_warmup,
↪ disable_progbar=True)
            mcmc.run(data)

        # use numpyro
        elif self.solver=='numpyro':

            nuts_kernel = nNUTS(self.model)
            mcmc = nMCMC(nuts_kernel, num_samples=num_samples, num_warmup=num_warmup,␣
↪progress_bar=False)
            mcmc.run(self.rng_key, data=data)

        # collect samples
        samples = mcmc.get_samples()['theta']
        return samples


    def beta_guide(self, data):
        """
        Defines the candidate parametrized variational distribution that we train to␣
↪approximate posterior with Pyro/Numpyro
        Here we use parameterized beta
        """
        if self.solver=='pyro':
            alpha_q = pyro.param('alpha_q', torch.tensor(0.5),
                                 constraint=constraints.positive)
            beta_q = pyro.param('beta_q', torch.tensor(0.5),
                                 constraint=constraints.positive)
            pyro.sample('theta', dist.Beta(alpha_q, beta_q))

        else:
            alpha_q = numpyro.param('alpha_q', 10,
                                 constraint=nconstraints.positive)
            beta_q = numpyro.param('beta_q', 10,
                                 constraint=nconstraints.positive)

            numpyro.sample('theta', ndist.Beta(alpha_q, beta_q))


    def truncnormal_guide(self, data):
        """
        Defines the candidate parametrized variational distribution that we train to␣
↪approximate posterior with Pyro/Numpyro
```

(continues on next page)

(continued from previous page)

```
        Here we use truncated normal on [0,1]
        """
        loc = numpyro.param('loc', 0.5,
                            constraint=nconstraints.interval(0.0, 1.0))
        scale = numpyro.param('scale', 1,
                            constraint=nconstraints.positive)
        numpyro.sample('theta', ndist.TruncatedNormal(loc, scale, low=0.0, high=1.0))


    def SVI_init(self, guide_dist, lr=0.0005):
        """
        Initiate SVI training mode with Adam optimizer
        NOTE: truncnormal_guide can only be used with numpyro solver
        """
        adam_params = {"lr": lr}

        if guide_dist=='beta':
            if self.solver=='pyro':
                optimizer = Adam(adam_params)
                svi = SVI(self.model, self.beta_guide, optimizer, loss=Trace_ELBO())

            elif self.solver=='numpyro':
                optimizer = nAdam(step_size=lr)
                svi = nSVI(self.model, self.beta_guide, optimizer, loss=nTrace_ELBO())

        elif guide_dist=='normal':
            # only allow numpyro
            if self.solver=='pyro':
                print("WARNING: Please use Numpyro with TruncatedNormal guide")
                svi = None

            elif self.solver=='numpyro':
                optimizer = nAdam(step_size=lr)
                svi = nSVI(self.model, self.truncnormal_guide, optimizer, loss=nTrace_
↪ELBO())
        else:
            print("WARNING: Please input either 'beta' or 'normal'")
            svi = None

        return svi

    def SVI_run(self, data, guide_dist, n_steps=10000):
        """
        Runs SVI and returns optimized parameters and losses

        Returns
        --------
        params : the learned parameters for guide
        losses : a vector of loss at each step
        """
        # tensorize data
        if not torch.is_tensor(data):
            data = torch.tensor(data)

        # initiate SVI
        svi = self.SVI_init(guide_dist=guide_dist)
```

(continues on next page)

(continued from previous page)

```
        # do gradient steps
        if self.solver=='pyro':
            # store loss vector
            losses = np.zeros(n_steps)
            for step in range(n_steps):
                losses[step] = svi.step(data)

            # pyro only supports beta VI distribution
            params = {
                'alpha_q': pyro.param('alpha_q').item(),
                'beta_q': pyro.param('beta_q').item()
                }

        elif self.solver=='numpyro':
            result = svi.run(self.rng_key, n_steps, data, progress_bar=False)
            params = dict(
                (key, np.asarray(value)) for key, value in result.params.items()
                )
            losses = np.asarray(result.losses)

        return params, losses
```

8.4 Alternative Prior Distributions

Let's see how well our sampling algorithm does in approximating

- a log normal distribution
- a uniform distribution

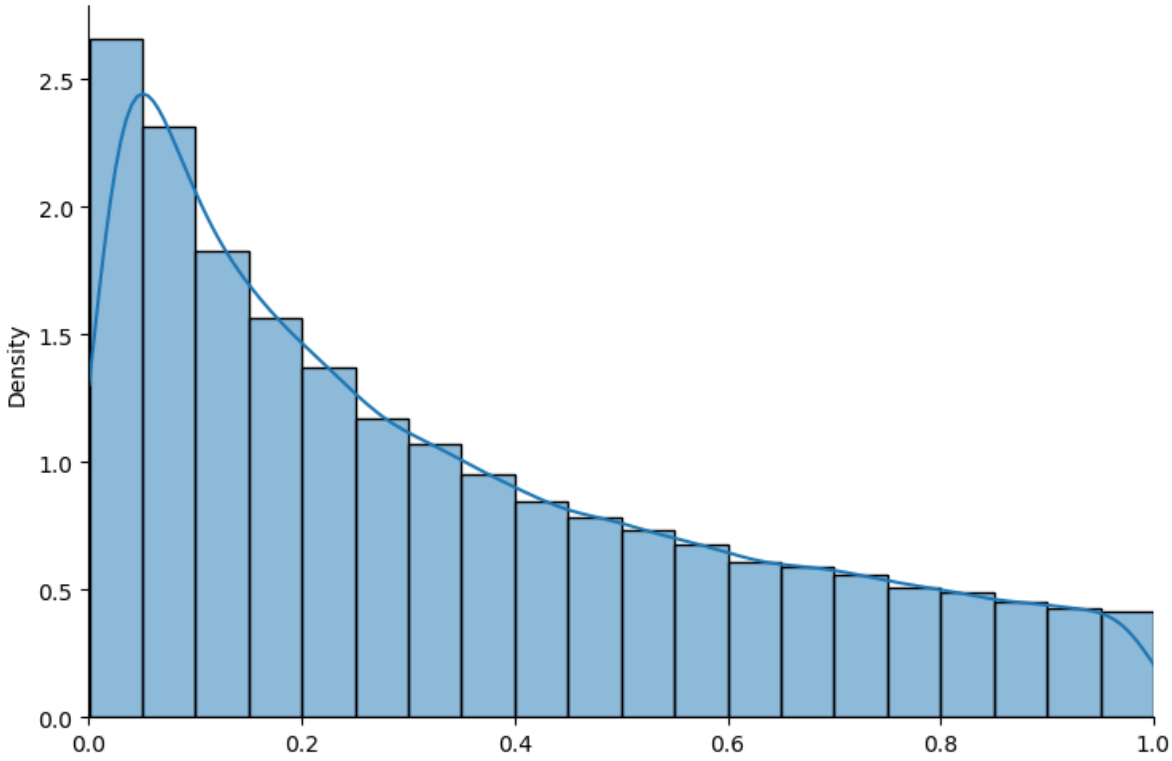
To examine our alternative prior distributions, we'll plot approximate prior distributions below by calling the `show_prior` method.

We verify that the rejection sampling strategy under `Pyro` produces the same log normal distribution as the truncated normal transformation under `Numpyro`.

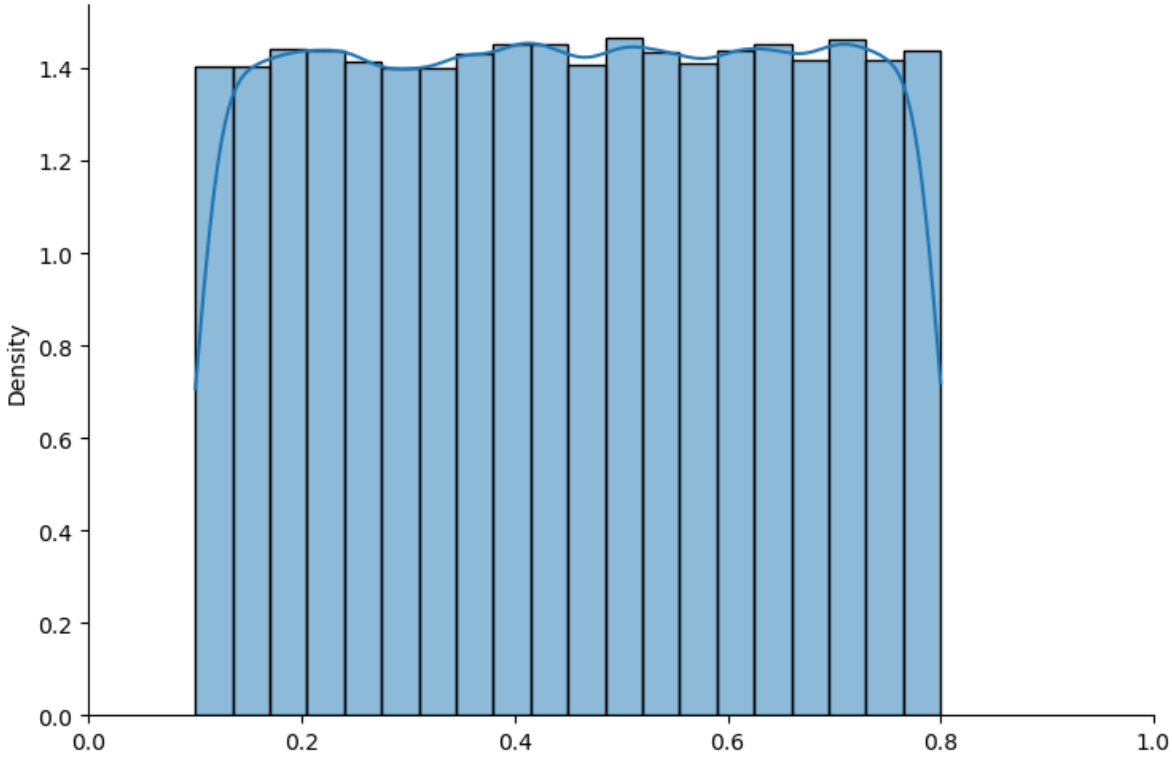
```
# truncated log normal
exampleLN = BayesianInference(param=(0,2), name_dist='lognormal', solver='numpyro')
exampleLN.show_prior(size=100000,bins=20)

# truncated uniform
exampleUN = BayesianInference(param=(0.1,0.8), name_dist='uniform', solver='numpyro')
exampleUN.show_prior(size=100000,bins=20)
```

```
CUDA backend failed to initialize: Found CUDA version 12010, but JAX was built␣
↪against version 12020, which is newer. The copy of CUDA that is installed must␣
↪be at least as new as the version against which JAX was built. (Set TF_CPP_MIN_
↪LOG_LEVEL=0 and rerun for more info.)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```




```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

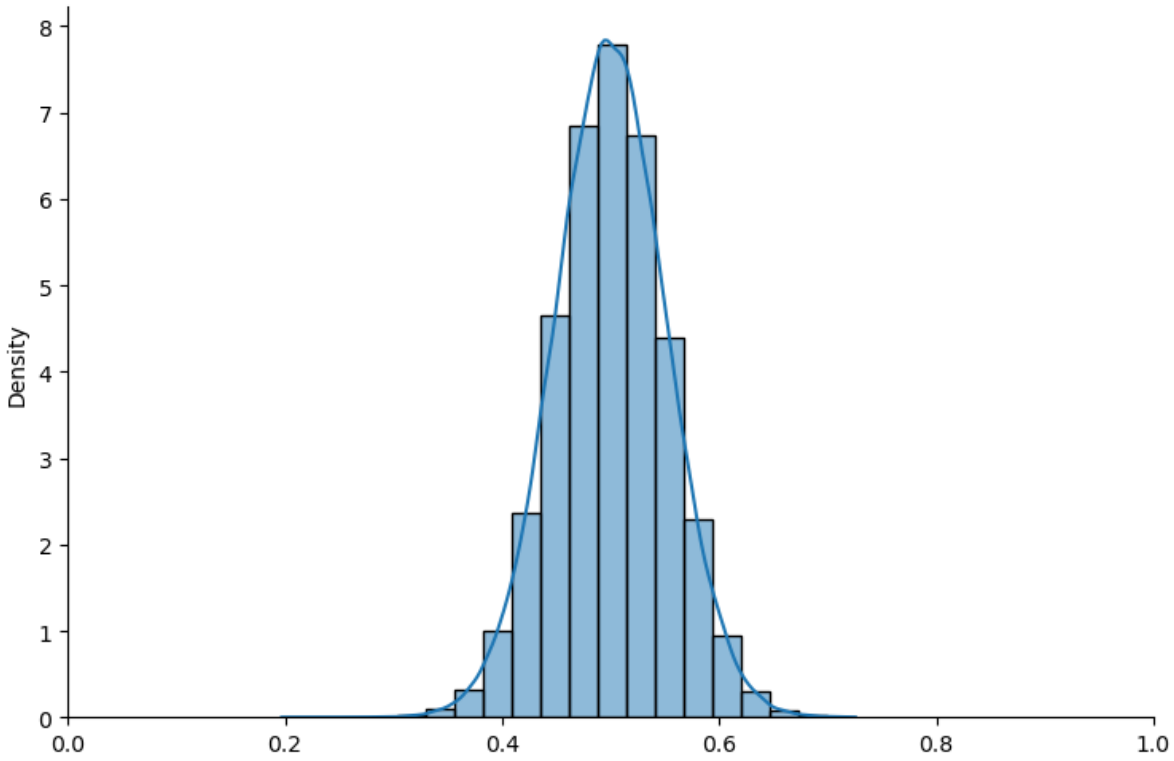


The above graphs show that sampling seems to work well with both distributions.

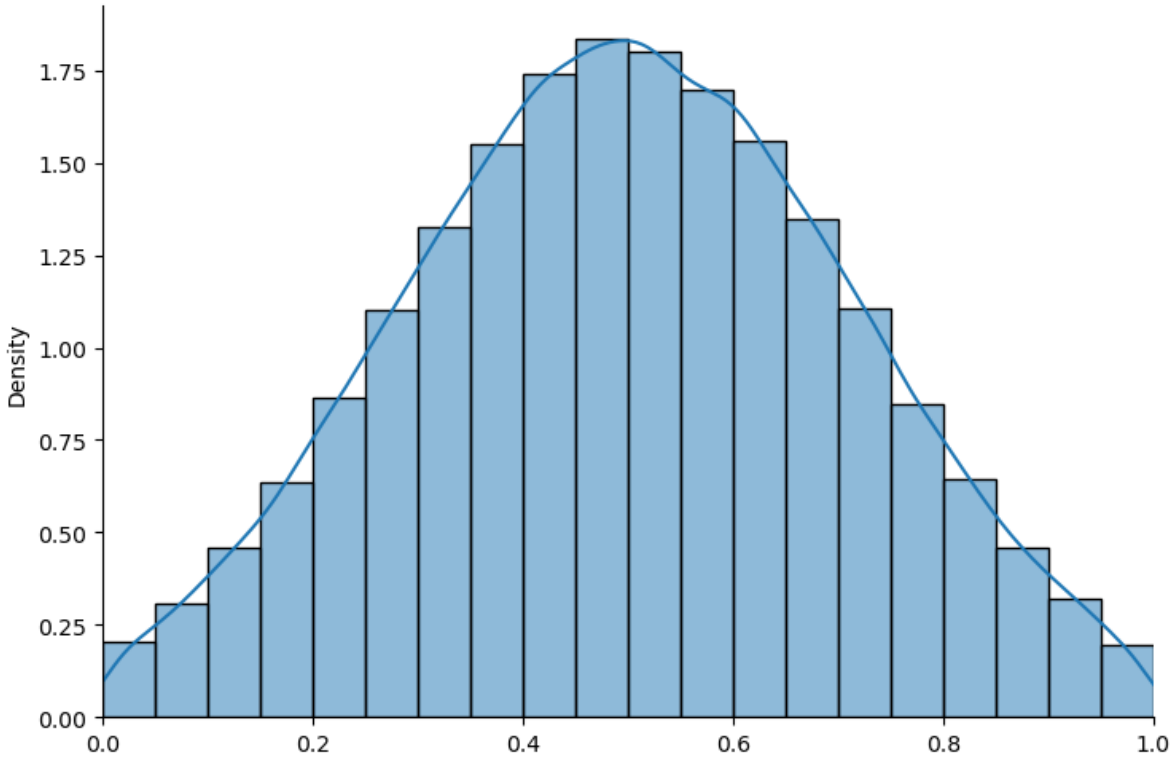
Now let's see how well things work with a couple of von Mises distributions.

```
# shifted von Mises
exampleVM = BayesianInference(param=10, name_dist='vonMises', solver='numpyro')
exampleVM.show_prior(size=100000,bins=20)

# truncated von Mises
exampleVM_trunc = BayesianInference(param=20, name_dist='vonMises', solver='pyro')
exampleVM_trunc.show_prior(size=100000,bins=20)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```




```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

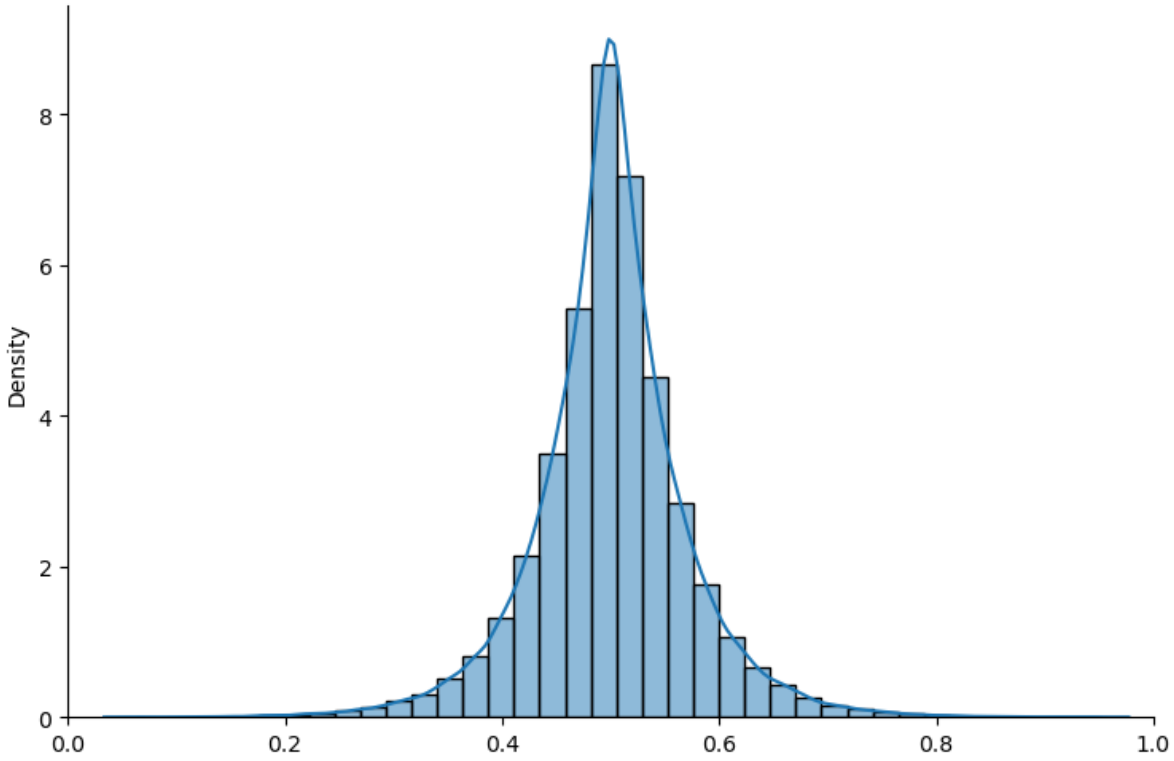


These graphs look good too.

Now let's try with a Laplace distribution.

```
# truncated Laplace
exampleLP = BayesianInference(param=(0.5,0.05), name_dist='laplace', solver='numpyro')
exampleLP.show_prior(size=100000,bins=40)
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```



Having assured ourselves that our sampler seems to do a good job, let's put it to work in using MCMC to compute posterior probabilities.

8.5 Posteriors Via MCMC and VI

We construct a class `BayesianInferencePlot` to implement MCMC or VI algorithms and plot multiple posteriors for different updating data sizes and different possible prior.

This class takes as inputs the true data generating parameter 'theta', a list of updating data sizes for multiple posterior plotting, and a defined and parametrized `BayesianInference` class.

It has two key methods:

- `BayesianInferencePlot.MCMC_plot()` takes wanted MCMC sample size as input and plot the output posteriors together with the prior defined in `BayesianInference` class.
- `BayesianInferencePlot.SVI_plot()` takes wanted VI distribution class ('beta' or 'normal') as input and plot the posteriors together with the prior.

```
class BayesianInferencePlot:
    """
    Easily implement the MCMC and VI inference for a given instance of␣
↪BayesianInference class and
    plot the prior together with multiple posteriors

    Parameters
    ----------
    theta : float.
        the true DGP parameter
```

(continues on next page)

(continued from previous page)

```
    N_list : list.
        a list of sample size
    BayesianInferenceClass : class.
        a class initiated using BayesianInference()

    """

    def __init__(self, theta, N_list, BayesianInferenceClass, binwidth=0.02):
        """
        Enter Parameters for data generation and plotting
        """
        self.theta = theta
        self.N_list = N_list
        self.BayesianInferenceClass = BayesianInferenceClass

        # plotting parameters
        self.binwidth = binwidth
        self.linewidth=0.05
        self.colorlist = sns.color_palette(n_colors=len(N_list))

        # data generation
        N_max = max(N_list)
        self.data = simulate_draw(theta, N_max)


    def MCMC_plot(self, num_samples, num_warmup=1000):
        """
        Parameters as in MCMC_sampling except that data is already defined
        """
        fig, ax = plt.subplots(figsize=(10, 6))

        # plot prior
        prior_sample = self.BayesianInferenceClass.show_prior(disp_plot=0)
        sns.histplot(
            data=prior_sample, kde=True, stat='density',
            binwidth=self.binwidth,
            color='#4C4E52',
            linewidth=self.linewidth,
            alpha=0.1,
            ax=ax,
            label='Prior Distribution'
            )

        # plot posteriors
        for id, n in enumerate(self.N_list):
            samples = self.BayesianInferenceClass.MCMC_sampling(
                self.data[:n], num_samples, num_warmup
            )
            sns.histplot(
                samples, kde=True, stat='density',
                binwidth=self.binwidth,
                linewidth=self.linewidth,
                alpha=0.2,
                color=self.colorlist[id-1],
                label=f'Posterior with $n={n}$'
                )
```

(continues on next page)

(continued from previous page)

```
        ax.legend()
        ax.set_title('MCMC Sampling density of Posterior Distributions', fontsize=15)
        plt.xlim(0, 1)
        plt.show()


    def SVI_fitting(self, guide_dist, params):
        """
        Fit the beta/truncnormal curve using parameters trained by SVI.
        I create plot using PDF given by scipy.stats distributions since torch.dist␣
↪do not have embedded PDF methods.
        """
        # create x axis
        xaxis = np.linspace(0,1,1000)
        if guide_dist=='beta':
            y = st.beta.pdf(xaxis, a=params['alpha_q'], b=params['beta_q'])

        elif guide_dist=='normal':

            # rescale upper/lower bound. See Scipy's truncnorm doc
            lower, upper = (0, 1)
            loc, scale = params['loc'], params['scale']
            a, b = (lower - loc) / scale, (upper - loc) / scale

            y = st.truncnorm.pdf(xaxis, a=a, b=b, loc=params['loc'], scale=params[
↪'scale'])
        return (xaxis, y)


    def SVI_plot(self, guide_dist, n_steps=2000):
        """
        Parameters as in SVI_run except that data is already defined
        """
        fig, ax = plt.subplots(figsize=(10, 6))

        # plot prior
        prior_sample = self.BayesianInferenceClass.show_prior(disp_plot=0)
        sns.histplot(
            data=prior_sample, kde=True, stat='density',
            binwidth=self.binwidth,
            color='#4C4E52',
            linewidth=self.linewidth,
            alpha=0.1,
            ax=ax,
            label='Prior Distribution'
            )

        # plot posteriors
        for id, n in enumerate(self.N_list):
            (params, losses) = self.BayesianInferenceClass.SVI_run(self.data[:n],␣
↪guide_dist, n_steps)
            x, y = self.SVI_fitting(guide_dist, params)
            ax.plot(x, y,
                alpha=1,
                color=self.colorlist[id-1],
                label=f'Posterior with $n={n}$'
```

(continues on next page)

(continued from previous page)

```
                )
        ax.legend()
        ax.set_title(f'SVI density of Posterior Distributions with {guide_dist} guide
↪', fontsize=15)
        plt.xlim(0, 1)
        plt.show()
```

Let's set some parameters that we'll use in all of the examples below.

To save computer time at first, notice that we'll set `MCMC_num_samples = 2000` and `SVI_num_steps = 5000`.

(Later, to increase accuracy of approximations, we'll want to increase these.)

```
num_list = [5,10,50,100,1000]
MCMC_num_samples = 2000
SVI_num_steps = 5000

# theta is the data generating process
true_theta = 0.8
```

8.5.1 Beta Prior and Posteriors:

Let's compare outcomes when we use a Beta prior.

For the same Beta prior, we shall

- compute posteriors analytically
- compute posteriors using MCMC via `Pyro` and `Numpyro`.
- compute posteriors using VI via `Pyro` and `Numpyro`.

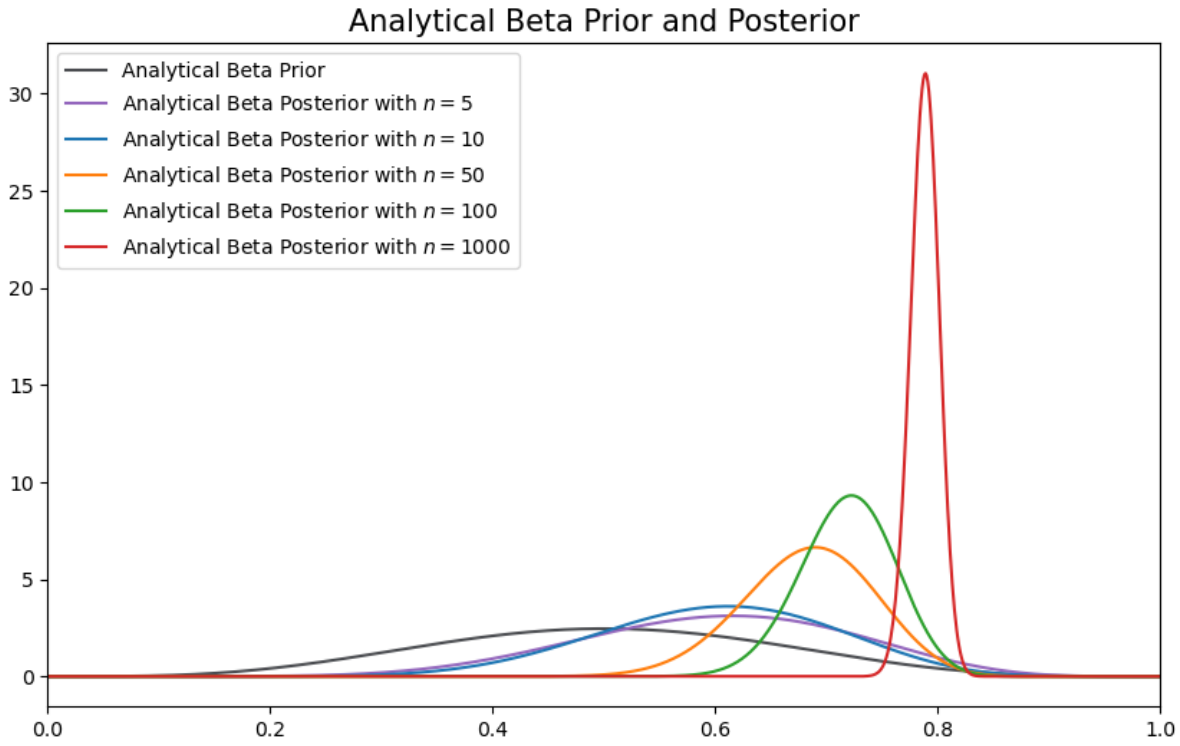
Let's start with the analytical method that we described in this quantecon lecture https://python.quantecon.org/prob_meaning.html

```
# First examine Beta priors
BETA_pyro = BayesianInference(param=(5,5), name_dist='beta', solver='pyro')
BETA_numpyro = BayesianInference(param=(5,5), name_dist='beta', solver='numpyro')

BETA_pyro_plot = BayesianInferencePlot(true_theta, num_list, BETA_pyro)
BETA_numpyro_plot = BayesianInferencePlot(true_theta, num_list, BETA_numpyro)


# plot analytical Beta prior and posteriors
xaxis = np.linspace(0,1,1000)
y_prior = st.beta.pdf(xaxis, 5, 5)

fig, ax = plt.subplots(figsize=(10, 6))
# plot analytical beta prior
ax.plot(xaxis, y_prior, label='Analytical Beta Prior', color='#4C4E52')

data, colorlist, N_list = BETA_pyro_plot.data, BETA_pyro_plot.colorlist, BETA_pyro_
↪plot.N_list
# plot analytical beta posteriors
for id, n in enumerate(N_list):
    func = analytical_beta_posterior(data[:n], alpha0=5, beta0=5)
```

(continues on next page)

(continued from previous page)

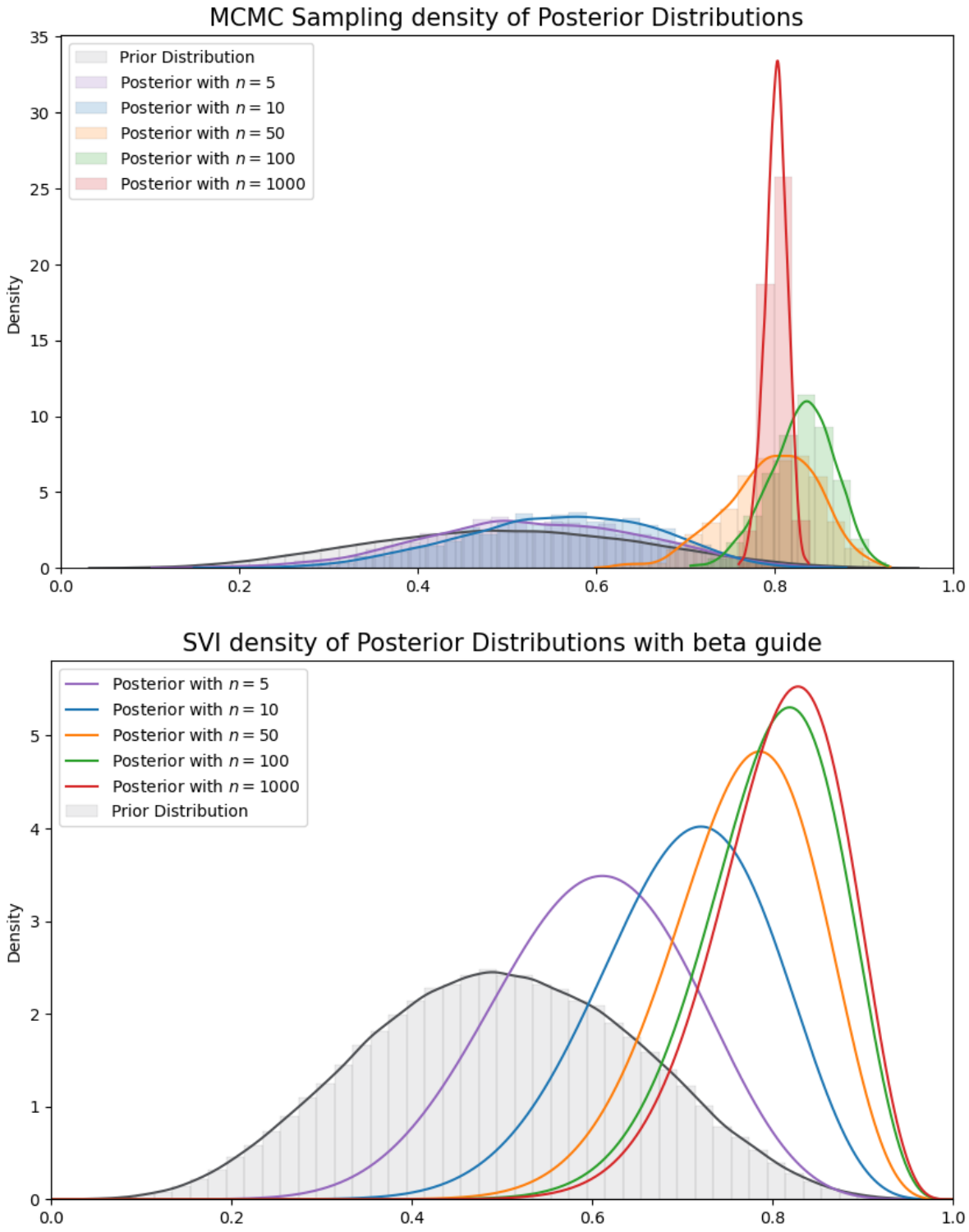
```
    y_posterior = func.pdf(xaxis)
    ax.plot(
        xaxis, y_posterior, color=colorlist[id-1], label=f'Analytical Beta Posterior␣
↪with $n={n}$')
ax.legend()
ax.set_title('Analytical Beta Prior and Posterior', fontsize=15)
plt.xlim(0, 1)
plt.show()
```



Now let's use MCMC while still using a beta prior.

We'll do this for both MCMC and VI.

```
BayesianInferencePlot(true_theta, num_list, BETA_pyro).MCMC_plot(num_samples=MCMC_num_
↪samples)
BayesianInferencePlot(true_theta, num_list, BETA_numpyro).SVI_plot(guide_dist='beta',␣
↪n_steps=SVI_num_steps)
```



Here the MCMC approximation looks good.

But the VI approximation doesn't look so good.

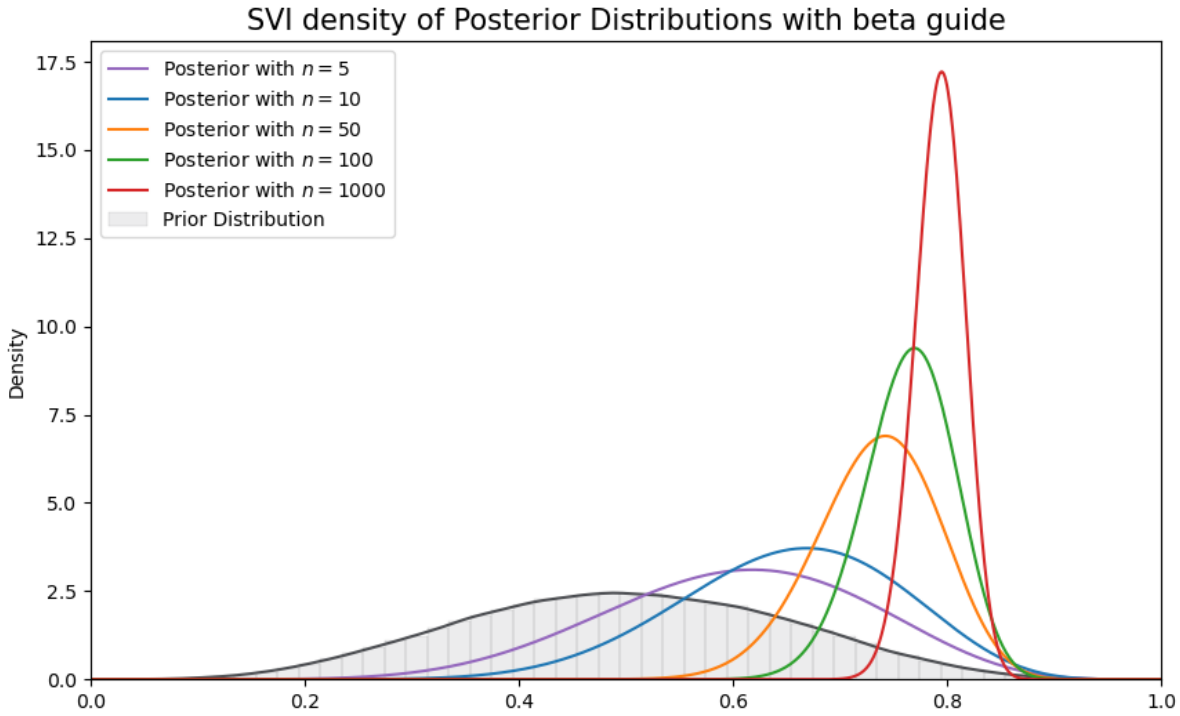
- even though we use the beta distribution as our guide, the VI approximated posterior distributions do not closely resemble the posteriors that we had just computed analytically.

(Here, our initial parameter for Beta guide is (0.5, 0.5).)

But if we increase the number of steps from 5000 to 10000 in VI as we now shall do, we'll get VI-approximated posteriors will be more accurate, as we shall see next.

(Increasing the step size increases computational time though).

```
BayesianInferencePlot(true_theta, num_list, BETA_numpyro).SVI_plot(guide_dist='beta',␣
↪n_steps=100000)
```



8.6 Non-conjugate Prior Distributions

Having assured ourselves that our MCMC and VI methods can work well when we have conjugate prior and so can also compute analytically, we next proceed to situations in which our prior is not a beta distribution, so we don't have a conjugate prior.

So we will have non-conjugate priors and are cast into situations in which we can't calculate posteriors analytically.

8.6.1 MCMC

First, we implement and display MCMC.

We first initialize the `BayesianInference` classes and then can directly call `BayesianInferencePlot` to plot both MCMC and SVI approximating posteriors.

```
# Initialize BayesianInference classes
# try uniform
STD_UNIFORM_pyro = BayesianInference(param=(0,1), name_dist='uniform', solver='pyro')
UNIFORM_numpyro = BayesianInference(param=(0.2,0.7), name_dist='uniform', solver=
↪'numpyro')
```

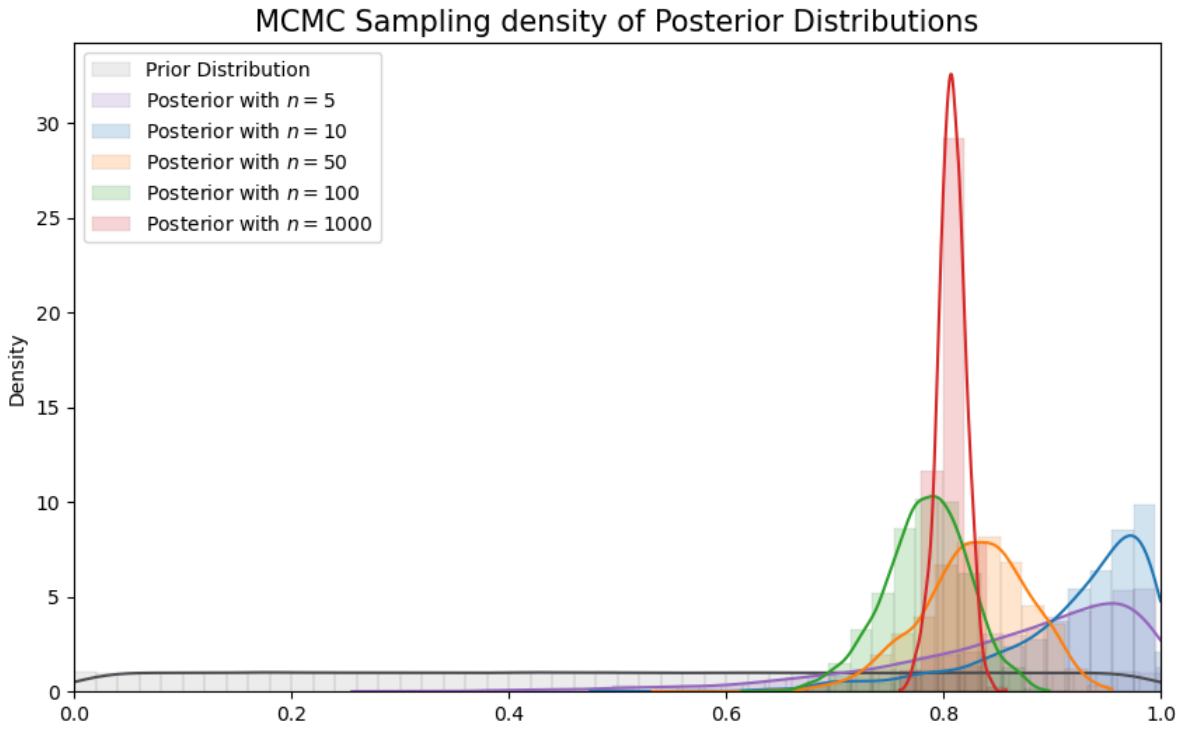
(continues on next page)

(continued from previous page)

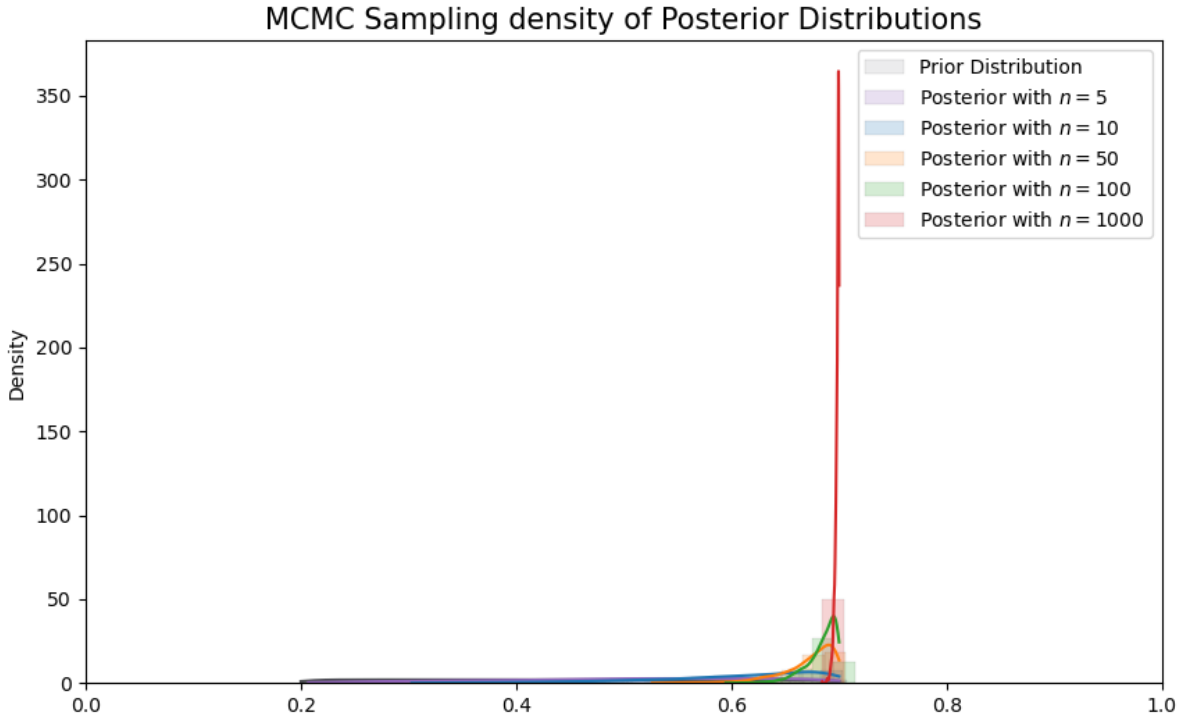
```
# try truncated lognormal
LOGNORMAL_numpyro = BayesianInference(param=(0,2), name_dist='lognormal', solver=
 ↪'numpyro')
LOGNORMAL_pyro = BayesianInference(param=(0,2), name_dist='lognormal', solver='pyro')

# try von Mises
# shifted von Mises
VONMISES_numpyro = BayesianInference(param=10, name_dist='vonMises', solver='numpyro')
# truncated von Mises
VONMISES_pyro = BayesianInference(param=40, name_dist='vonMises', solver='pyro')

# try laplace
LAPLACE_numpyro = BayesianInference(param=(0.5, 0.07), name_dist='laplace', solver=
 ↪'numpyro')
```

```
# Uniform
example_CLASS = STD_UNIFORM_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
 ↪num_samples)

example_CLASS = UNIFORM_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
 ↪num_samples)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: pyro
```




```
=======INFO=======
Parameters: (0.2, 0.7)
Prior Dist: uniform
Solver: numpyro
```



In the situation depicted above, we have assumed a $Uniform(\underline{\theta}, \overline{\theta})$ prior that puts zero probability outside a bounded

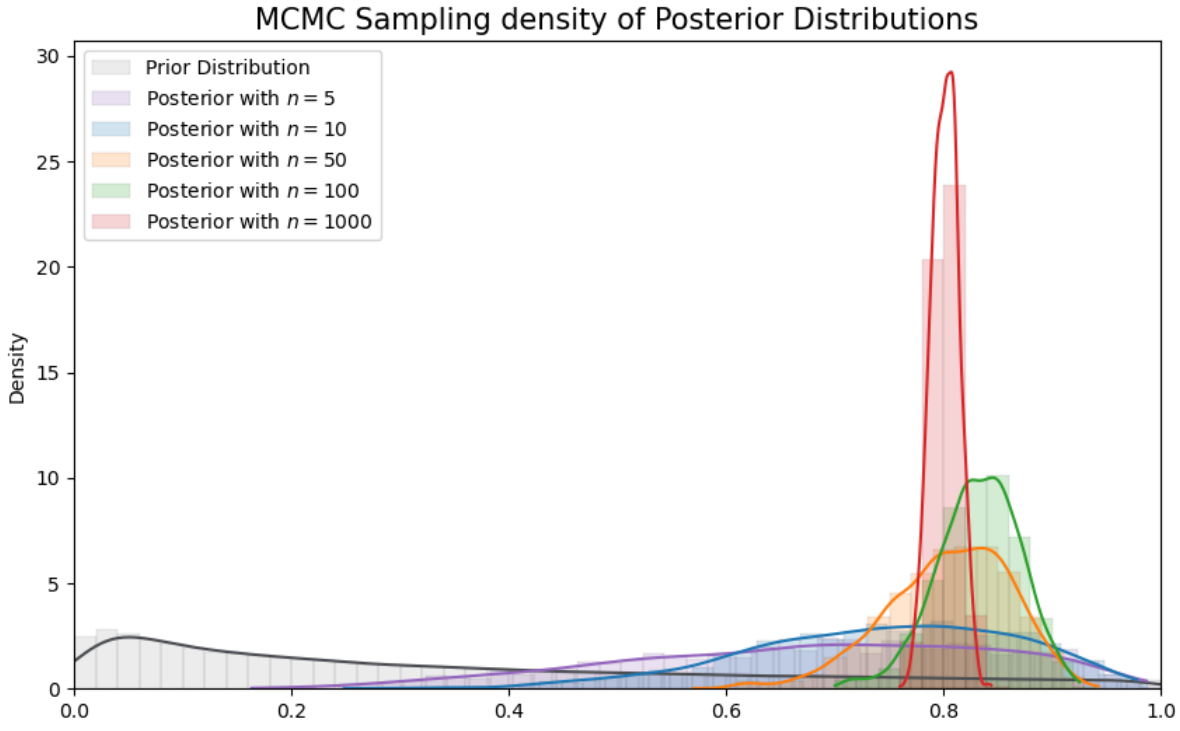
support that excludes the true value.

Consequently, the posterior cannot put positive probability above $\overline{\theta}$ or below $\underline{\theta}$.

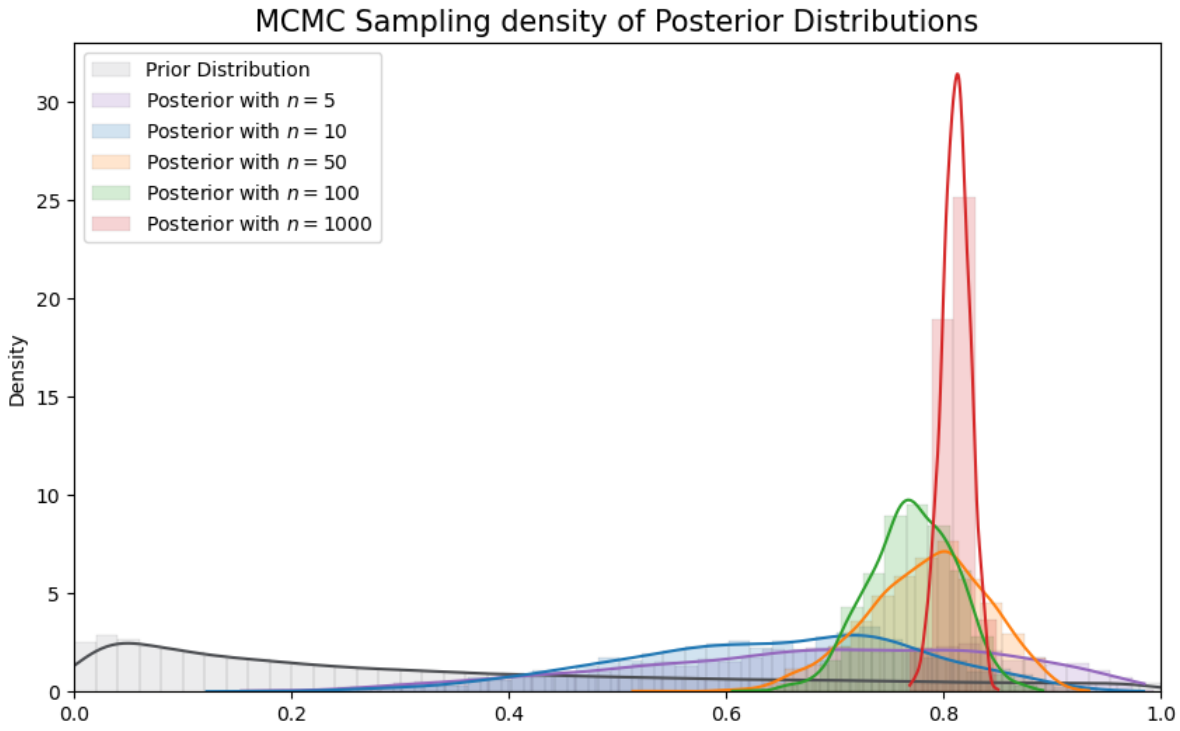
Note how when the true data-generating θ is located at 0.8 as it is here, when n gets large, the posterior concentrate on the upper bound of the support of the prior, 0.7 here.

```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)

example_CLASS = LOGNORMAL_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```

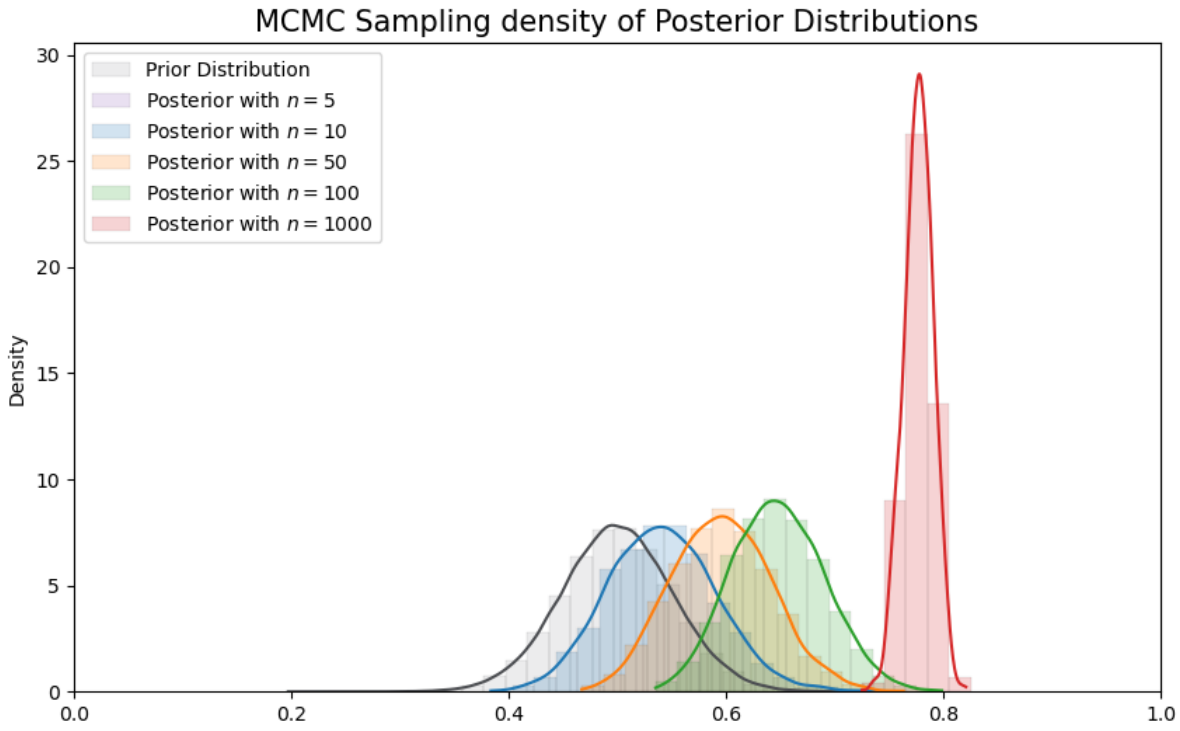



```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: pyro
```

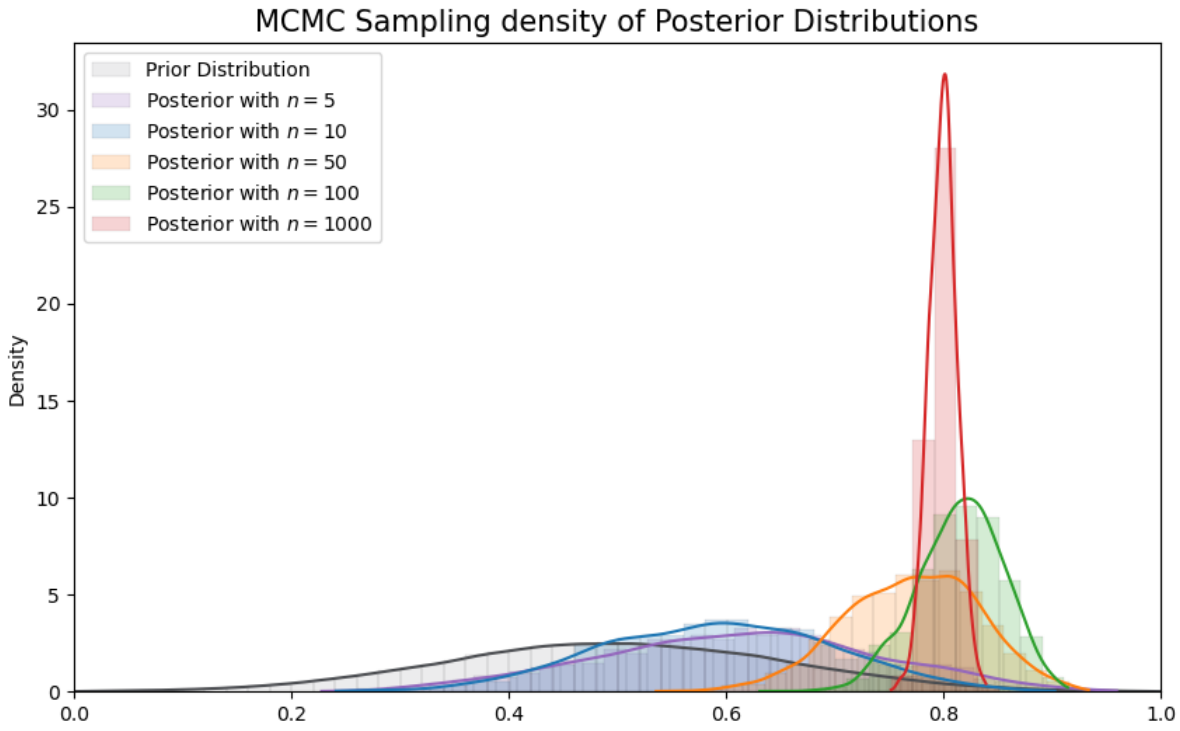



```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNOTE: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
 ↪num_samples)

example_CLASS = VONMISES_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNOTE: Truncated von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
 ↪num_samples)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NOTE: Shifted von Mises
```

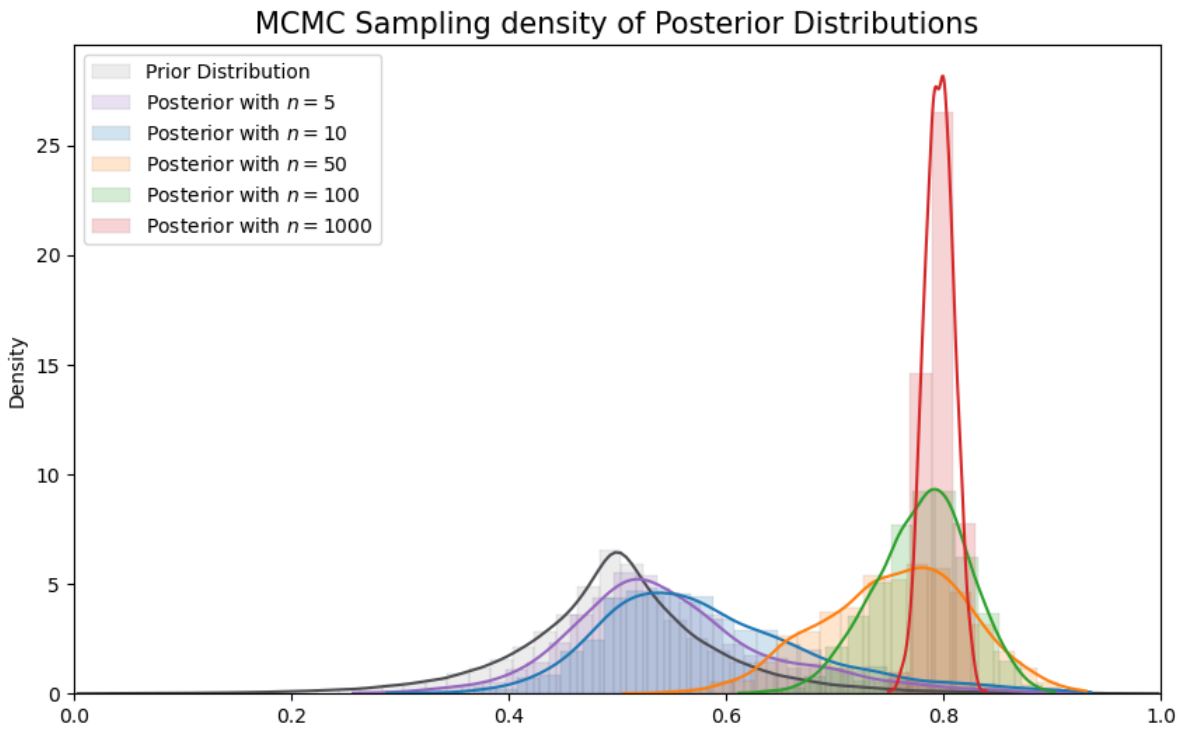



```
=======INFO=======
Parameters: 40
Prior Dist: vonMises
Solver: pyro

NOTE: Truncated von Mises
```




```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).MCMC_plot(num_samples=MCMC_
↪num_samples)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```



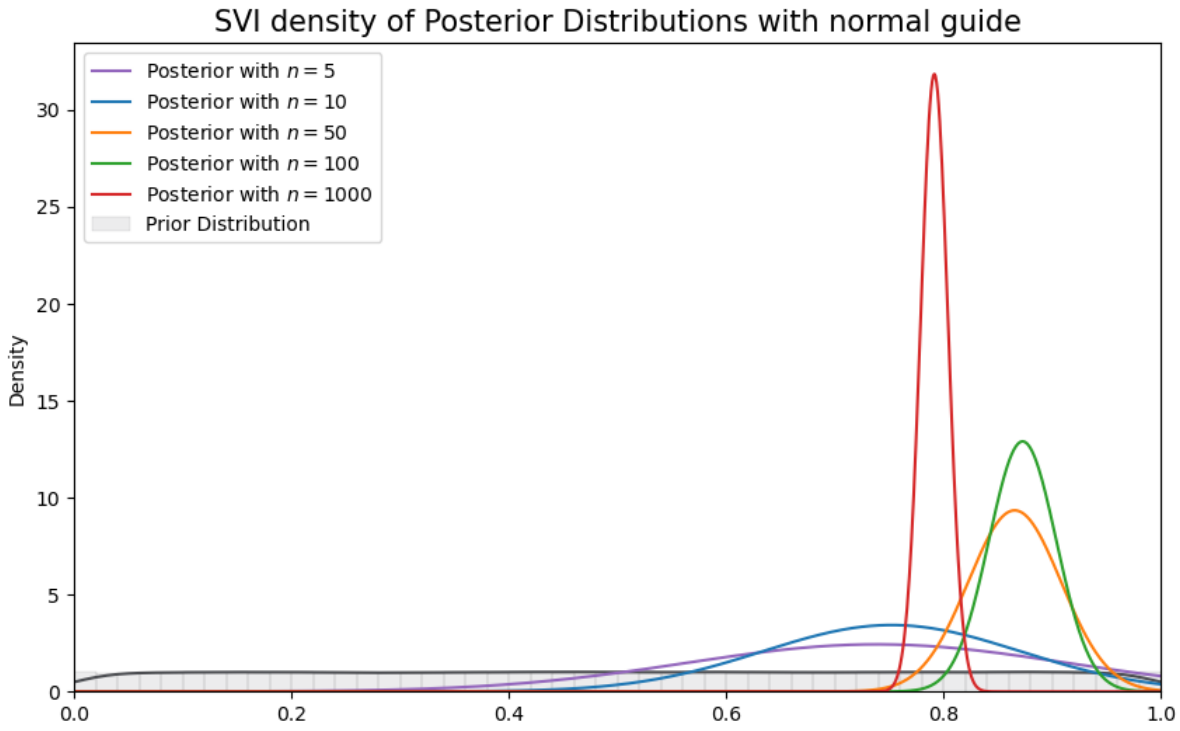
To get more accuracy we will now increase the number of steps for Variational Inference (VI)

```
SVI_num_steps = 50000
```

VI with a Truncated Normal Guide

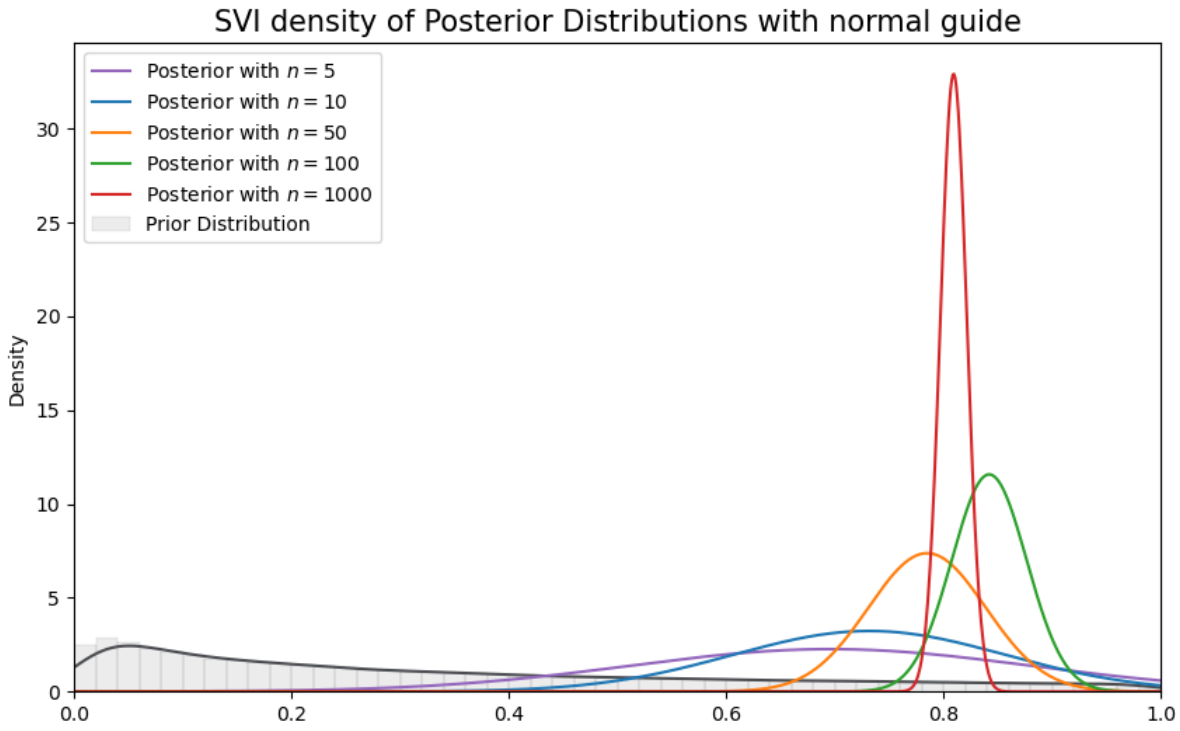
```
# Uniform
example_CLASS = BayesianInference(param=(0,1), name_dist='uniform', solver='numpyro')
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: numpyro
```



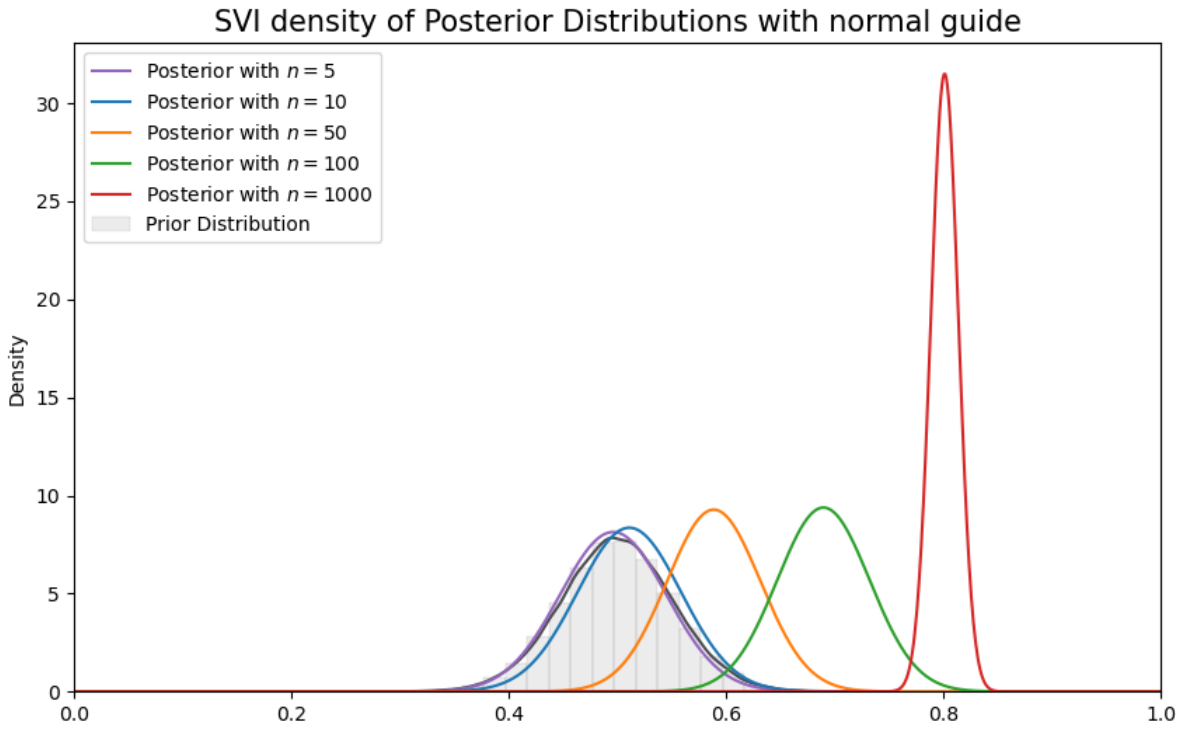

```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```



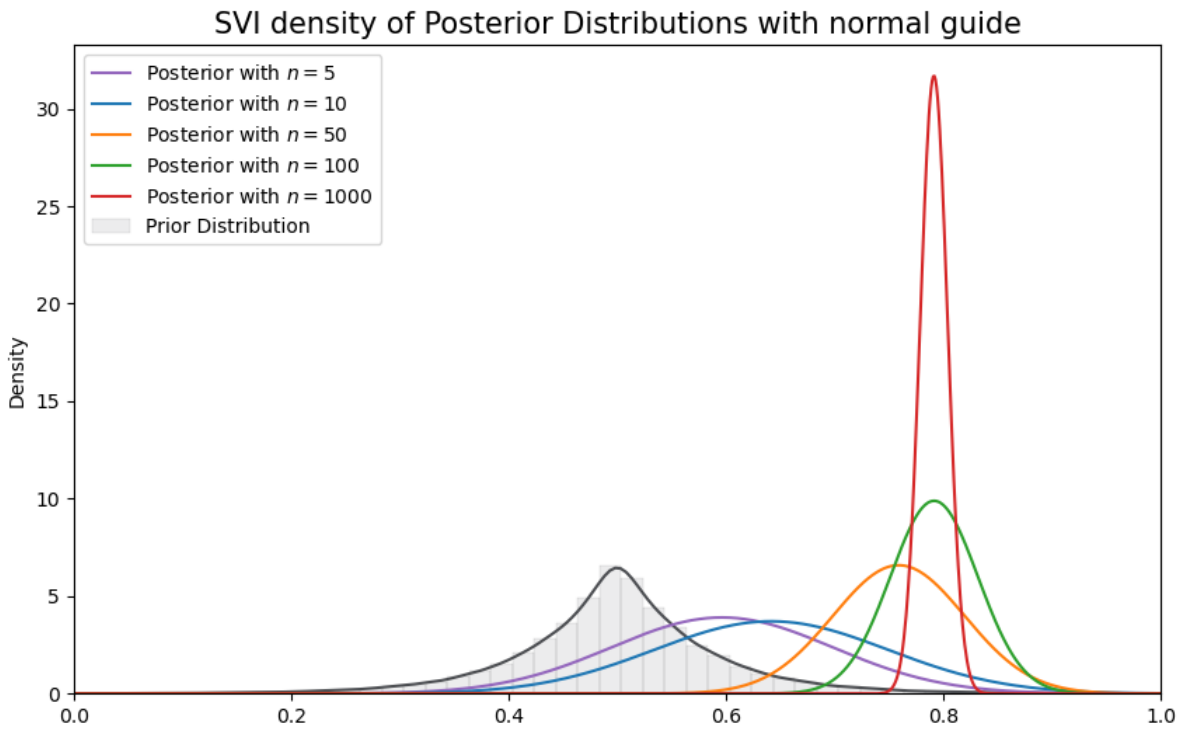

```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
 ↪', n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NB: Shifted von Mises
```




```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='normal
↪', n_steps=SVI_num_steps)
```

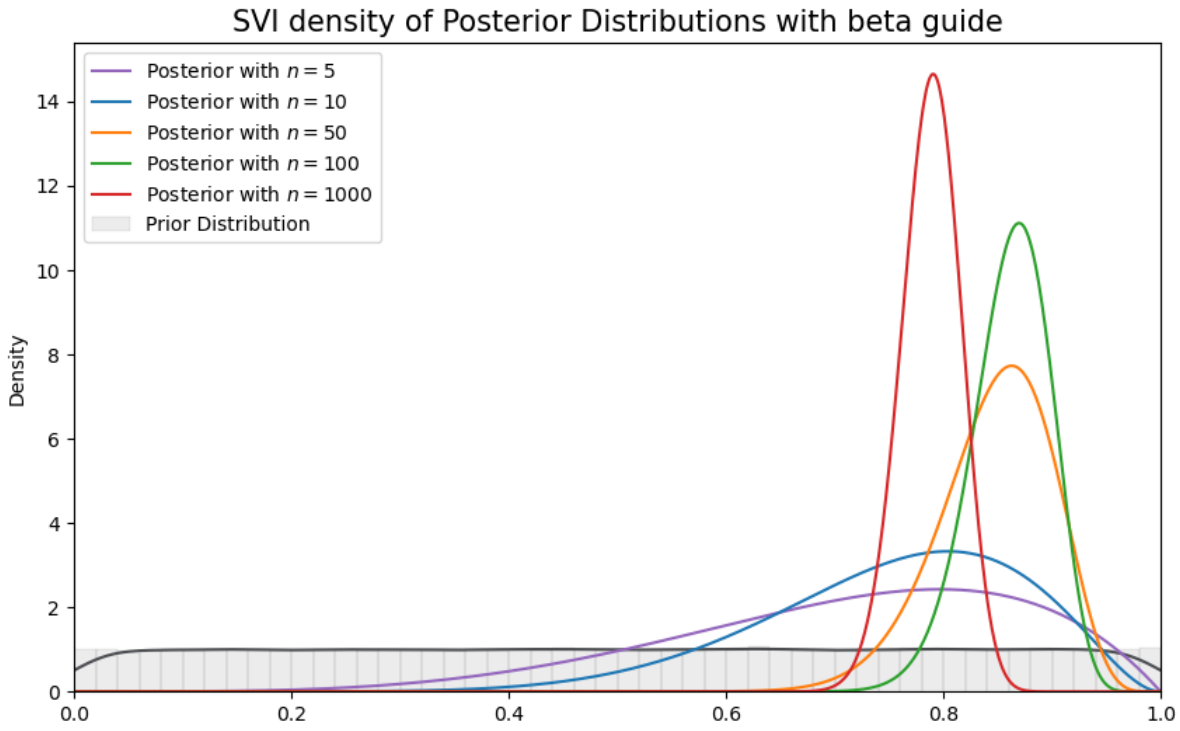
```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```



Variational Inference with a Beta Guide Distribution

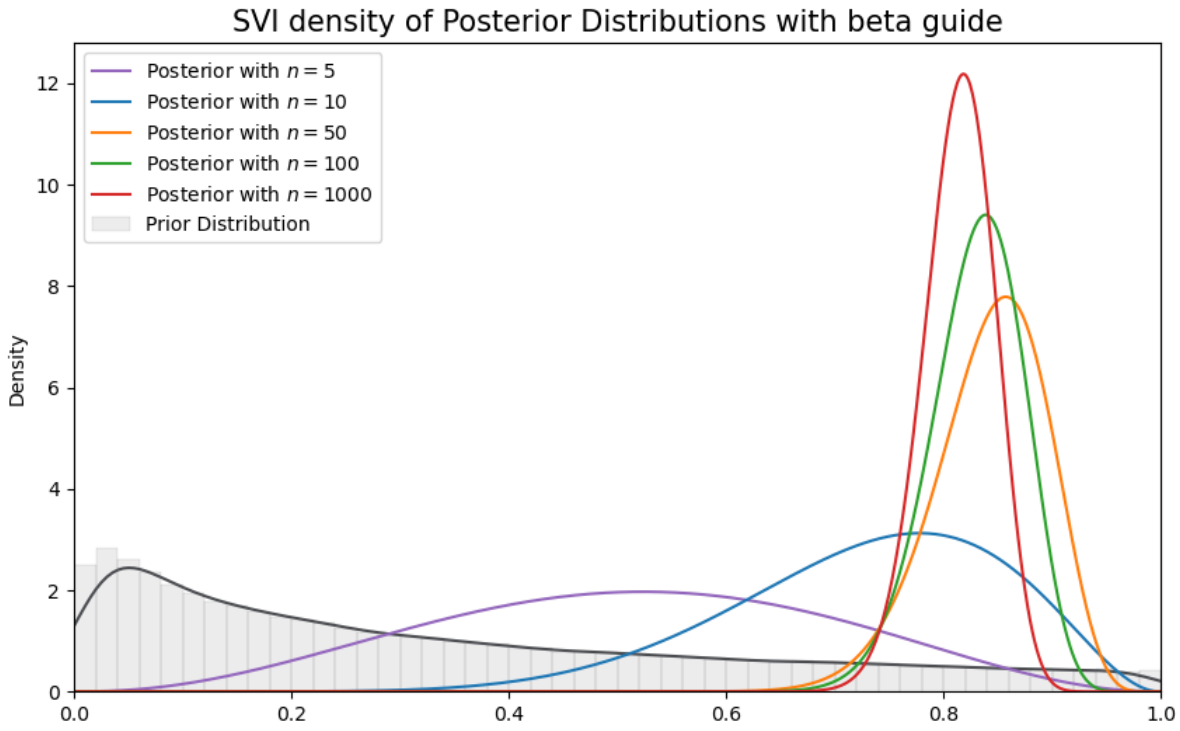
```
# Uniform
example_CLASS = STD_UNIFORM_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
Solver: pyro
```

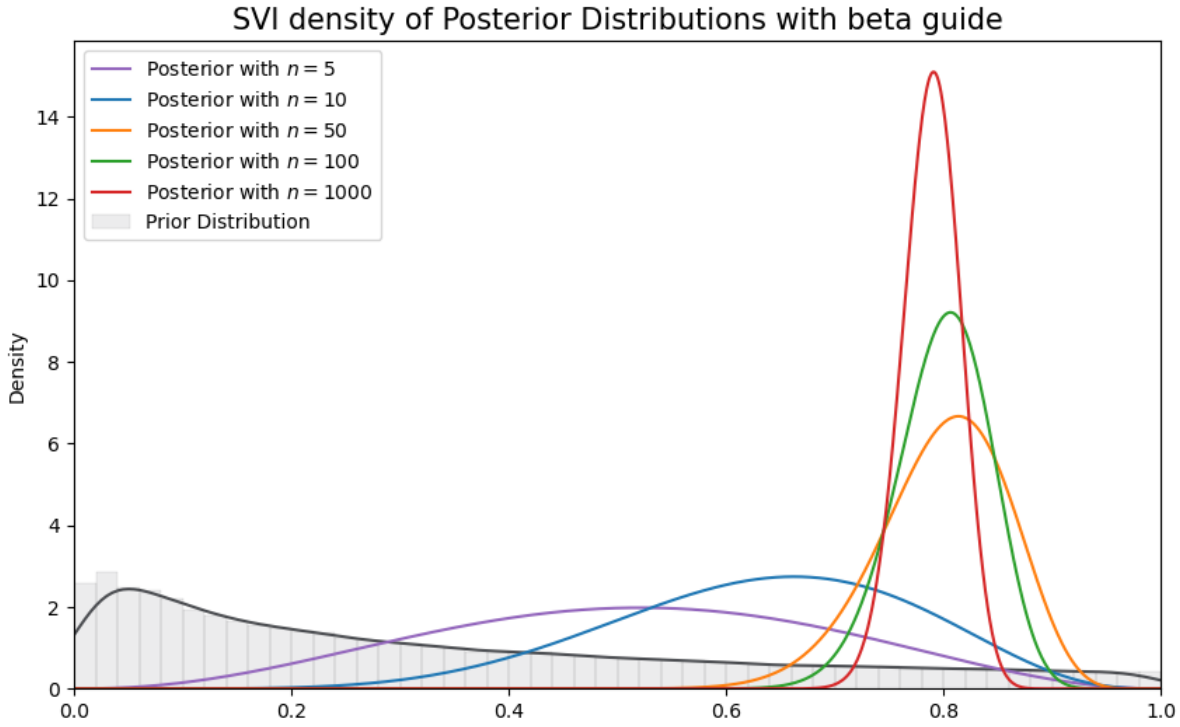



```
# Log Normal
example_CLASS = LOGNORMAL_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
 ↪ n_steps=SVI_num_steps)

example_CLASS = LOGNORMAL_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
 ↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: numpyro
```

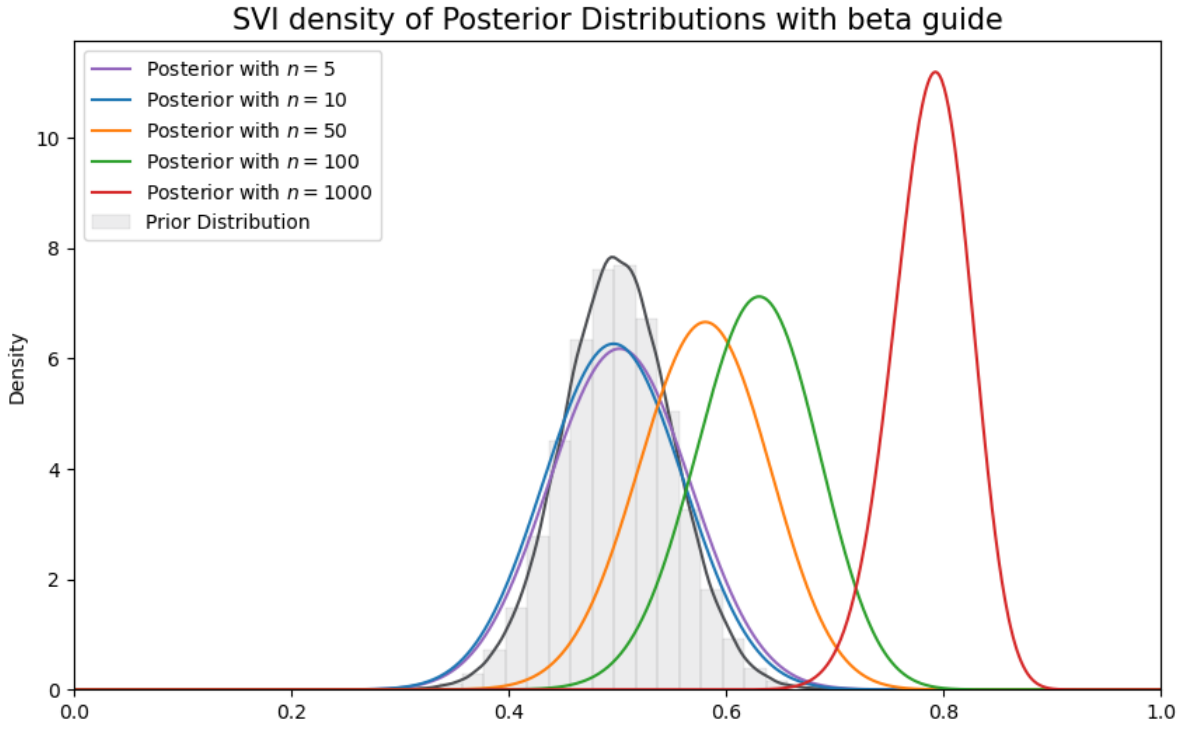



```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
Solver: pyro
```

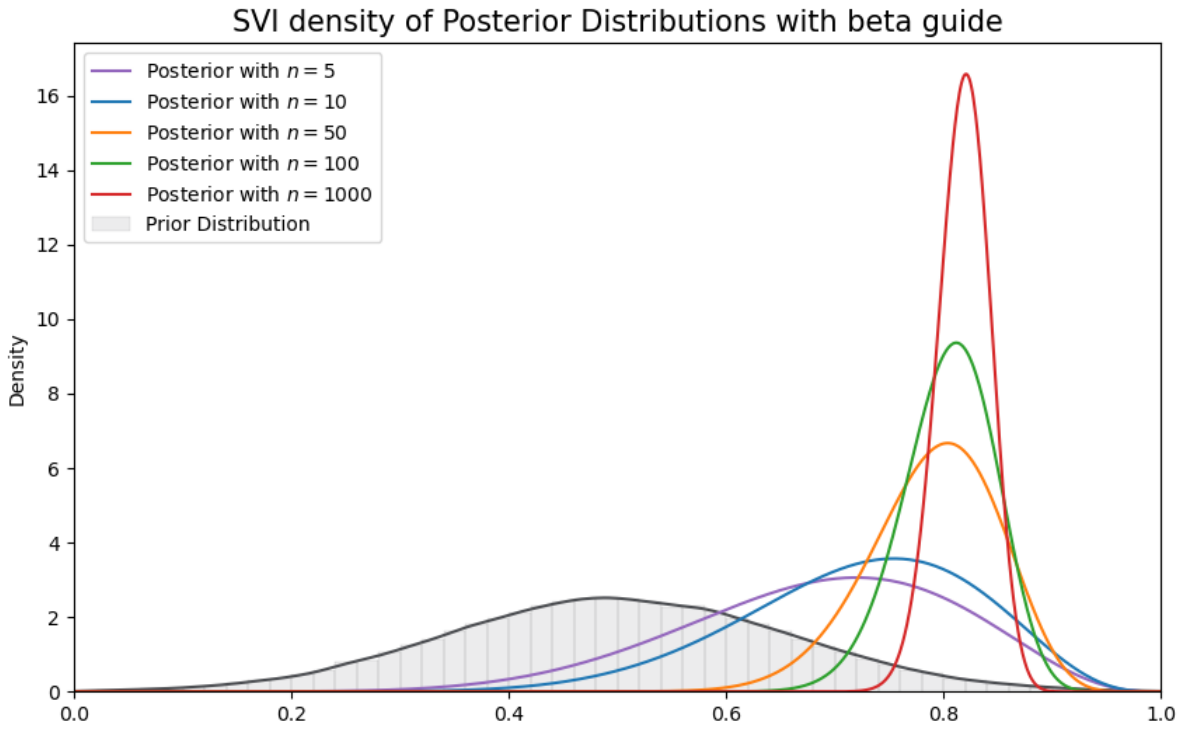


```
# Von Mises
example_CLASS = VONMISES_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Shifted von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)

example_CLASS = VONMISES_pyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
print('\nNB: Truncated von Mises')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Solver: numpyro

NB: Shifted von Mises
```

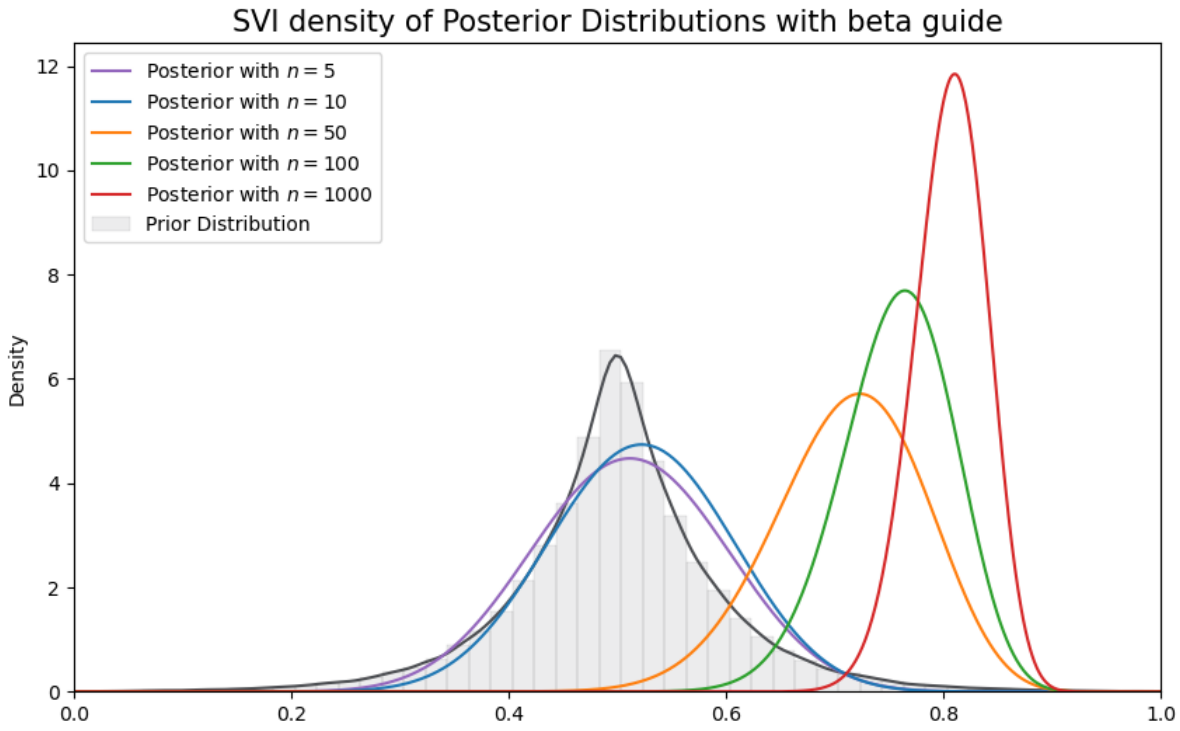



```
=======INFO=======
Parameters: 40
Prior Dist: vonMises
Solver: pyro

NB: Truncated von Mises
```




```
# Laplace
example_CLASS = LAPLACE_numpyro
print(f'=======INFO=======\nParameters: {example_CLASS.param}\nPrior Dist: {example_
 ↪CLASS.name_dist}\nSolver: {example_CLASS.solver}')
BayesianInferencePlot(true_theta, num_list, example_CLASS).SVI_plot(guide_dist='beta',
 ↪ n_steps=SVI_num_steps)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
Solver: numpyro
```


SVI density of Posterior Distributions with beta guide
Posterior with $n = 5$
Posterior with $n = 10$
Posterior with $n = 50$
Posterior with $n = 100$
Posterior with $n = 1000$
Prior Distribution
Density
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0

CHAPTER

NINE

POSTERIOR DISTRIBUTIONS FOR AR(1) PARAMETERS

We'll begin with some Python imports.

```
!pip install arviz pymc numpyro jax
```

```
import arviz as az
import pymc as pmc
import numpyro
from numpyro import distributions as dist

import numpy as np
import jax.numpy as jnp
from jax import random
import matplotlib.pyplot as plt

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

This lecture uses Bayesian methods offered by pymc and numpyro to make statistical inferences about two parameters of a univariate first-order autoregression.

The model is a good laboratory for illustrating consequences of alternative ways of modeling the distribution of the initial y_0:

- As a fixed number
- As a random variable drawn from the stationary distribution of the $\{y_t\}$ stochastic process

The first component of the statistical model is

$$y_{t+1} = \rho y_t + \sigma_x \epsilon_{t+1}, \quad t \geq 0 \tag{9.1}$$

where the scalars ρ and σ_x satisfy $|\rho| < 1$ and $\sigma_x > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The second component of the statistical model is

$$y_0 \sim N(\mu_0, \sigma_0^2) \tag{9.2}$$

Consider a sample $\{y_t\}_{t=0}^T$ governed by this statistical model.

The model implies that the likelihood function of $\{y_t\}_{t=0}^T$ can be **factored**:

$$f(y_T, y_{T-1}, \ldots, y_0) = f(y_T|y_{T-1}) f(y_{T-1}|y_{T-2}) \cdots f(y_1|y_0) f(y_0)$$

where we use f to denote a generic probability density.

The statistical model (9.1)-(9.2) implies

$$f(y_t|y_{t-1}) \sim \mathcal{N}(\rho y_{t-1}, \sigma_x^2)$$
$$f(y_0) \sim \mathcal{N}(\mu_0, \sigma_0^2)$$

We want to study how inferences about the unknown parameters (ρ, σ_x) depend on what is assumed about the parameters μ_0, σ_0 of the distribution of y_0.

Below, we study two widely used alternative assumptions:

- $(\mu_0, \sigma_0) = (y_0, 0)$ which means that y_0 is drawn from the distribution $\mathcal{N}(y_0, 0)$; in effect, we are **conditioning on an observed initial value**.
- μ_0, σ_0 are functions of ρ, σ_x because y_0 is drawn from the stationary distribution implied by ρ, σ_x.

Note: We do **not** treat a third possible case in which μ_0, σ_0 are free parameters to be estimated.

Unknown parameters are ρ, σ_x.

We have independent **prior probability distributions** for ρ, σ_x and want to compute a posterior probability distribution after observing a sample $\{y_t\}_{t=0}^T$.

The notebook uses `pymc4` and `numpyro` to compute a posterior distribution of ρ, σ_x. We will use NUTS samplers to generate samples from the posterior in a chain. Both of these libraries support NUTS samplers.

NUTS is a form of Monte Carlo Markov Chain (MCMC) algorithm that bypasses random walk behaviour and allows for convergence to a target distribution more quickly. This not only has the advantage of speed, but allows for complex models to be fitted without having to employ specialised knowledge regarding the theory underlying those fitting methods.

Thus, we explore consequences of making these alternative assumptions about the distribution of y_0:

- A first procedure is to condition on whatever value of y_0 is observed. This amounts to assuming that the probability distribution of the random variable y_0 is a Dirac delta function that puts probability one on the observed value of y_0.
- A second procedure assumes that y_0 is drawn from the stationary distribution of a process described by (9.1) so that $y_0 \sim N\left(0, \frac{\sigma_x^2}{(1-\rho)^2}\right)$

When the initial value y_0 is far out in a tail of the stationary distribution, conditioning on an initial value gives a posterior that is **more accurate** in a sense that we'll explain.

Basically, when y_0 happens to be in a tail of the stationary distribution and we **don't condition on** y_0, the likelihood function for $\{y_t\}_{t=0}^T$ adjusts the posterior distribution of the parameter pair ρ, σ_x to make the observed value of y_0 more likely than it really is under the stationary distribution, thereby adversely twisting the posterior in short samples.

An example below shows how not conditioning on y_0 adversely shifts the posterior probability distribution of ρ toward larger values.

We begin by solving a **direct problem** that simulates an AR(1) process.

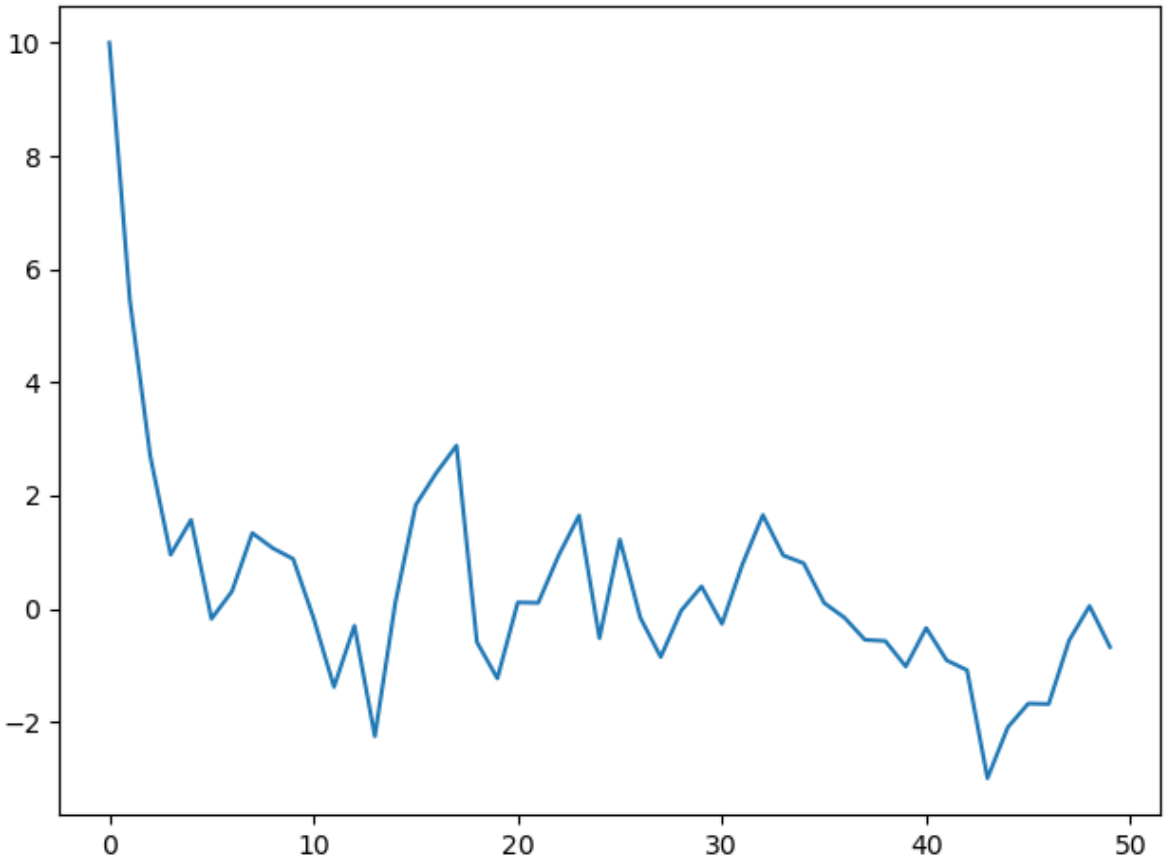
How we select the initial value y_0 matters.

- If we think y_0 is drawn from the stationary distribution $\mathcal{N}(0, \frac{\sigma_x^2}{1-\rho^2})$, then it is a good idea to use this distribution as $f(y_0)$. Why? Because y_0 contains information about ρ, σ_x.
- If we suspect that y_0 is far in the tails of the stationary distribution – so that variation in early observations in the sample have a significant **transient component** – it is better to condition on y_0 by setting $f(y_0) = 1$.

To illustrate the issue, we'll begin by choosing an initial y_0 that is far out in a tail of the stationary distribution.

```
def ar1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0.,sigma,T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho*y[t-1] + eps[t]

    return y

sigma =  1.
rho = 0.5
T = 50

np.random.seed(145353452)
y = ar1_simulate(rho, sigma, 10, T)
```

```
plt.plot(y)
plt.tight_layout()
```



Now we shall use Bayes' law to construct a posterior distribution, conditioning on the initial value of y_0.

(Later we'll assume that y_0 is drawn from the stationary distribution, but not now.)

First we'll use **pymc4**.

9.1 PyMC Implementation

For a normal distribution in `pymc`, $var = 1/\tau = \sigma^2$.

```
AR1_model = pmc.Model()

with AR1_model:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

    # Expected value of y at the next period (rho * y)
    yhat = rho * y[:-1]

    # Likelihood of the actual realization
    y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
```

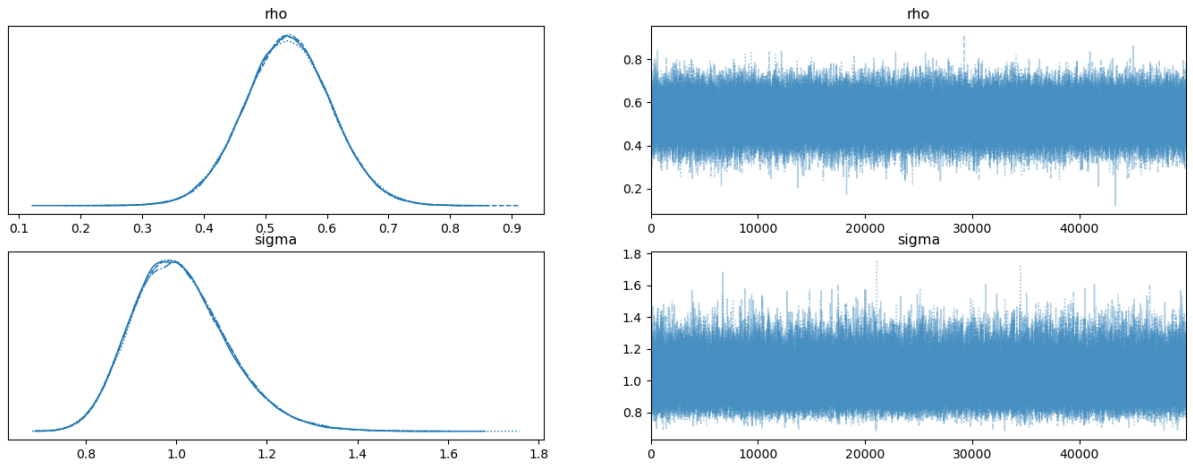
pmc.sample by default uses the NUTS samplers to generate samples as shown in the below cell:

```
with AR1_model:
    trace = pmc.sample(50000, tune=10000, return_inferencedata=True)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
with AR1_model:
    az.plot_trace(trace, figsize=(17,6))
```



Evidently, the posteriors aren't centered on the true values of $.5, 1$ that we used to generate the data.

This is a symptom of the classic **Hurwicz bias** for first order autoregressive processes (see Leonid Hurwicz [Hur50].)

The Hurwicz bias is worse the smaller is the sample (see [OW69]).

Be that as it may, here is more information about the posterior.

```
with AR1_model:
    summary = az.summary(trace, round_to=4)

summary
```

```
         mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho    0.5364  0.0709  0.4035   0.6707     0.0002   0.0001  168990.5957
sigma  1.0102  0.1064  0.8201   1.2126     0.0003   0.0002  179997.0195

          ess_tail  r_hat
rho    122188.2762    1.0
sigma  142155.7267    1.0
```

Now we shall compute a posterior distribution after seeing the same data but instead assuming that y_0 is drawn from the stationary distribution.

This means that

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

We alter the code as follows:

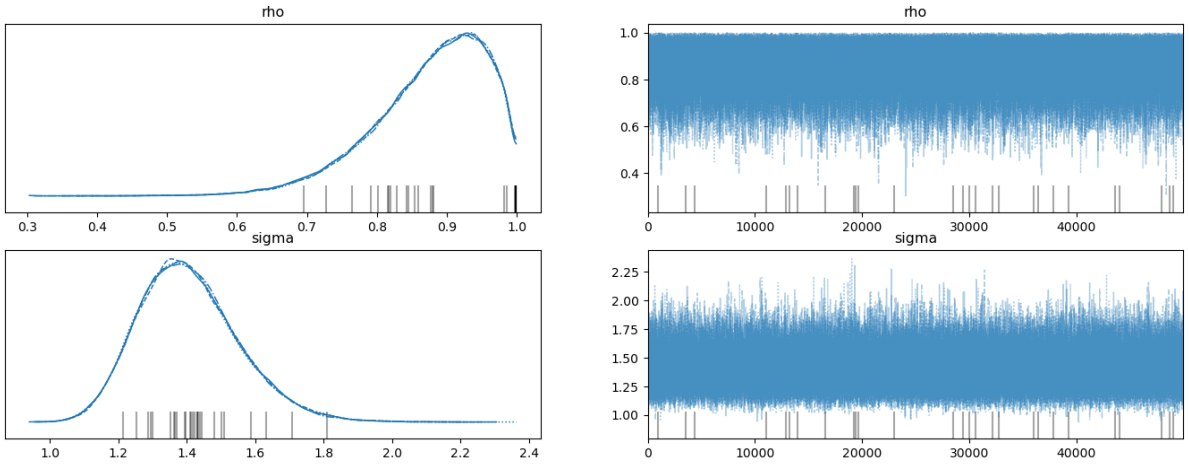
```
AR1_model_y0 = pmc.Model()

with AR1_model_y0:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma=np.sqrt(10))

    # Standard deviation of ergodic y
    y_sd = sigma / np.sqrt(1 - rho**2)

    # yhat
    yhat = rho * y[:-1]
    y_data = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
    y0_data = pmc.Normal('y0_obs', mu=0., sigma=y_sd, observed=y[0])
```

```
with AR1_model_y0:
    trace_y0 = pmc.sample(50000, tune=10000, return_inferencedata=True)

# Grey vertical lines are the cases of divergence
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
with AR1_model_y0:
    az.plot_trace(trace_y0, figsize=(17,6))
```




```
with AR1_model:
    summary_y0 = az.summary(trace_y0, round_to=4)

summary_y0
```

```
          mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho     0.8757  0.0812  0.7320   0.9985     0.0002   0.0002  107159.5599
sigma   1.4052  0.1469  1.1443   1.6868     0.0004   0.0003  142322.9231

           ess_tail  r_hat
rho     79689.2489    1.0
sigma  113205.7043    1.0
```

Please note how the posterior for ρ has shifted to the right relative to when we conditioned on y_0 instead of assuming that y_0 is drawn from the stationary distribution.

Think about why this happens.

Hint: It is connected to how Bayes Law (conditional probability) solves an **inverse problem** by putting high probability on parameter values that make observations more likely.

We'll return to this issue after we use `numpyro` to compute posteriors under our two alternative assumptions about the distribution of y_0.

We'll now repeat the calculations using `numpyro`.

9.2 Numpyro Implementation

```
def plot_posterior(sample):
    """
    Plot trace and histogram
    """
    # To np array
    rhos = sample['rho']
    sigmas = sample['sigma']
    rhos, sigmas, = np.array(rhos), np.array(sigmas)
```

(continues on next page)

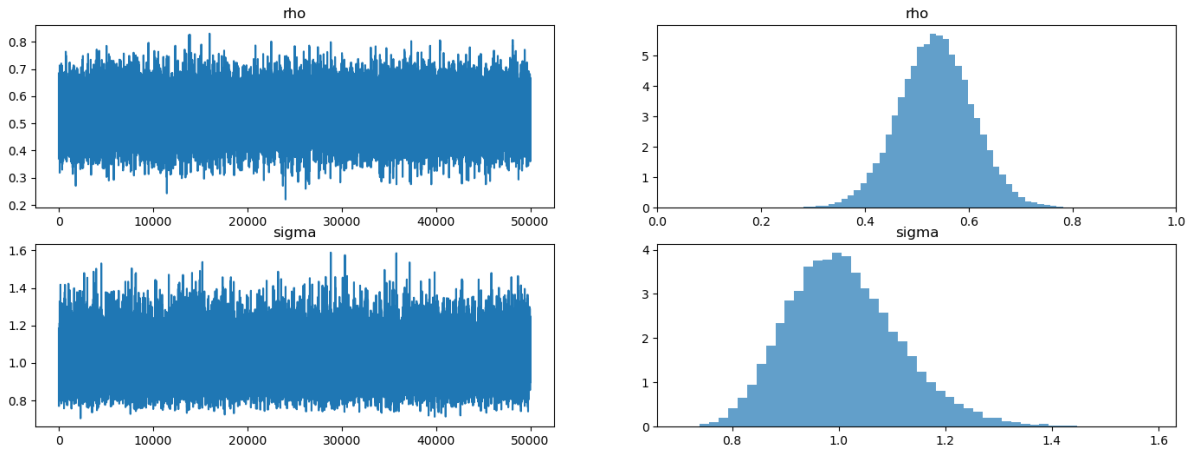
(continued from previous page)

```
    fig, axs = plt.subplots(2, 2, figsize=(17, 6))
    # Plot trace
    axs[0, 0].plot(rhos)    # rho
    axs[1, 0].plot(sigmas) # sigma

    # Plot posterior
    axs[0, 1].hist(rhos, bins=50, density=True, alpha=0.7)
    axs[0, 1].set_xlim([0, 1])
    axs[1, 1].hist(sigmas, bins=50, density=True, alpha=0.7)

    axs[0, 0].set_title("rho")
    axs[0, 1].set_title("rho")
    axs[1, 0].set_title("sigma")
    axs[1, 1].set_title("sigma")
    plt.show()
```

```
def AR1_model(data):
    # set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]

    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
NUTS_kernel = numpyro.infer.NUTS(AR1_model)

# Run MCMC
mcmc = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
 ↪bar=False)
mcmc.run(rng_key=random.PRNGKey(1), data=y)
```

```
plot_posterior(mcmc.get_samples())
```




```
mcmc.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho       0.54      0.07      0.54      0.42      0.66  34735.88      1.00
     sigma       1.01      0.11      1.00      0.83      1.18  33245.42      1.00

Number of divergences: 0
```

Next, we again compute the posterior under the assumption that y_0 is drawn from the stationary distribution, so that

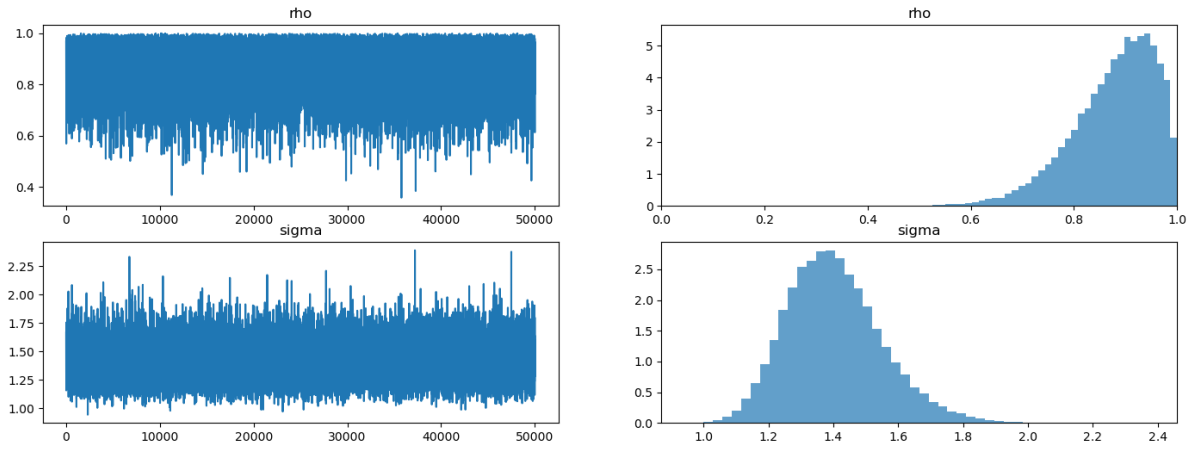
$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

Here's the new code to achieve this.

```
def AR1_model_y0(data):
    # Set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Standard deviation of ergodic y
    y_sd = sigma / jnp.sqrt(1 - rho**2)

    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]

    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
    y0_data = numpyro.sample('y0_obs', dist.Normal(loc=0., scale=y_sd), obs=data[0])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
NUTS_kernel = numpyro.infer.NUTS(AR1_model_y0)

# Run MCMC
mcmc2 = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
 ↪bar=False)
mcmc2.run(rng_key=random.PRNGKey(1), data=y)
```

```
plot_posterior(mcmc2.get_samples())
```




```
mcmc2.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho       0.88      0.08      0.89      0.76      1.00  29063.44      1.00
     sigma       1.40      0.15      1.39      1.17      1.64  25674.09      1.00

Number of divergences: 0
```

Look what happened to the posterior!

It has moved far from the true values of the parameters used to generate the data because of how Bayes' Law (i.e., conditional probability) is telling `numpyro` to explain what it interprets as "explosive" observations early in the sample.

Bayes' Law is able to generate a plausible likelihood for the first observation by driving $\rho \to 1$ and $\sigma \uparrow$ in order to raise the variance of the stationary distribution.

Our example illustrates the importance of what you assume about the distribution of initial conditions.

CHAPTER

TEN

FORECASTING AN AR(1) PROCESS

```
!pip install arviz pymc
```

This lecture describes methods for forecasting statistics that are functions of future values of a univariate autogressive process.

The methods are designed to take into account two possible sources of uncertainty about these statistics:

- random shocks that impinge of the transition law
- uncertainty about the parameter values of the AR(1) process

We consider two sorts of statistics:

- prospective values y_{t+j} of a random process $\{y_t\}$ that is governed by the AR(1) process
- sample path properties that are defined as non-linear functions of future values $\{y_{t+j}\}_{j \geq 1}$ at time t

Sample path properties are things like "time to next turning point" or "time to next recession".

To investigate sample path properties we'll use a simulation procedure recommended by Wecker [Wec79].

To acknowledge uncertainty about parameters, we'll deploy `pymc` to construct a Bayesian joint posterior distribution for unknown parameters.

Let's start with some imports.

```
import numpy as np
import arviz as az
import pymc as pmc
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('white')
colors = sns.color_palette()

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

10.1 A Univariate First-Order Autoregressive Process

Consider the univariate AR(1) model:

$$y_{t+1} = \rho y_t + \sigma \epsilon_{t+1}, \quad t \geq 0 \tag{10.1}$$

where the scalars ρ and σ satisfy $|\rho| < 1$ and $\sigma > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The initial condition y_0 is a known number.

Equation (10.1) implies that for $t \geq 0$, the conditional density of y_{t+1} is

$$f(y_{t+1}|y_t; \rho, \sigma) \sim \mathcal{N}(\rho y_t, \sigma^2) \tag{10.2}$$

Further, equation (10.1) also implies that for $t \geq 0$, the conditional density of y_{t+j} for $j \geq 1$ is

$$f(y_{t+j}|y_t; \rho, \sigma) \sim \mathcal{N}\left(\rho^j y_t, \sigma^2 \frac{1-\rho^{2j}}{1-\rho^2}\right) \tag{10.3}$$

The predictive distribution (10.3) that assumes that the parameters ρ, σ are known, which we express by conditioning on them.

We also want to compute a predictive distribution that does not condition on ρ, σ but instead takes account of our uncertainty about them.

We form this predictive distribution by integrating (10.3) with respect to a joint posterior distribution $\pi_t(\rho, \sigma|y^t)$ that conditions on an observed history $y^t = \{y_s\}_{s=0}^t$:

$$f(y_{t+j}|y^t) = \int f(y_{t+j}|y_t; \rho, \sigma)\pi_t(\rho, \sigma|y^t)d\rho d\sigma \tag{10.4}$$

Predictive distribution (10.3) assumes that parameters (ρ, σ) are known.

Predictive distribution (10.4) assumes that parameters (ρ, σ) are uncertain, but have known probability distribution $\pi_t(\rho, \sigma|y^t)$.

We also want to compute some predictive distributions of "sample path statistics" that might include, for example

- the time until the next "recession",
- the minimum value of Y over the next 8 periods,
- "severe recession", and
- the time until the next turning point (positive or negative).

To accomplish that for situations in which we are uncertain about parameter values, we shall extend Wecker's [Wec79] approach in the following way.

- first simulate an initial path of length T_0;
- for a given prior, draw a sample of size N from the posterior joint distribution of parameters (ρ, σ) after observing the initial path;
- for each draw $n = 0, 1, ..., N$, simulate a "future path" of length T_1 with parameters (ρ_n, σ_n) and compute our three "sample path statistics";
- finally, plot the desired statistics from the N samples as an empirical distribution.

10.2 Implementation

First, we'll simulate a sample path from which to launch our forecasts.

In addition to plotting the sample path, under the assumption that the true parameter values are known, we'll plot .9 and .95 coverage intervals using conditional distribution (10.3) described above.

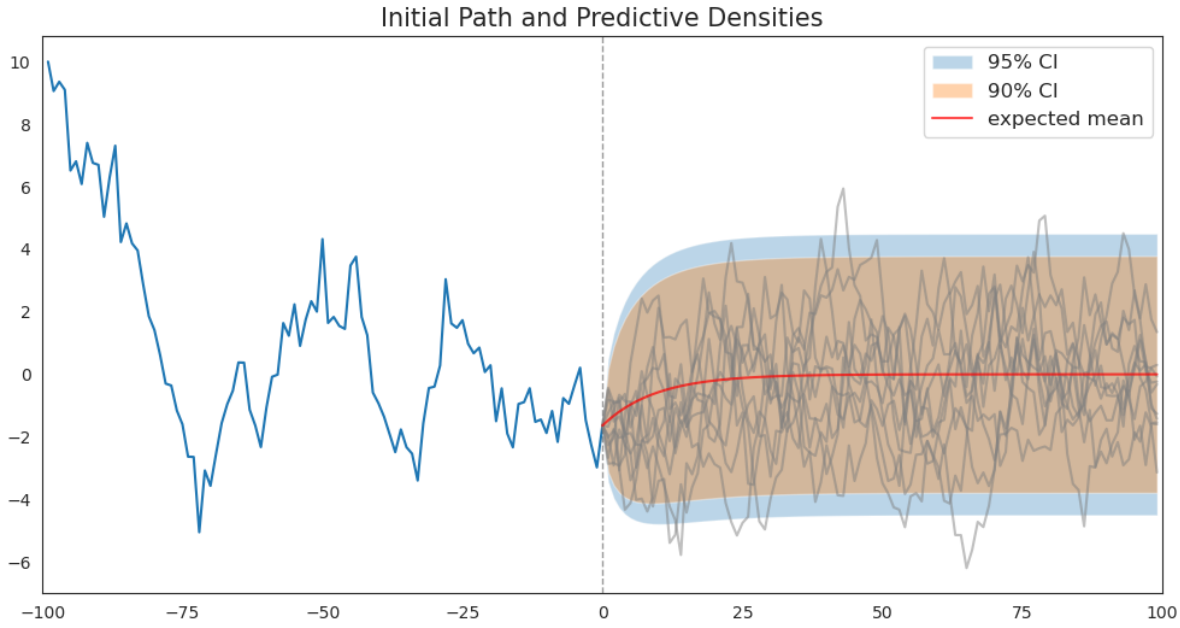
We'll also plot a bunch of samples of sequences of future values and watch where they fall relative to the coverage interval.

```
def AR1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0, sigma, T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho * y[t-1] + eps[t]

    return y


def plot_initial_path(initial_path):
    """
    Plot the initial path and the preceding predictive densities
    """
    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    fig, ax = plt.subplots(1, 1, figsize=(12, 6))
    ax.set_title("Initial Path and Predictive Densities", fontsize=15)
    ax.plot(np.arange(-T0 + 1, 1), initial_path)
    ax.set_xlim([-T0, T1])
    ax.axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for i in range(10):
        y_future = AR1_simulate(rho, sigma, y0, T1)
        ax.plot(np.arange(T1), y_future, color='grey', alpha=.5)

    # Plot 90% CI
    ax.fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3, label='95%␣
↪CI')
    ax.fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35, label='90
↪% CI')
    ax.plot(np.arange(T1), center, color='red', alpha=.7, label='expected mean')
    ax.legend(fontsize=12)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
sigma =  1
rho = 0.9
T0, T1 = 100, 100
y0 = 10

# Simulate
np.random.seed(145)
initial_path = AR1_simulate(rho, sigma, y0, T0)

# Plot
plot_initial_path(initial_path)
```



As functions of forecast horizon, the coverage intervals have shapes like those described in https://python.quantecon.org/perm_income_cons.html

10.3 Predictive Distributions of Path Properties

Wecker [Wec79] proposed using simulation techniques to characterize predictive distribution of some statistics that are non-linear functions of y.

He called these functions "path properties" to contrast them with properties of single data points.

He studied two special prospective path properties of a given series $\{y_t\}$.

The first was **time until the next turning point**.

- he defined a **"turning point"** to be the date of the second of two successive declines in y.

To examine this statistic, let Z be an indicator process

$$Z_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_t(\omega) < Y_{t-1}(\omega) < Y_{t-2}(\omega) \geq Y_{t-3}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Then the random variable **time until the next turning point** is defined as the following **stopping time** with respect to Z:

$$W_t(\omega) := \inf\{k \geq 1 \mid Z_{t+k}(\omega) = 1\}$$

Wecker [Wec79] also studied **the minimum value of Y over the next 8 quarters** which can be defined as the random variable.

$$M_t(\omega) := \min\{Y_{t+1}(\omega); Y_{t+2}(\omega); \dots; Y_{t+8}(\omega)\}$$

It is interesting to study yet another possible concept of a **turning point**.

Thus, let

$$T_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_{t-2}(\omega) > Y_{t-1}(\omega) > Y_t(\omega) \text{ and } Y_t(\omega) < Y_{t+1}(\omega) < Y_{t+2}(\omega) \\ -1 & \text{if } Y_{t-2}(\omega) < Y_{t-1}(\omega) < Y_t(\omega) \text{ and } Y_t(\omega) > Y_{t+1}(\omega) > Y_{t+2}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Define a **positive turning point today or tomorrow** statistic as

$$P_t(\omega) := \begin{cases} 1 & \text{if } T_t(\omega) = 1 \text{ or } T_{t+1}(\omega) = 1 \\ 0 & \text{otherwise} \end{cases}$$

This is designed to express the event

- ``after one or two decrease(s), Y will grow for two consecutive quarters"

Following [Wec79], we can use simulations to calculate probabilities of P_t and N_t for each period t.

10.4 A Wecker-Like Algorithm

The procedure consists of the following steps:

- index a sample path by ω_i
- for a given date t, simulate I sample paths of length N

$$Y(\omega_i) = \{Y_{t+1}(\omega_i), Y_{t+2}(\omega_i), \dots, Y_{t+N}(\omega_i)\}_{i=1}^{I}$$

- for each path ω_i, compute the associated value of $W_t(\omega_i), W_{t+1}(\omega_i), \dots$
- consider the sets $\{W_t(\omega_i)\}_{i=1}^{T}$, $\{W_{t+1}(\omega_i)\}_{i=1}^{T}$, $\dots$, $\{W_{t+N}(\omega_i)\}_{i=1}^{T}$ as samples from the predictive distributions $f(W_{t+1} \mid y_t, \dots)$, $f(W_{t+2} \mid y_t, y_{t-1}, \dots)$, ..., $f(W_{t+N} \mid y_t, y_{t-1}, \dots)$.

10.5 Using Simulations to Approximate a Posterior Distribution

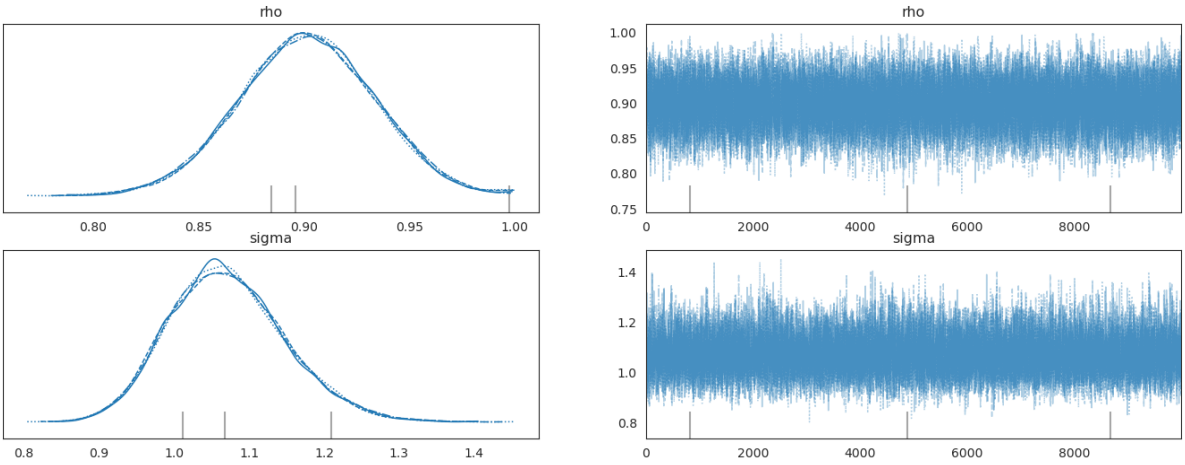
The next code cells use `pymc` to compute the time t posterior distribution of ρ, σ.

Note that in defining the likelihood function, we choose to condition on the initial value y_0.

```
def draw_from_posterior(sample):
    """
    Draw a sample of size N from the posterior distribution.
    """

    AR1_model = pmc.Model()

    with AR1_model:

        # Start with priors
        rho = pmc.Uniform('rho',lower=-1.,upper=1.)  # Assume stable rho
        sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

        # Expected value of y at the next period (rho * y)
        yhat = rho * sample[:-1]

        # Likelihood of the actual realization.
        y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=sample[1:])

    with AR1_model:
        trace = pmc.sample(10000, tune=5000)

    # check condition
    with AR1_model:
        az.plot_trace(trace, figsize=(17, 6))

    rhos = trace.posterior.rho.values.flatten()
    sigmas = trace.posterior.sigma.values.flatten()

    post_sample = {
        'rho': rhos,
        'sigma': sigmas
    }

    return post_sample

post_samples = draw_from_posterior(initial_path)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```



The graphs on the left portray posterior marginal distributions.

10.6 Calculating Sample Path Statistics

Our next step is to prepare Python code to compute our sample path statistics.

```
# define statistics
def next_recession(omega):
    n = omega.shape[0] - 3
    z = np.zeros(n, dtype=int)

    for i in range(n):
        z[i] = int(omega[i] <= omega[i+1] and omega[i+1] > omega[i+2] and omega[i+2] >
 ↪ omega[i+3])

    if np.any(z) == False:
        return 500
    else:
        return np.where(z==1)[0][0] + 1

def minimum_value(omega):
    return min(omega[:8])

def severe_recession(omega):
    z = np.diff(omega)
    n = z.shape[0]

    sr = (z < -.02).astype(int)
    indices = np.where(sr == 1)[0]

    if len(indices) == 0:
        return T1
    else:
        return indices[0] + 1

def next_turning_point(omega):
    """
    Suppose that omega is of length 6
```

(continues on next page)

(continued from previous page)

```
        y_{t-2}, y_{t-1}, y_{t}, y_{t+1}, y_{t+2}, y_{t+3}

    that is sufficient for determining the value of P/N
    """

    n = np.asarray(omega).shape[0] - 4
    T = np.zeros(n, dtype=int)

    for i in range(n):
        if ((omega[i] > omega[i+1]) and (omega[i+1] > omega[i+2]) and
            (omega[i+2] < omega[i+3]) and (omega[i+3] < omega[i+4])):
            T[i] = 1
        elif ((omega[i] < omega[i+1]) and (omega[i+1] < omega[i+2]) and
            (omega[i+2] > omega[i+3]) and (omega[i+3] > omega[i+4])):
            T[i] = -1

    up_turn = np.where(T == 1)[0][0] + 1 if (1 in T) == True else T1
    down_turn = np.where(T == -1)[0][0] + 1 if (-1 in T) == True else T1

    return up_turn, down_turn
```

10.7 Original Wecker Method

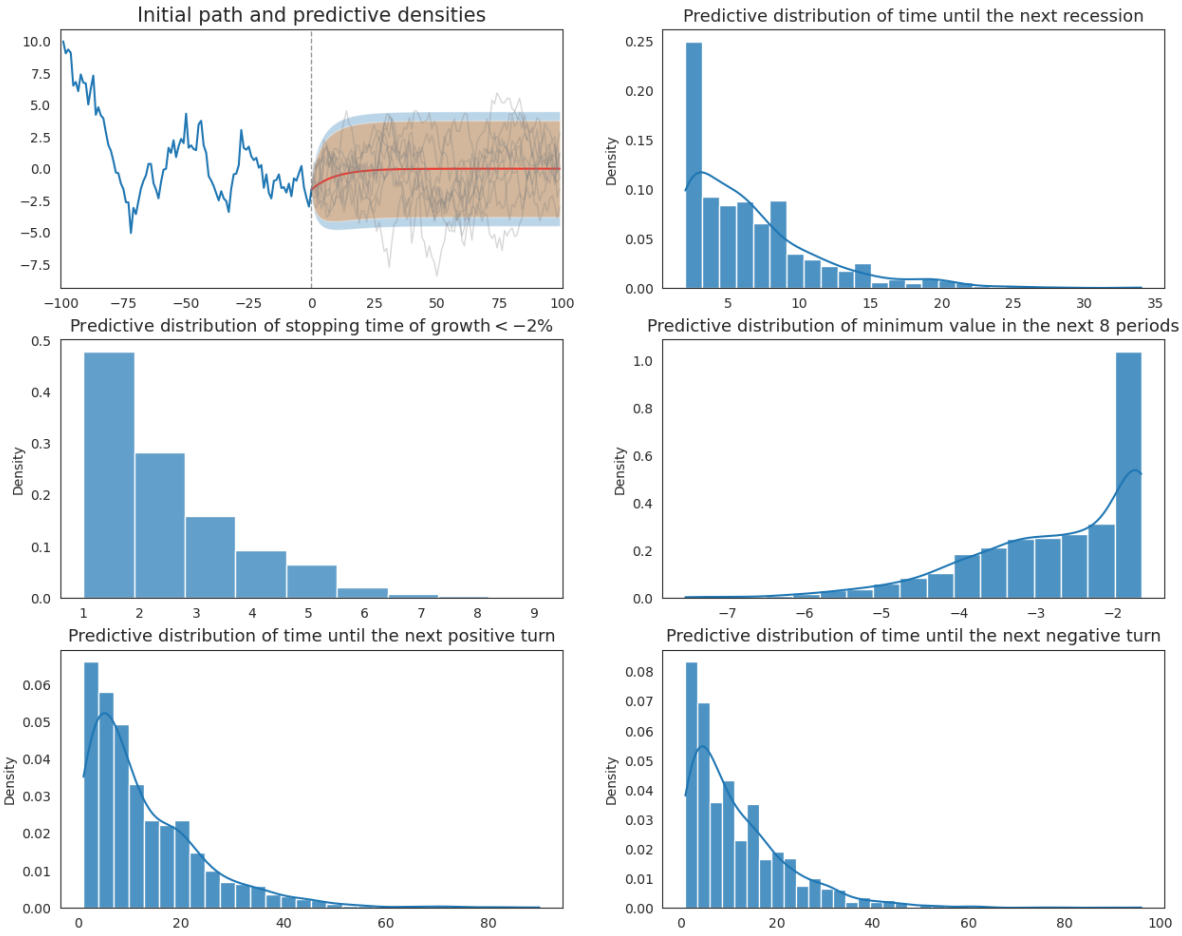
Now we apply Wecker's original method by simulating future paths and compute predictive distributions, conditioning on the true parameters associated with the data-generating model.

```
def plot_Wecker(initial_path, N, ax):
    """
    Plot the predictive distributions from "pure" Wecker's method.
    """
    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)

    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    ax[0, 0].set_title("Initial path and predictive densities", fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Plot 90% CI
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3)
```

(continues on next page)

(continued from previous page)

```
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35)
    ax[0, 0].plot(np.arange(T1), center, color='red', alpha=.7)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho, sigma, initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n%(N/10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.8, label=
↪'True parameters')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=0.9,␣
↪alpha=.7, label='True parameters')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.8, label=
↪'True parameters')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.8, label=
↪'True parameters')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.8,␣
↪label='True parameters')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15,12))
plot_Wecker(initial_path, 1000, ax)
plt.show()
```



10.8 Extended Wecker Method

Now we apply we apply our "extended" Wecker method based on predictive densities of y defined by (10.4) that acknowledge posterior uncertainty in the parameters ρ, σ.

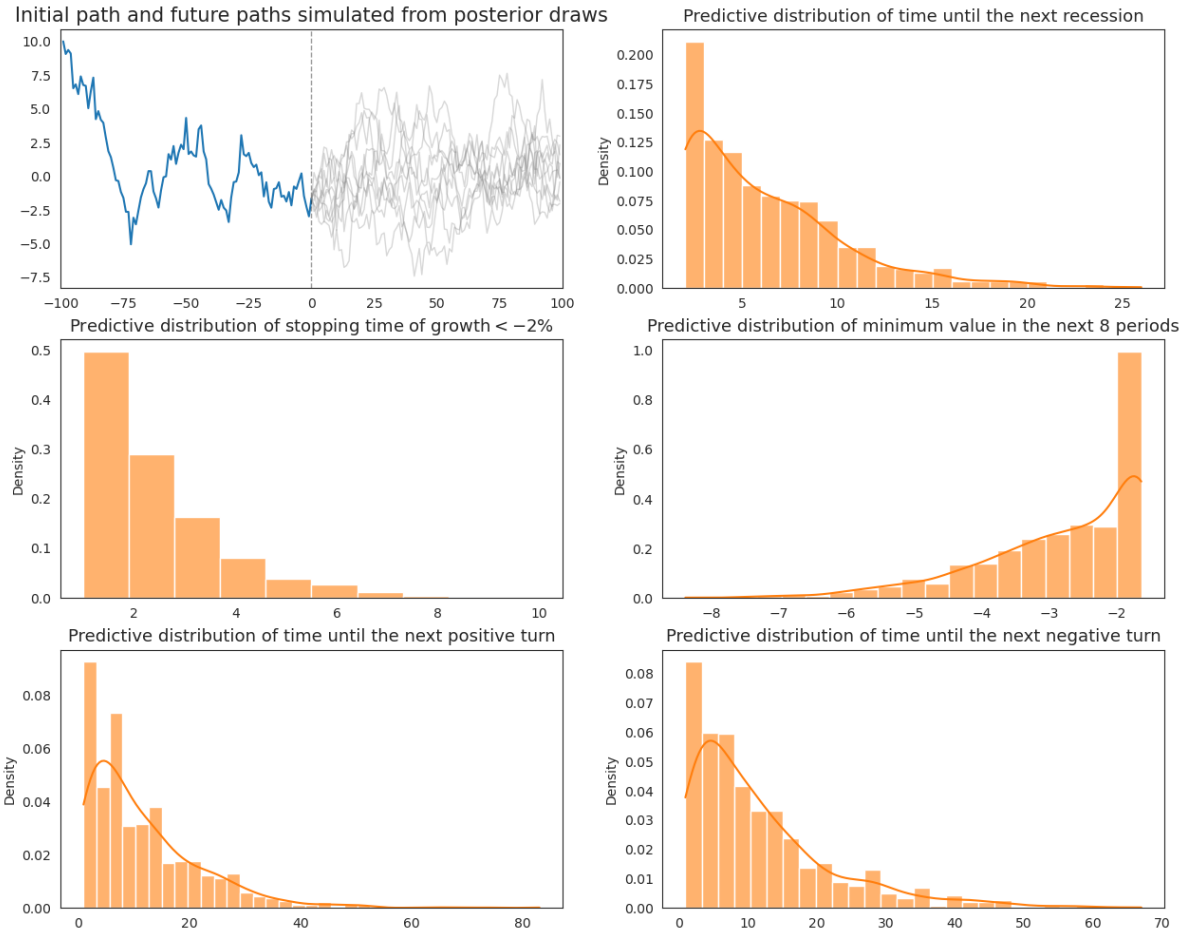
To approximate the intergration on the right side of (10.4), we repeatedly draw parameters from the joint posterior distribution each time we simulate a sequence of future values from model (10.1).

```
def plot_extended_Wecker(post_samples, initial_path, N, ax):
    """
    Plot the extended Wecker's predictive distribution
    """
    # Select a sample
    index = np.random.choice(np.arange(len(post_samples['rho'])), N + 1,␣
↪replace=False)
    rho_sample = post_samples['rho'][index]
    sigma_sample = post_samples['sigma'][index]

    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
```

(continues on next page)

(continued from previous page)

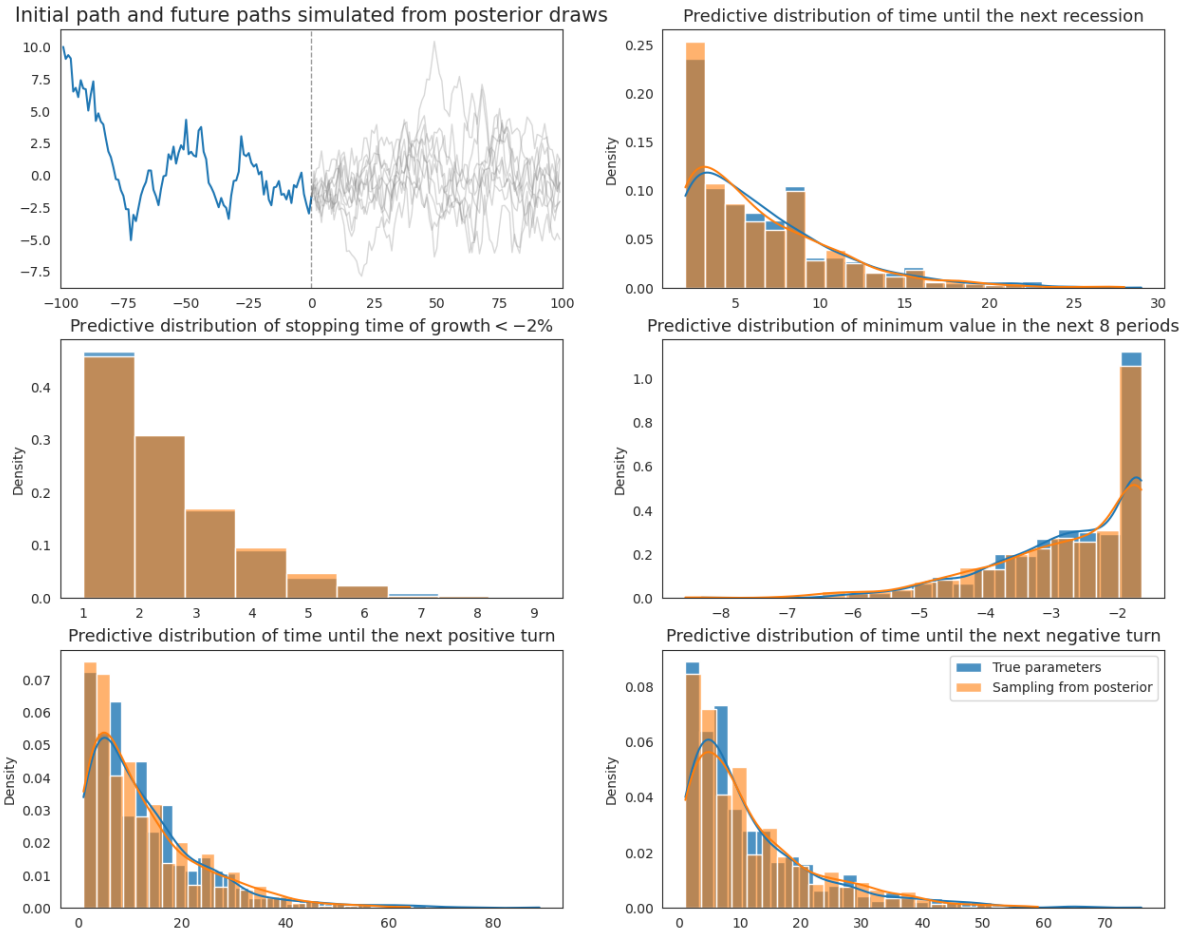
```
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)

    # Plot
    ax[0, 0].set_title("Initial path and future paths simulated from posterior draws",
↪ fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho_sample[n], sigma_sample[n], initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n % (N / 10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=.9,␣
↪alpha=.6, color=colors[1], label='Sampling from posterior')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15, 12))
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.show()
```



10.9 Comparison

Finally, we plot both the original Wecker method and the extended method with parameter values drawn from the posterior together to compare the differences that emerge from pretending to know parameter values when they are actually uncertain.

```
fig, ax = plt.subplots(3, 2, figsize=(15,12))
plot_Wecker(initial_path, 1000, ax)
ax[0, 0].clear()
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.legend()
plt.show()
```


Initial path and future paths simulated from posterior draws
Predictive distribution of time until the next recession
Density
Predictive distribution of stopping time of growth < −2%
Density
Predictive distribution of minimum value in the next 8 periods
Density
Predictive distribution of time until the next positive turn
Density
Predictive distribution of time until the next negative turn
Density
True parameters
Sampling from posterior

CHAPTER

ELEVEN

LIKELIHOOD RATIO PROCESSES

Contents

11.1 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll use a setting described in *this lecture*.

Among things that we'll learn are

- A peculiar property of likelihood ratio processes
- How a likelihood ratio process is a key ingredient in frequentist hypothesis testing
- How a **receiver operator characteristic curve** summarizes information about a false alarm probability and power in frequentist hypothesis testing
- How during World War II the United States Navy devised a decision rule that Captain Garret L. Schyler challenged and asked Milton Friedman to justify to him, a topic to be studied in *this lecture*

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
```

(continues on next page)

(continued from previous page)

```
from math import gamma
from scipy.integrate import quad
```

11.2 Likelihood Ratio Process

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion* or *multiplicative decomposition*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [NP33].

To help us appreciate how things work, the following Python code evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributionss, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

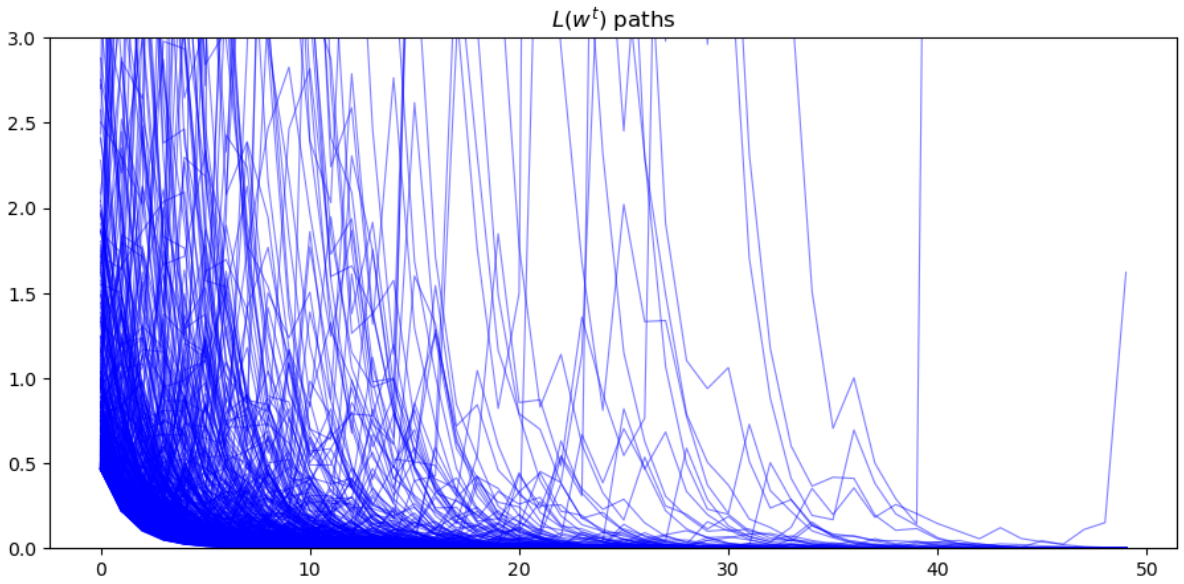
```
@njit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

11.3 Nature Permanently Draws from Density g

We first simulate the likelihood ratio process when nature permanently draws from g.

```
l_arr_g = simulate(G_a, G_b)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
N, T = l_arr_g.shape

for i in range(N):

    plt.plot(range(T), l_seq_g[i, :], color='b', lw=0.8, alpha=0.5)

plt.ylim([0, 3])
plt.title("$L(w^{t})$ paths");
```

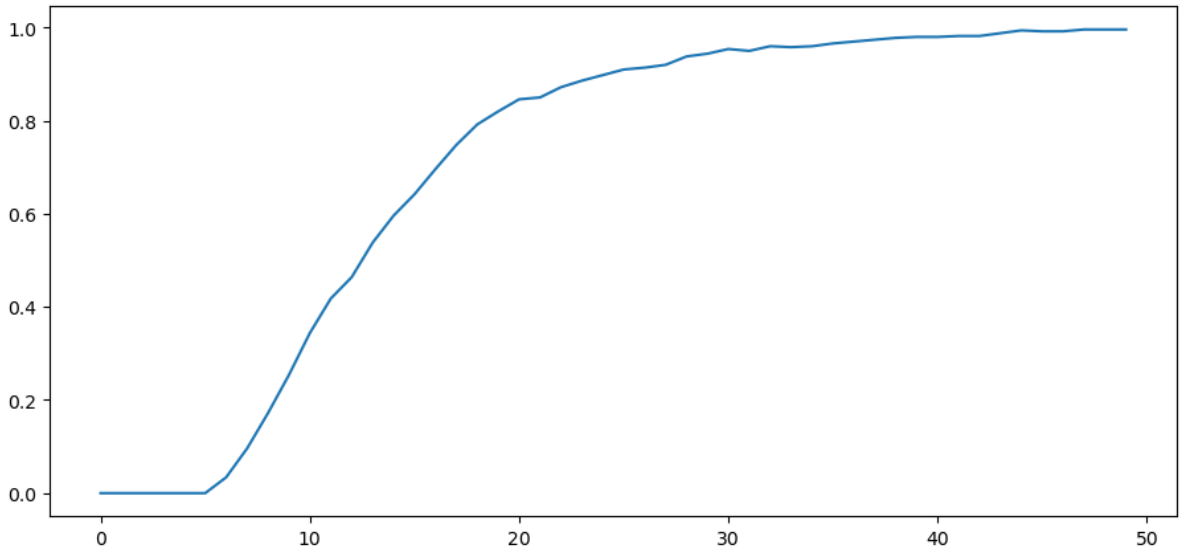


Evidently, as sample length T grows, most probability mass shifts toward zero

To see it this more clearly clearly, we plot over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

```
plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f2e14276650>]
```



Despite the evident convergence of most probability mass to a very small interval near 0, the unconditional mean of $L\left(w^t\right)$ under probability density g is identically 1 for all t.

To verify this assertion, first notice that as mentioned earlier the unconditional mean $E\left[\ell\left(w_t\right) \mid q=g\right]$ is 1 for all t:

$$\begin{aligned} E\left[\ell\left(w_t\right) \mid q=g\right] &= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} g\left(w_t\right) dw_t \\ &= \int f\left(w_t\right) dw_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E\left[L\left(w^1\right) \mid q=g\right] &= E\left[\ell\left(w_1\right) \mid q=g\right] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t) L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E\left[L\left(w^t\right) \mid q=g\right] &= E\left[L\left(w^{t-1}\right) \ell\left(w_t\right) \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g, w^{t-1}\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) E\left[\ell\left(w_t\right) \mid q=g\right] \mid q=g\right] \\ &= E\left[L\left(w^{t-1}\right) \mid q=g\right] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E\left[L\left(w^t\right) \mid q=g\right] = 1$ for all $t \geq 1$.

11.4 Peculiar Property

How can $E\left[L\left(w^t\right) \mid q=g\right] = 1$ possibly be true when most probability mass of the likelihood ratio process is piling up near 0 as $t \to +\infty$?

The answer has to be that as $t \to +\infty$, the distribution of L_t becomes more and more fat-tailed: enough mass shifts to larger and larger values of L_t to make the mean of L_t continue to be one despite most of the probability mass piling up near 0.

To illustrate this peculiar property, we simulate many paths and calculate the unconditional mean of $L\left(w^t\right)$ by averaging across these many paths at each t.

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

It would be useful to use simulations to verify that unconditional means $E\left[L\left(w^t\right)\right]$ equal unity by averaging across sample paths.

But it would be too computer-time-consuming for us to that here simply by applying a standard Monte Carlo simulation approach.

The reason is that the distribution of $L\left(w^t\right)$ is extremely skewed for large values of t.

Because the probability density in the right tail is close to 0, it just takes too much computer time to sample enough points from the right tail.

We explain the problem in more detail in *this lecture*.

There we describe a way to an alternative way to compute the mean of a likelihood ratio by computing the mean of a *different* random variable by sampling from a *different* probability distribution.

11.5 Nature Permanently Draws from Density f

Now suppose that before time 0 nature permanently decided to draw repeatedly from density f.

While the mean of the likelihood ratio $\ell(w_t)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$\begin{aligned} E[\ell(w_t) \mid q = f] &= \int \frac{f(w_t)}{g(w_t)} f(w_t)\, dw_t \\ &= \int \frac{f(w_t)}{g(w_t)} \frac{f(w_t)}{g(w_t)} g(w_t)\, dw_t \\ &= \int \ell(w_t)^2 g(w_t)\, dw_t \\ &= E\left[\ell(w_t)^2 \mid q = g\right] \\ &= E[\ell(w_t) \mid q = g]^2 + Var(\ell(w_t) \mid q = g) \\ &> E[\ell(w_t) \mid q = g]^2 = 1 \end{aligned}$$

This in turn implies that the unconditional mean of the likelihood ratio process $L(w^t)$ diverges toward $+\infty$.

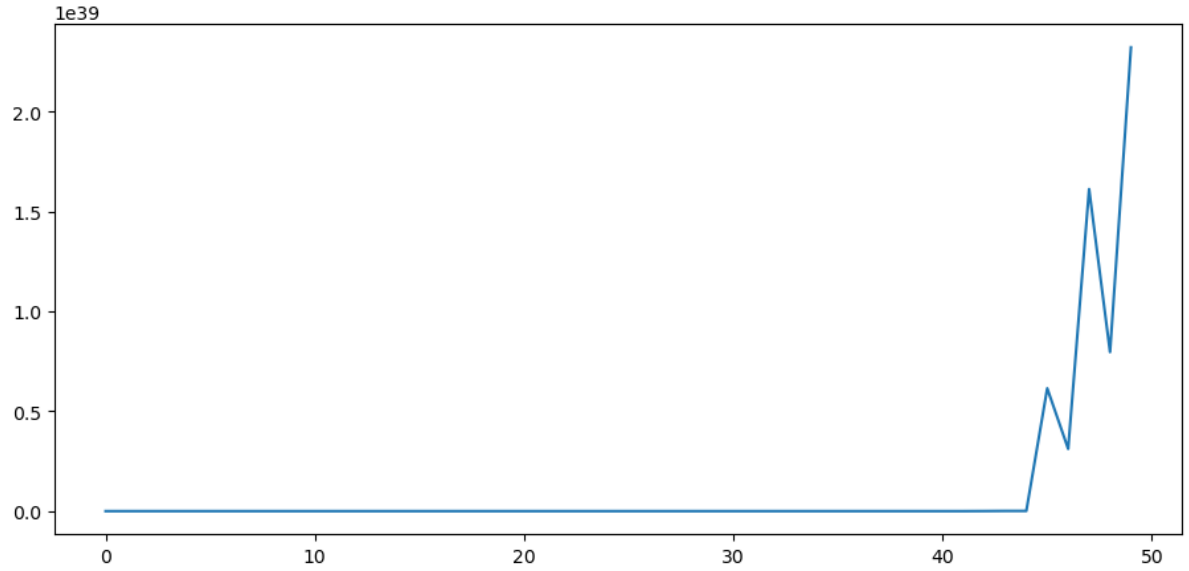
Simulations below confirm this conclusion.

Please note the scale of the y axis.

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

```
N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
```

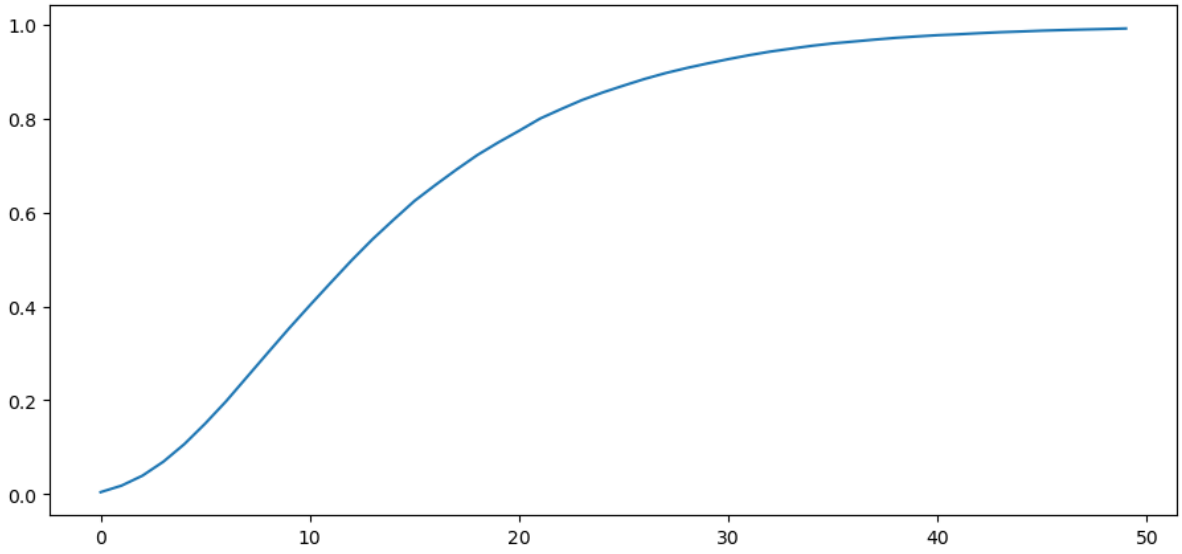
```
[<matplotlib.lines.Line2D at 0x7f2e140331d0>]
```



We also plot the probability that $L(w^t)$ falls into the interval $[10000, \infty)$ as a function of time and watch how fast probability mass diverges to $+\infty$.

```
plt.plot(range(T), np.sum(l_seq_f > 10000, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f2e13d88650>]
```



11.6 Likelihood Ratio Test

We now describe how to employ the machinery of Neyman and Pearson [NP33] to test the hypothesis that history w^t is generated by repeated IID draws from density g.

Denote q as the data generating process, so that $q = f$ or g.

Upon observing a sample $\{W_i\}_{i=1}^t$, we want to decide whether nature is drawing from g or from f by performing a (frequentist) hypothesis test.

We specify

- Null hypothesis H_0: $q = f$,
- Alternative hypothesis H_1: $q = g$.

Neyman and Pearson proved that the best way to test this hypothesis is to use a **likelihood ratio test** that takes the form:

- reject H_0 if $L(W^t) < c$,
- accept H_0 otherwise.

where c is a given discrimination threshold, to be chosen in a way we'll soon describe.

This test is *best* in the sense that it is a **uniformly most powerful** test.

To understand what this means, we have to define probabilities of two important events that allow us to characterize a test associated with a given threshold c.

The two probabilities are:

- Probability of detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr\{L(w^t) < c \mid q = g\}$$

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr\{L(w^t) < c \mid q = f\}$$

The Neyman-Pearson Lemma states that among all possible tests, a likelihood ratio test maximizes the probability of detection for a given probability of false alarm.

Another way to say the same thing is that among all possible tests, a likelihood ratio test maximizes **power** for a given **significance level**.

To have made a good inference, we want a small probability of false alarm and a large probability of detection.

With sample size t fixed, we can change our two probabilities by adjusting c.

A troublesome "that's life" fact is that these two probabilities move in the same direction as we vary the critical value c.

Without specifying quantitative losses from making Type I and Type II errors, there is little that we can say about how we *should* trade off probabilities of the two types of mistakes.

We do know that increasing sample size t improves statistical inference.

Below we plot some informative figures that illustrate this.

We also present a classical frequentist method for choosing a sample size t.

Let's start with a case in which we fix the threshold c at 1.

```
c = 1
```

Below we plot empirical distributions of logarithms of the cumulative likelihood ratios simulated above, which are generated by either f or g.

Taking logarithms has no effect on calculating the probabilities because the log is a monotonic transformation.

As t increases, the probabilities of making Type I and Type II errors both decrease, which is good.

This is because most of the probability mass of $\log(L(w^t))$ moves toward $-\infty$ when g is the data generating process, ; while $\log(L(w^t))$ goes to ∞ when data are generated by f.

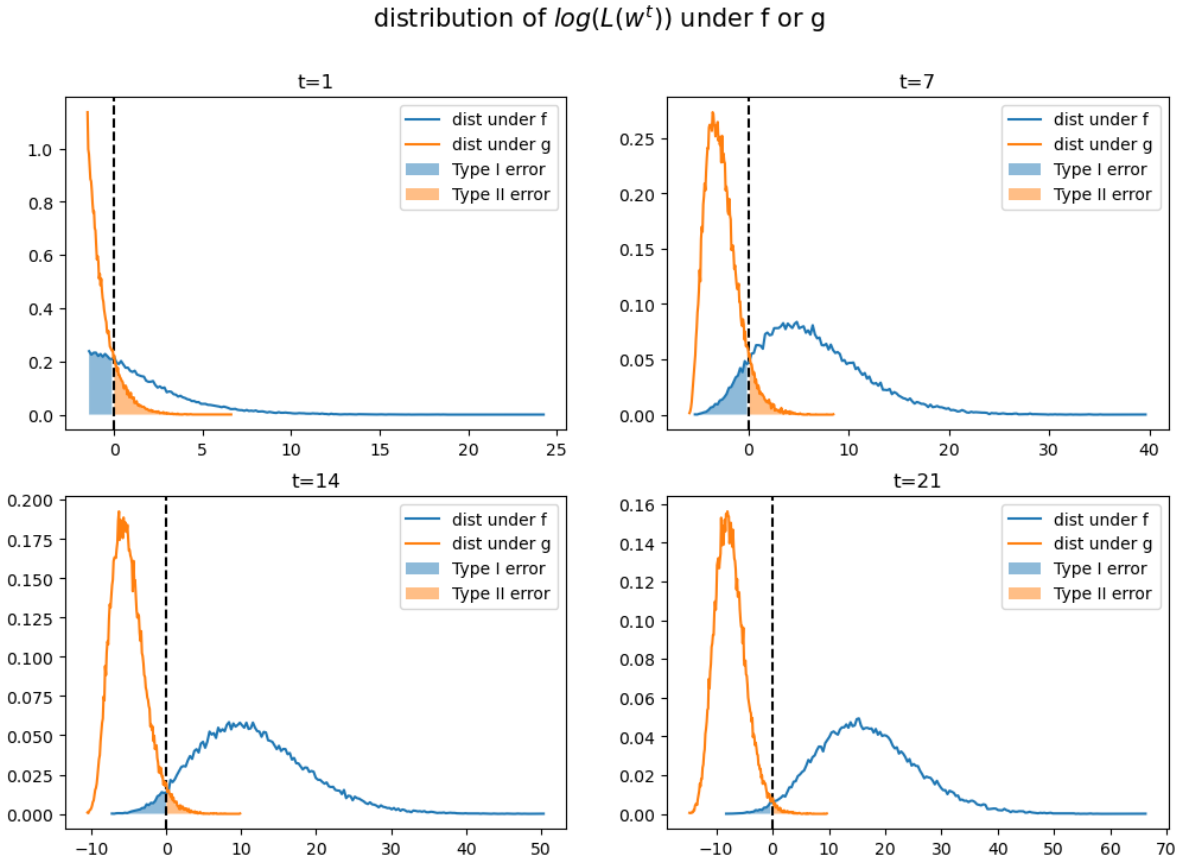
That disparate behavior of $\log(L(w^t))$ under f and q is what makes it possible to distinguish $q = f$ from $q = g$.

```
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle('distribution of $log(L(w^t))$ under f or g', fontsize=15)

for i, t in enumerate([1, 7, 14, 21]):
    nr = i // 2
    nc = i % 2

    axs[nr, nc].axvline(np.log(c), color="k", ls="--")

    hist_f, x_f = np.histogram(np.log(l_seq_f[:, t]), 200, density=True)
    hist_g, x_g = np.histogram(np.log(l_seq_g[:, t]), 200, density=True)

    axs[nr, nc].plot(x_f[1:], hist_f, label="dist under f")
    axs[nr, nc].plot(x_g[1:], hist_g, label="dist under g")

    for i, (x, hist, label) in enumerate(zip([x_f, x_g], [hist_f, hist_g], ["Type I␣
↪error", "Type II error"])):
        ind = x[1:] <= np.log(c) if i == 0 else x[1:] > np.log(c)
        axs[nr, nc].fill_between(x[1:][ind], hist[ind], alpha=0.5, label=label)
```

(continues on next page)

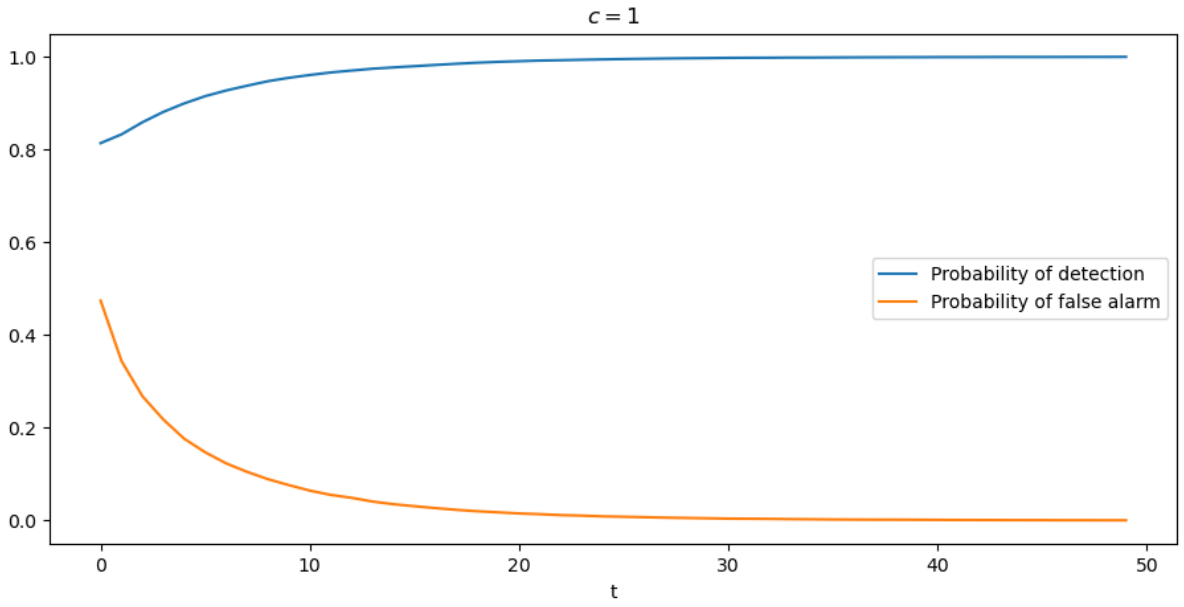
(continued from previous page)

```
    axs[nr, nc].legend()
    axs[nr, nc].set_title(f"t={t}")

plt.show()
```



The graph below shows more clearly that, when we hold the threshold c fixed, the probability of detection monotonically increases with increases in t and that the probability of a false alarm monotonically decreases.

```
PD = np.empty(T)
PFA = np.empty(T)

for t in range(T):
    PD[t] = np.sum(l_seq_g[:, t] < c) / N
    PFA[t] = np.sum(l_seq_f[:, t] < c) / N

plt.plot(range(T), PD, label="Probability of detection")
plt.plot(range(T), PFA, label="Probability of false alarm")
plt.xlabel("t")
plt.title("$c=1$")
plt.legend()
plt.show()
```



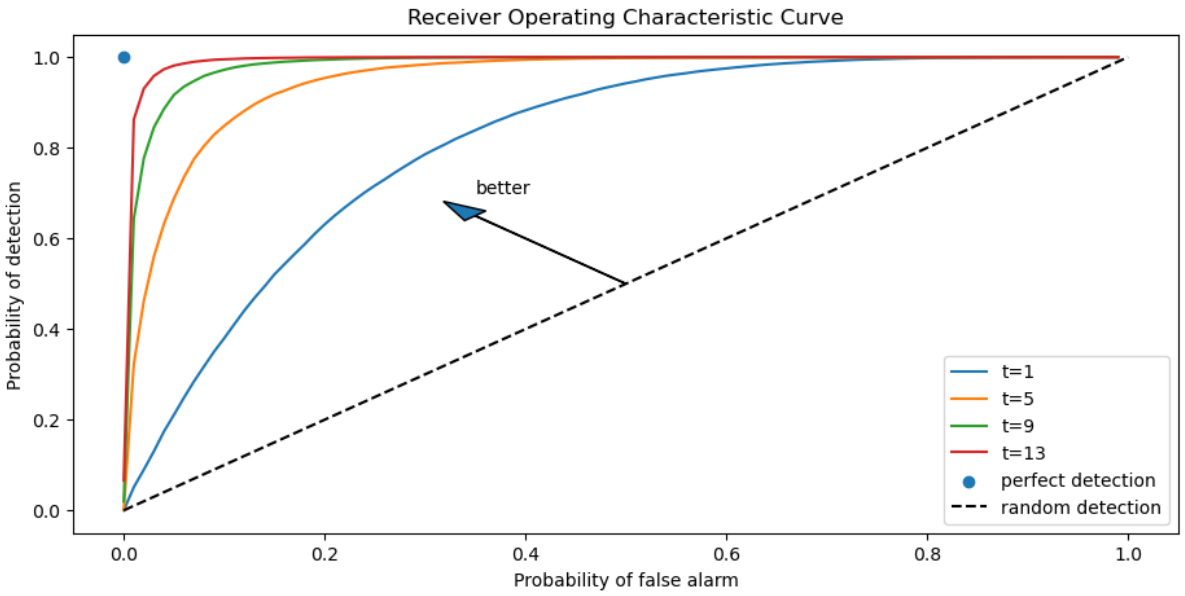
For a given sample size t, the threshold c uniquely pins down probabilities of both types of error.

If for a fixed t we now free up and move c, we will sweep out the probability of detection as a function of the probability of false alarm.

This produces what is called a receiver operating characteristic curve.

Below, we plot receiver operating characteristic curves for different sample sizes t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(l_seq_f[:, t], PFA)
    PD = [np.sum(l_seq_g[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
plt.show()
```



Notice that as t increases, we are assured a larger probability of detection and a smaller probability of false alarm associated with a given discrimination threshold c.

As $t \to +\infty$, we approach the perfect detection curve that is indicated by a right angle hinging on the blue dot.

For a given sample size t, the discrimination threshold c determines a point on the receiver operating characteristic curve.

It is up to the test designer to trade off probabilities of making the two types of errors.

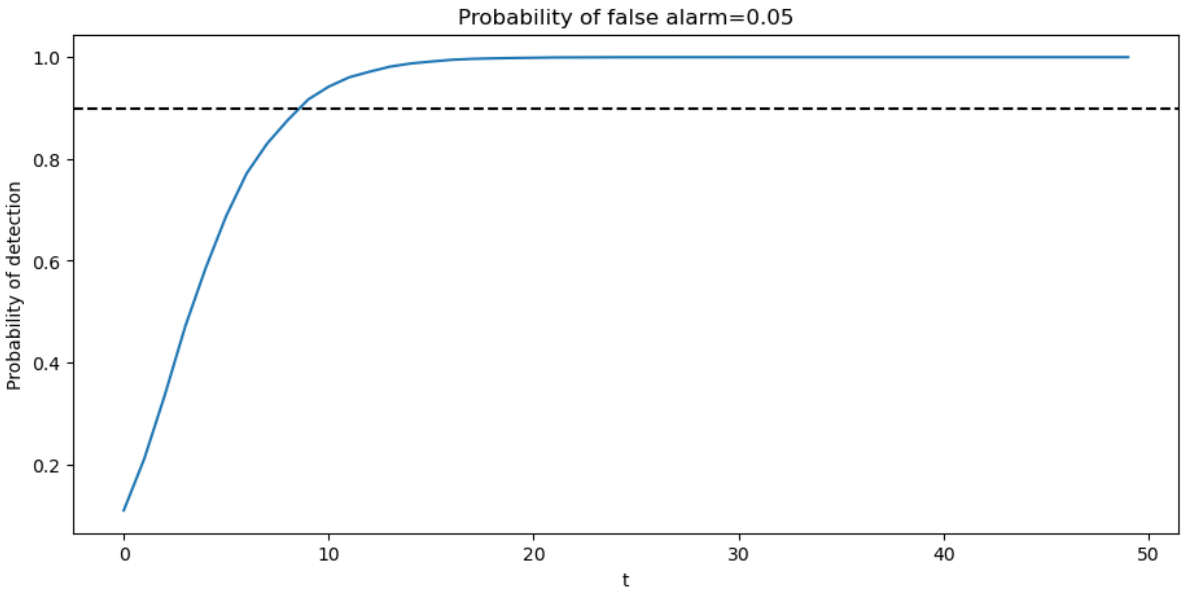
But we know how to choose the smallest sample size to achieve given targets for the probabilities.

Typically, frequentists aim for a high probability of detection that respects an upper bound on the probability of false alarm.

Below we show an example in which we fix the probability of false alarm at 0.05.

The required sample size for making a decision is then determined by a target probability of detection, for example, 0.9, as depicted in the following graph.

```
PFA = 0.05
PD = np.empty(T)

for t in range(T):

    c = np.percentile(l_seq_f[:, t], PFA * 100)
    PD[t] = np.sum(l_seq_g[:, t] < c) / N

plt.plot(range(T), PD)
plt.axhline(0.9, color="k", ls="--")

plt.xlabel("t")
plt.ylabel("Probability of detection")
plt.title(f"Probability of false alarm={PFA}")
plt.show()
```



The United States Navy evidently used a procedure like this to select a sample size t for doing quality control tests during World War II.

A Navy Captain who had been ordered to perform tests of this kind had doubts about it that he presented to Milton Friedman, as we describe in *this lecture*.

11.7 Kullback–Leibler Divergence

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's watch how how the cumulated likelihood ratios f/g behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence**.

It is also called **relative entropy**.

It measures how one probability distribution differs from another.

In our application, we want to measure how f or g diverges from h

The two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$\begin{aligned} K_f &= E_h \left[\log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} \right] \\ &= \int \log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} h(w) \, dw \\ &= \int \log \left(\frac{f(w)}{h(w)} \right) f(w) \, dw \end{aligned}$$

$$
\begin{aligned}
K_g &= E_h \left[\log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} \right] \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} h(w) \, dw \\
&= \int \log \left(\frac{g(w)}{h(w)} \right) g(w) \, dw
\end{aligned}
$$

When $K_g < K_f$, g is closer to h than f is.

- In that case we'll find that $L(w^t) \to 0$.

When $K_g > K_f$, f is closer to h than g is.

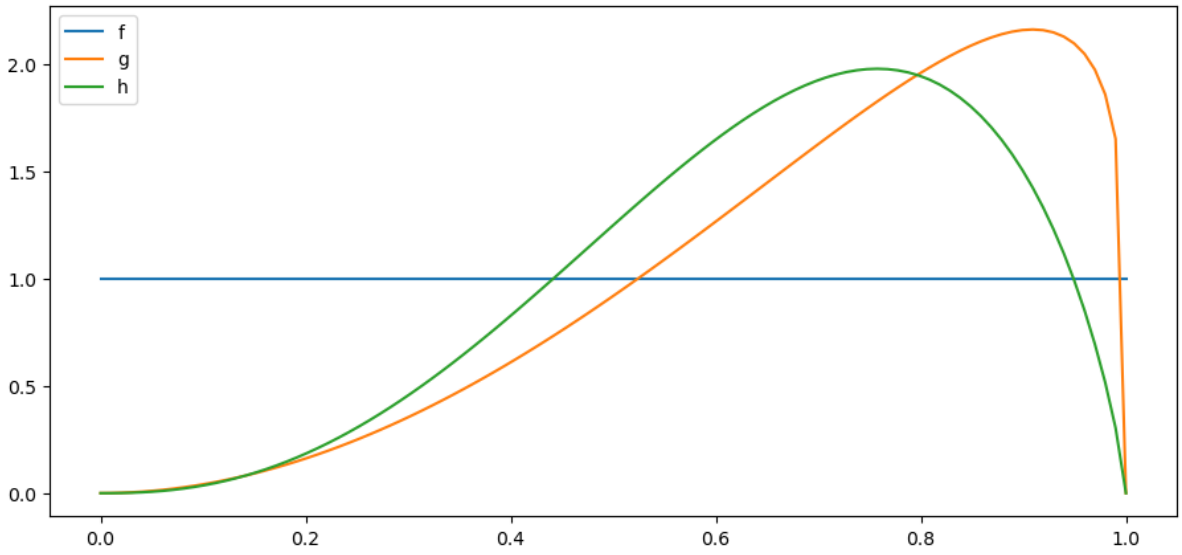
- In that case we'll find that $L(w^t) \to +\infty$

We'll now experiment with an h is also a beta distribution

We'll start by setting parameters G_a and G_b so that h is closer to g

```
H_a, H_b = 3.5, 1.8

h = njit(lambda x: p(x, H_a, H_b))
```

```
x_range = np.linspace(0, 1, 100)
plt.plot(x_range, f(x_range), label='f')
plt.plot(x_range, g(x_range), label='g')
plt.plot(x_range, h(x_range), label='h')

plt.legend()
plt.show()
```



Let's compute the Kullback–Leibler discrepancies by quadrature integration.

```
def KL_integrand(w, q, h):

    m = q(w) / h(w)

    return np.log(m) * q(w)
```

```
def compute_KL(h, f, g):

    Kf, _ = quad(KL_integrand, 0, 1, args=(f, h))
    Kg, _ = quad(KL_integrand, 0, 1, args=(g, h))

    return Kf, Kg
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

```
(0.7902536603660161, 0.08554075759988769)
```

We have $K_g < K_f$.

Next, we can verify our conjecture about $L\left(w^t\right)$ by simulation.

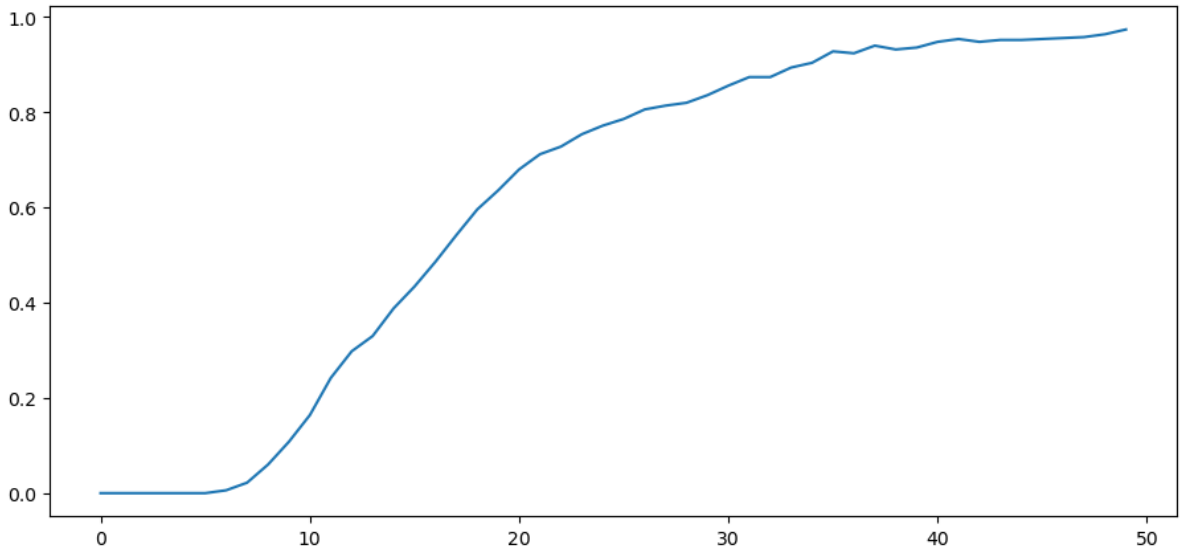
```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

The figure below plots over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

Notice that it converges to 1 as expected when g is closer to h than f is.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h <= 0.01, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f2e13bbb1d0>]
```



We can also try an h that is closer to f than is g so that now K_g is larger than K_f.

```
H_a, H_b = 1.2, 1.2
h = njit(lambda x: p(x, H_a, H_b))
```

```
Kf, Kg = compute_KL(h, f, g)
Kf, Kg
```

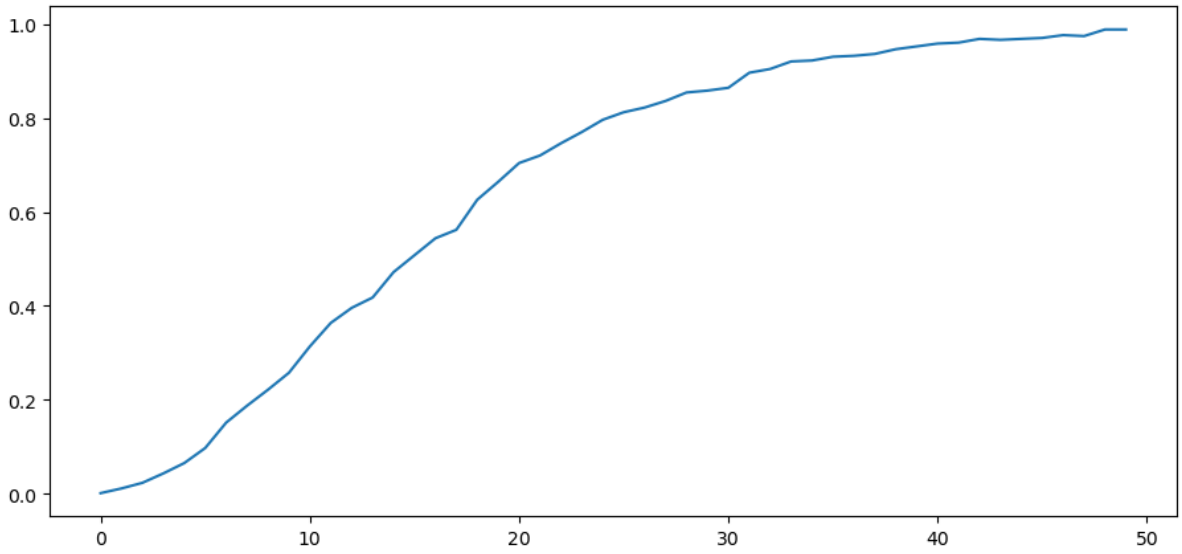
```
(0.01239249754452668, 0.35377684280997646)
```

```
l_arr_h = simulate(H_a, H_b)
l_seq_h = np.cumprod(l_arr_h, axis=1)
```

Now probability mass of $L\left(w^t\right)$ falling above 10000 diverges to $+\infty$.

```
N, T = l_arr_h.shape
plt.plot(range(T), np.sum(l_seq_h > 10000, axis=0) / N)
```

```
[<matplotlib.lines.Line2D at 0x7f2e10116790>]
```



11.8 Sequels

Likelihood processes play an important role in Bayesian learning, as described in *this lecture* and as applied in this lecture.

Likelihood ratio processes appear again in this lecture, which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

CHAPTER

TWELVE

COMPUTING MEAN OF A LIKELIHOOD RATIO PROCESS

Contents

12.1 Overview

In *this lecture* we described a peculiar property of a likelihood ratio process, namely, that it's mean equals one for all $t \geq 0$ despite it's converging to zero almost surely.

While it is easy to verify that peculiar properly analytically (i.e., in population), it is challenging to use a computer simulation to verify it via an application of a law of large numbers that entails studying sample averages of repeated simulations.

To confront this challenge, this lecture puts **importance sampling** to work to accelerate convergence of sample averages to population means.

We use importance sampling to estimate the mean of a cumulative likelihood ratio $L\left(\omega^t\right) = \prod_{i=1}^{t} \ell\left(\omega_i\right)$.

We start by importing some Python packages.

```
import numpy as np
from numba import njit, vectorize, prange
import matplotlib.pyplot as plt
from math import gamma
```

12.2 Mathematical Expectation of Likelihood Ratio

In *this lecture*, we studied a likelihood ratio $\ell(\omega_t)$

$$\ell(\omega_t) = \frac{f(\omega_t)}{g(\omega_t)}$$

where f and g are densities for Beta distributions with parameters F_a, F_b, G_a, G_b.

Assume that an i.i.d. random variable $\omega_t \in \Omega$ is generated by g.

The **cumulative likelihood ratio** $L(\omega^t)$ is

$$L(\omega^t) = \prod_{i=1}^{t} \ell(\omega_i)$$

Our goal is to approximate the mathematical expectation $E[L(\omega^t)]$ well.

In *this lecture*, we showed that $E[L(\omega^t)]$ equals 1 for all t. We want to check out how well this holds if we replace E by with sample averages from simulations.

This turns out to be easier said than done because for Beta distributions assumed above, $L(\omega^t)$ has a very skewed distribution with a very long tail as $t \to \infty$.

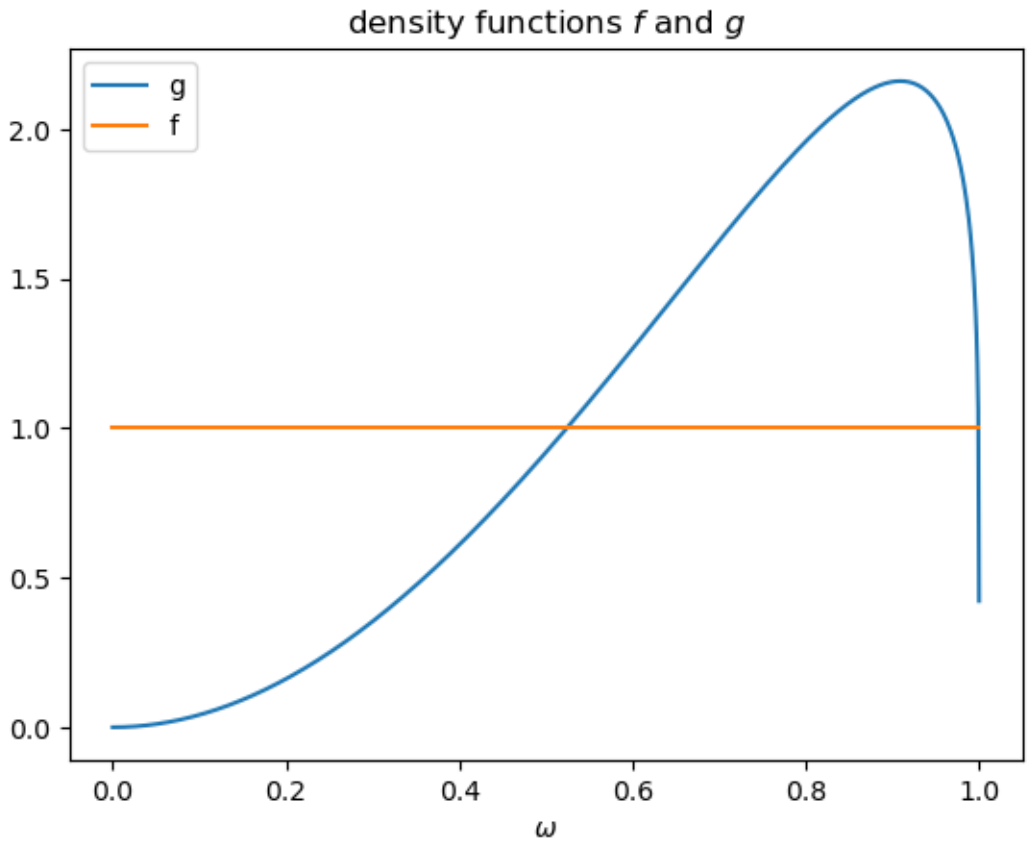
This property makes it difficult efficiently and accurately to estimate the mean by standard Monte Carlo simulation methods.

In this lecture we explore how a standard Monte Carlo method fails and how **importance sampling** provides a more computationally efficient way to approximate the mean of the cumulative likelihood ratio.

We first take a look at the density functions `f` and `g` .

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(w, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * w ** (a-1) * (1 - w) ** (b-1)

# The two density functions.
f = njit(lambda w: p(w, F_a, F_b))
g = njit(lambda w: p(w, G_a, G_b))
```

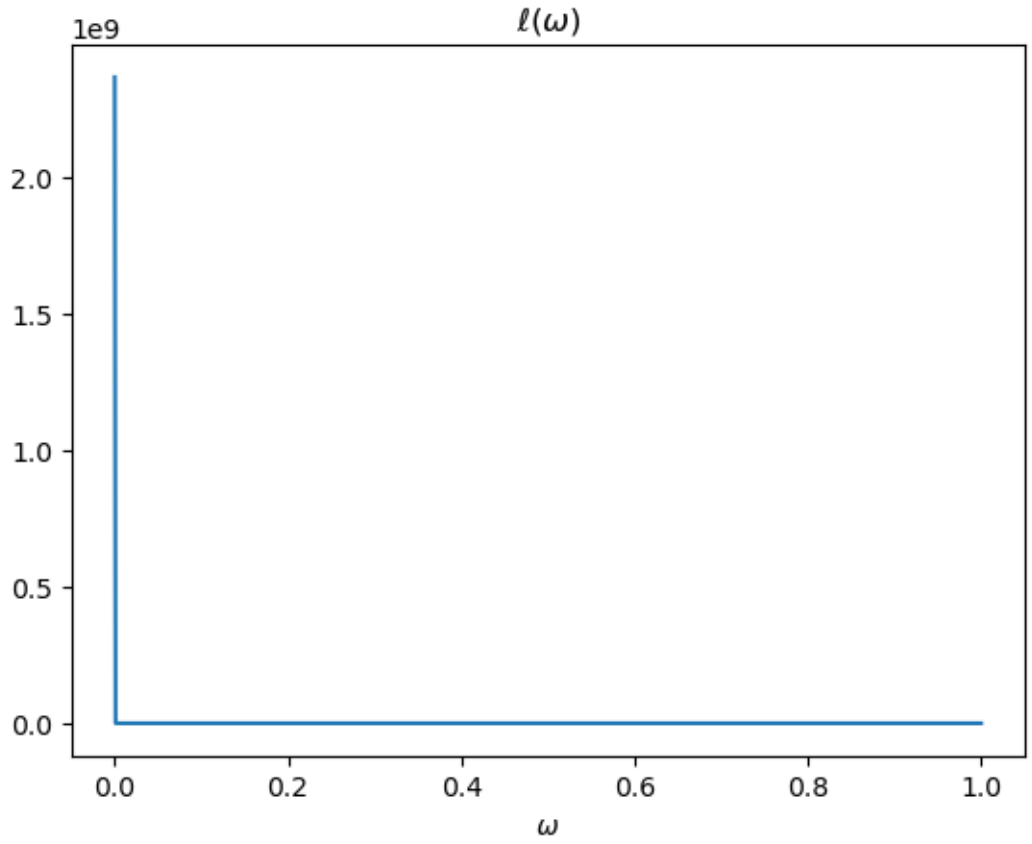
```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label='g')
plt.plot(w_range, f(w_range), label='f')
plt.xlabel('$\omega$')
plt.legend()
plt.title('density functions $f$ and $g$')
plt.show()
```



The likelihood ratio is `l(w)=f(w)/g(w)`.

```
l = njit(lambda w: f(w) / g(w))
```

```
plt.plot(w_range, l(w_range))
plt.title('$\ell(\omega)$')
plt.xlabel('$\omega$')
plt.show()
```



The above plots shows that as $\omega \to 0$, $f(\omega)$ is unchanged and $g(\omega) \to 0$, so the likelihood ratio approaches infinity.

A Monte Carlo approximation of $\hat{E}[L(\omega^t)] = \hat{E}\left[\prod_{i=1}^t \ell(\omega_i)\right]$ would repeatedly draw ω from g, calculate the likelihood ratio $\ell(\omega) = \frac{f(\omega)}{g(\omega)}$ for each draw, then average these over all draws.

Because $g(\omega) \to 0$ as $\omega \to 0$, such a simulation procedure undersamples a part of the sample space $[0, 1]$ that it is important to visit often in order to do a good job of approximating the mathematical expectation of the likelihood ratio $\ell(\omega)$.

We illustrate this numerically below.

12.3 Importance sampling

We circumvent the issue by using a *change of distribution* called **importance sampling**.

Instead of drawing from g to generate data during the simulation, we use an alternative distribution h to generate draws of ω.

The idea is to design h so that it oversamples the region of Ω where $\ell(\omega_t)$ has large values but low density under g.

After we construct a sample in this way, we must then weight each realization by the likelihood ratio of g and h when we compute the empirical mean of the likelihood ratio.

By doing this, we properly account for the fact that we are using h and not g to simulate data.

To illustrate, suppose were interested in $E[\ell(\omega)]$.

We could simply compute:

$$\hat{E}^g\left[\ell\left(\omega\right)\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^g)$$

where ω_i^g indicates that ω_i is drawn from g.

But using our insight from importance sampling, we could instead calculate the object:

$$\hat{E}^h\left[\ell\left(\omega\right)\frac{g(w)}{h(w)}\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^h)\frac{g(w_i^h)}{h(w_i^h)}$$

where w_i is now drawn from importance distribution h.

Notice that the above two are exactly the same population objects:

$$E^g\left[\ell\left(\omega\right)\right] = \int_\Omega \ell(\omega)g(\omega)d\omega = \int_\Omega \ell(\omega)\frac{g(\omega)}{h(\omega)}h(\omega)d\omega = E^h\left[\ell\left(\omega\right)\frac{g(\omega)}{h(\omega)}\right]$$

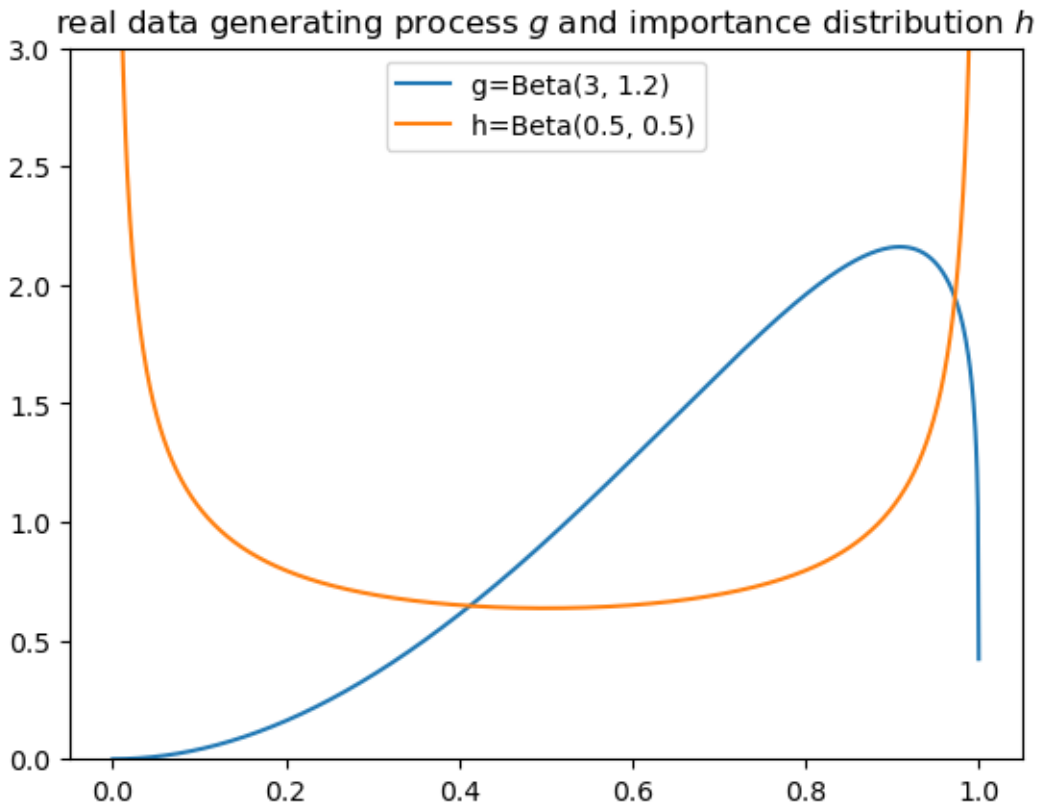
12.4 Selecting a Sampling Distribution

Since we must use an h that has larger mass in parts of the distribution to which g puts low mass, we use $h = Beta(0.5, 0.5)$ as our importance distribution.

The plots compare g and h.

```
g_a, g_b = G_a, G_b
h_a, h_b = 0.5, 0.5
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, 0.5, 0.5), label=f'h=Beta({h_a}, {h_b})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



12.5 Approximating a cumulative likelihood ratio

We now study how to use importance sampling to approximate $E\left[L(\omega^t)\right] = \left[\prod_{i=1}^{T} \ell\left(\omega_i\right)\right]$.

As above, our plan is to draw sequences ω^t from q and then re-weight the likelihood ratio appropriately:

$$\hat{E}^p\left[L\left(\omega^t\right)\right] = \hat{E}^p\left[\prod_{t=1}^{T} \ell\left(\omega_t\right)\right] = \hat{E}^q\left[\prod_{t=1}^{T} \ell\left(\omega_t\right) \frac{p\left(\omega_t\right)}{q\left(\omega_t\right)}\right] = \frac{1}{N}\sum_{i=1}^{N}\left(\prod_{t=1}^{T} \ell(\omega_{i,t}^h) \frac{p\left(\omega_{i,t}^h\right)}{q\left(\omega_{i,t}^h\right)}\right)$$

where the last equality uses $\omega_{i,t}^h$ drawn from the importance distribution q.

Here $\frac{p\left(\omega_{i,t}^q\right)}{q\left(\omega_{i,t}^q\right)}$ is the weight we assign to each data point $\omega_{i,t}^q$.

Below we prepare a Python function for computing the importance sampling estimates given any beta distributions p, q.

```
@njit(parallel=True)
def estimate(p_a, p_b, q_a, q_b, T=1, N=10000):

    μ_L = 0
    for i in prange(N):

        L = 1
        weight = 1
        for t in range(T):
            w = np.random.beta(q_a, q_b)
```

(continues on next page)

(continued from previous page)

```
            l = f(w) / g(w)

            L *= l
            weight *= p(w, p_a, p_b) / p(w, q_a, q_b)

        μ_L += L * weight

    μ_L /= N

    return μ_L
```

Consider the case when $T = 1$, which amounts to approximating $E_0 [\ell(\omega)]$

For the standard Monte Carlo estimate, we can set $p = g$ and $q = g$.

```
estimate(g_a, g_b, g_a, g_b, T=1, N=10000)
```

```
0.9884940566454353
```

For our importance sampling estimate, we set $q = h$.

```
estimate(g_a, g_b, h_a, h_b, T=1, N=10000)
```

```
0.9956184668220487
```

Evidently, even at T=1, our importance sampling estimate is closer to 1 than is the Monte Carlo estimate.

Bigger differences arise when computing expectations over longer sequences, $E_0 [L(\omega^t)]$.

Setting $T = 10$, we find that the Monte Carlo method severely underestimates the mean while importance sampling still produces an estimate close to its theoretical value of unity.

```
estimate(g_a, g_b, g_a, g_b, T=10, N=10000)
```

```
2.799078852376603
```

```
estimate(g_a, g_b, h_a, h_b, T=10, N=10000)
```

```
1.0029913559520476
```

12.6 Distribution of Sample Mean

We next study the bias and efficiency of the Monte Carlo and importance sampling approaches.

The code below produces distributions of estimates using both Monte Carlo and importance sampling methods.

```
@njit(parallel=True)
def simulate(p_a, p_b, q_a, q_b, N_simu, T=1):
```

(continues on next page)

(continued from previous page)

```
    μ_L_p = np.empty(N_simu)
    μ_L_q = np.empty(N_simu)

    for i in prange(N_simu):
        μ_L_p[i] = estimate(p_a, p_b, p_a, p_b, T=T)
        μ_L_q[i] = estimate(p_a, p_b, q_a, q_b, T=T)

    return μ_L_p, μ_L_q
```

Again, we first consider estimating $E[\ell(\omega)]$ by setting T=1.

We simulate 1000 times for each method.

```
N_simu = 1000
μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu)
```

```
# standard Monte Carlo (mean and std)
np.nanmean(μ_L_p), np.nanvar(μ_L_p)
```

```
(0.9975552019449763, 0.007800114741886066)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

```
(0.9997480315023506, 2.4625062564437882e-05)
```

Although both methods tend to provide a mean estimate of $E[\ell(\omega)]$ close to 1, the importance sampling estimates have smaller variance.

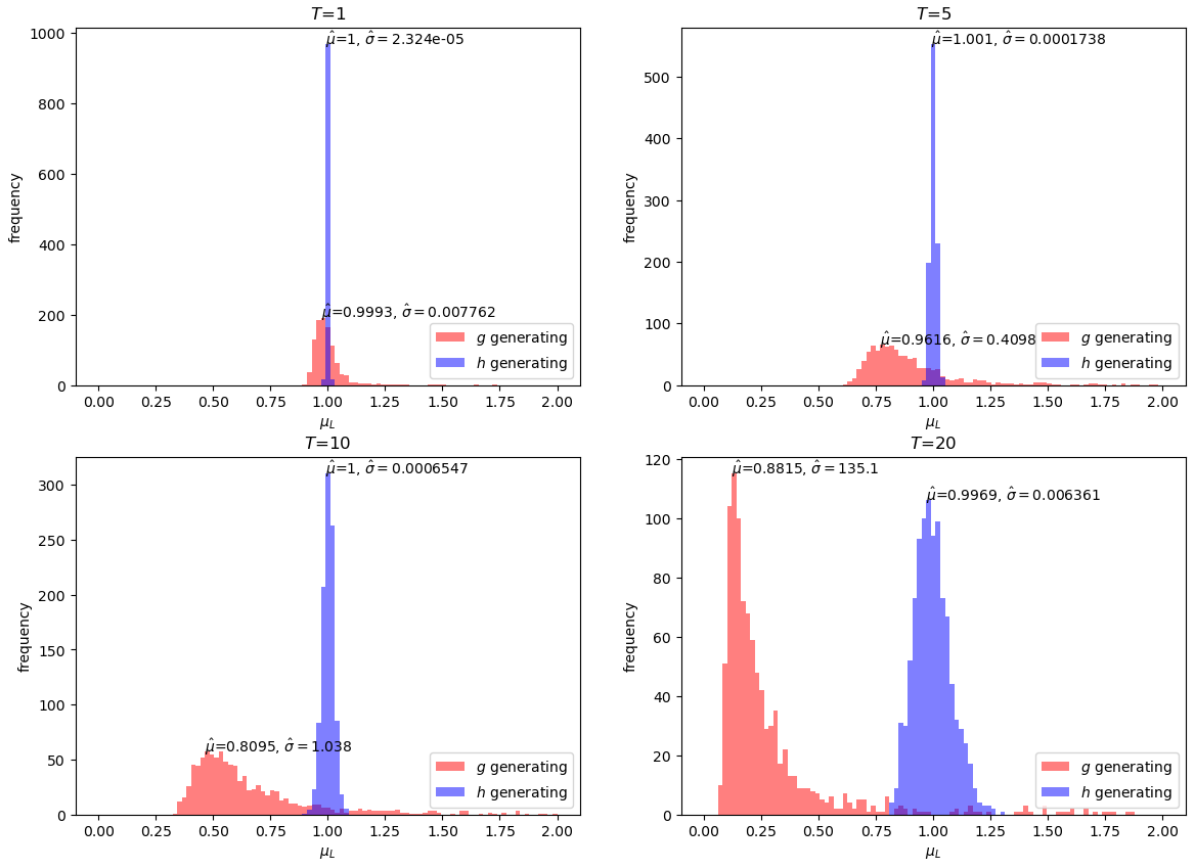
Next, we present distributions of estimates for $\hat{E}[L(\omega^t)]$, in cases for $T = 1, 5, 10, 20$.

```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 5, 10, 20]):
    row = i // 2
    col = i % 2

    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[row, col].set_xlabel('$μ_L$')
    axs[row, col].set_ylabel('frequency')
    axs[row, col].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[row, col].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5,␣
↪label='$g$ generating')
    n_q, bins_q, _ = axs[row, col].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5,␣
↪label='$h$ generating')
    axs[row, col].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
```

(continues on next page)

(continued from previous page)

```
                                    [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[row, col].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=
↪$'+f'{σ_hat:.4g}')

plt.show()
```



The simulation exercises above show that the importance sampling estimates are unbiased under all T while the standard Monte Carlo estimates are biased downwards.

Evidently, the bias increases with increases in T.

12.7 More Thoughts about Choice of Sampling Distribution

Above, we arbitraily chose $h = Beta(0.5, 0.5)$ as the importance distribution.

Is there an optimal importance distribution?

In our particular case, since we know in advance that $E_0 \left[L \left(\omega^t \right) \right] = 1$.

We can use that knowledge to our advantage.

Thus, suppose that we simply use $h = f$.

When estimating the mean of the likelihood ratio (T=1), we get:

$$\hat{E}^f\left[\ell(\omega)\frac{g(\omega)}{f(\omega)}\right] = \hat{E}^f\left[\frac{f(\omega)}{g(\omega)}\frac{g(\omega)}{f(\omega)}\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^f)\frac{g(w_i^f)}{f(w_i^f)} = 1$$

```
μ_L_p, μ_L_q = simulate(g_a, g_b, F_a, F_b, N_simu)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

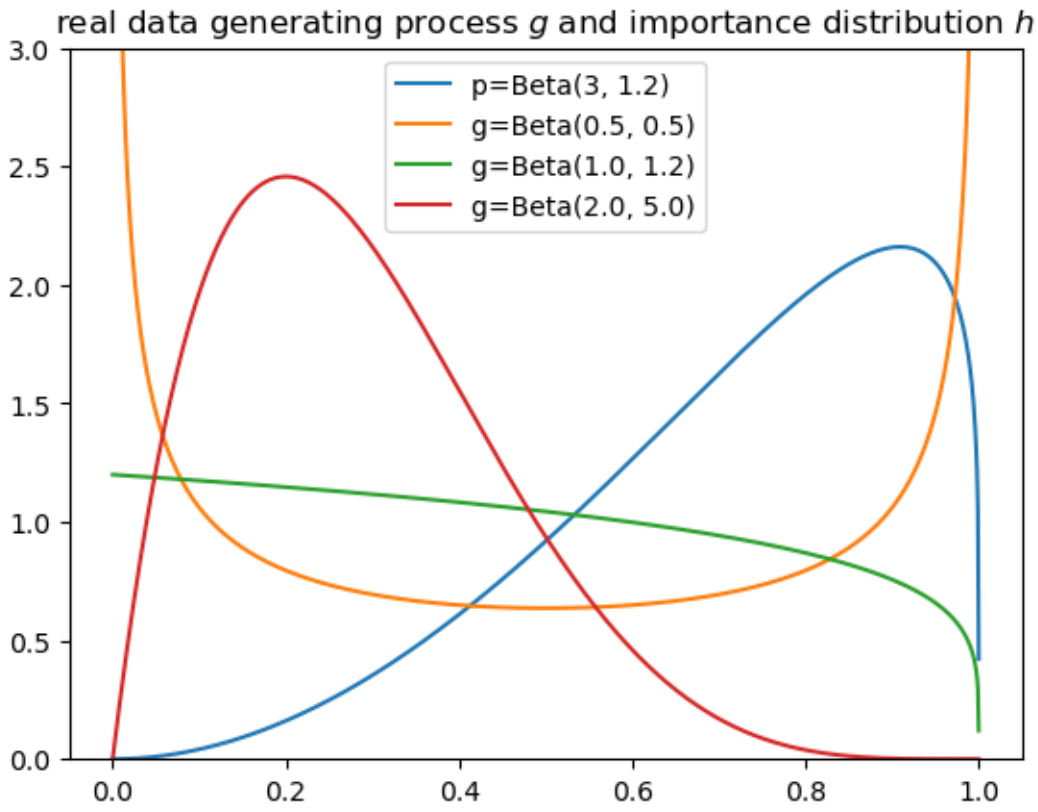
```
(1.0, 0.0)
```

We could also use other distributions as our importance distribution.

Below we choose just a few and compare their sampling properties.

```
a_list = [0.5, 1., 2.]
b_list = [0.5, 1.2, 5.]
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'p=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, a_list[0], b_list[0]), label=f'g=Beta({a_list[0]}, {b_
↪list[0]})')
plt.plot(w_range, p(w_range, a_list[1], b_list[1]), label=f'g=Beta({a_list[1]}, {b_
↪list[1]})')
plt.plot(w_range, p(w_range, a_list[2], b_list[2]), label=f'g=Beta({a_list[2]}, {b_
↪list[2]})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



We consider two additonal distributions.

As a reminder h_1 is the original $Beta(0.5, 0.5)$ distribution that we used above.

h_2 is the $Beta(1, 1.2)$ distribution.

Note how h_2 has a similar shape to g at higher values of distribution but more mass at lower values.

Our hunch is that h_2 should be a good importance sampling distribution.

h_3 is the $Beta(2, 5)$ distribution.

Note how h_3 has zero mass at values very close to 0 and at values close to 1.

Our hunch is that h_3 will be a poor importance sampling distribution.

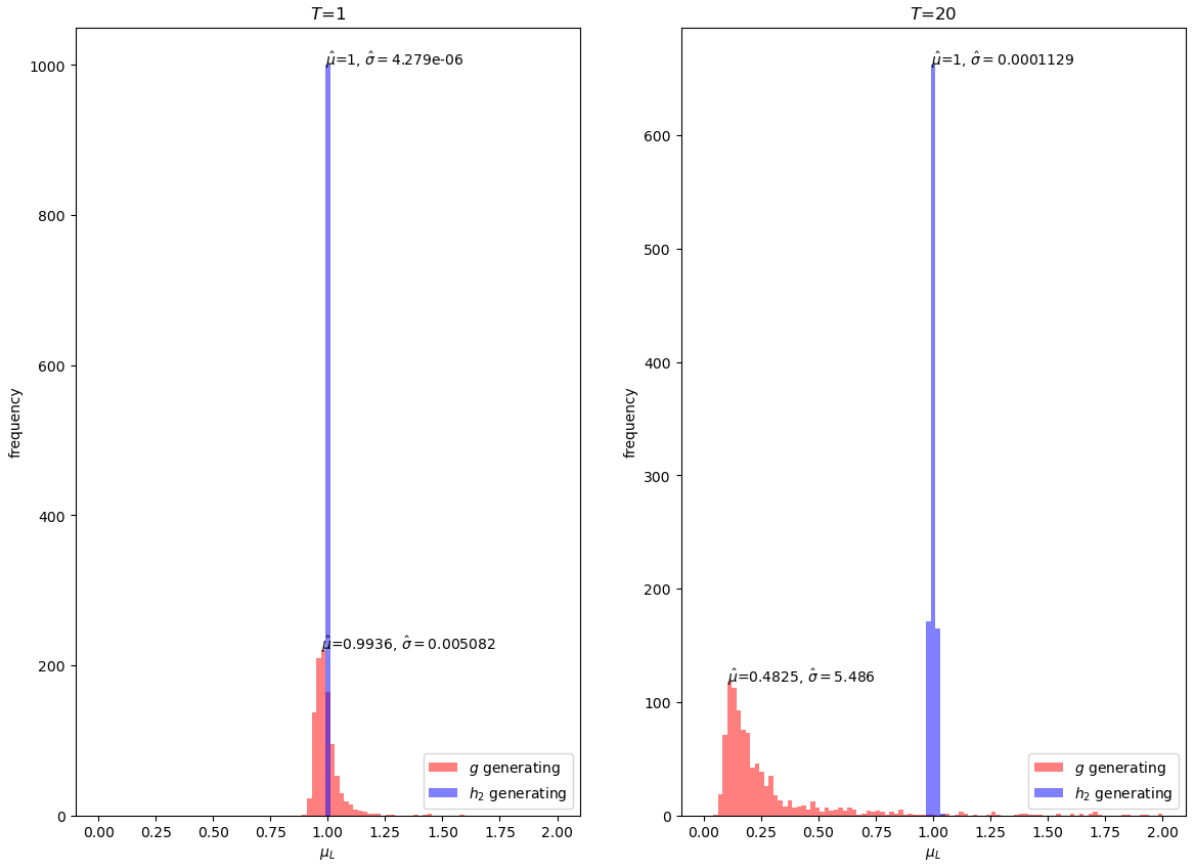
We first simulate a plot the distribution of estimates for $\hat{E}\left[L\left(\omega^t\right)\right]$ using h_2 as the importance sampling distribution.

```
h_a = a_list[1]
h_b = b_list[1]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)
```

(continues on next page)

(continued from previous page)

```
    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪2$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



Our simulations suggest that indeed h_2 is a quite good importance sampling distribution for our problem.

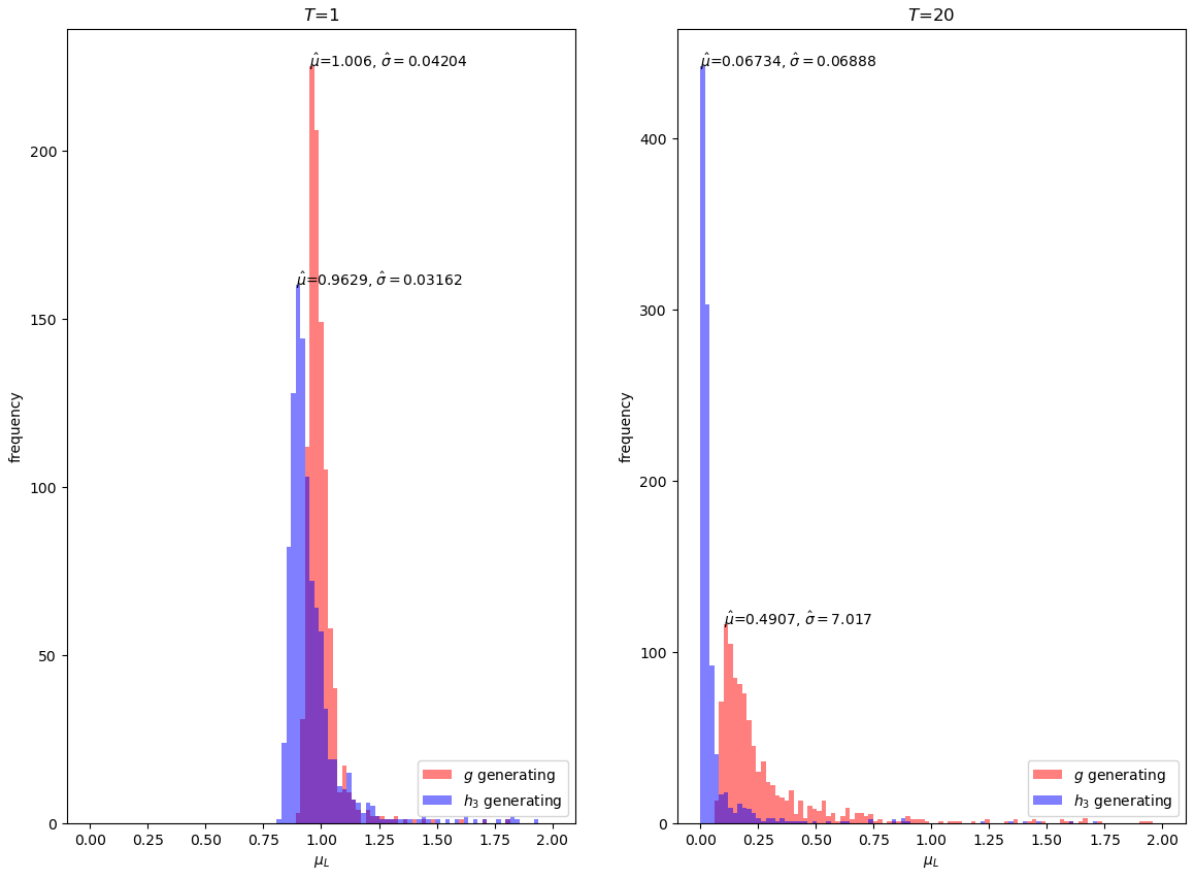
Even at $T = 20$, the mean is very close to 1 and the variance is small.

```
h_a = a_list[2]
h_b = b_list[2]

fig, axs = plt.subplots(1,2, figsize=(14, 10))
```

(continues on next page)

(continued from previous page)

```
μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪3$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], '$\hat{μ}$='+f'{μ_hat:.4g}'+', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



However, h_3 is evidently a poor importance sampling distribution forpir problem, with a mean estimate far away from 1 for $T = 20$.

Notice that evan at $T = 1$, the mean estimate with importance sampling is more biased than just sampling with g itself.

Thus, our simulations suggest that we would be better off simply using Monte Carlo approximations under g than using h_3 as an importance sampling distribution for our problem.

CHAPTER

THIRTEEN

A PROBLEM THAT STUMPED MILTON FRIEDMAN

(and that Abraham Wald solved by inventing sequential analysis)

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

13.1 Overview

This lecture describes a statistical decision problem presented to Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

This problem led Abraham Wald [Wal47] to formulate **sequential analysis**, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem.

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error occures when you accept a null hypothesis that is false

- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

This lecture uses ideas studied in *this lecture*, *this lecture*. and *this lecture*.

13.2 Origin of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [FF98], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University.

Note: See pages 25 and 26 of Allen Wallis's 1980 article [Wal80] about the Statistical Research Group at Columbia University during World War II for his account of the episode and for important contributions that Harold Hotelling made to formulating the problem. Also see chapter 5 of Jennifer Burns book about Milton Friedman [Bur23].

Let's listen to Milton Friedman tell us what happened

> In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round, it assigns the value 1 or 0 to A accordingly as its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.
>
> The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.
>
> When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for ....

Friedman and Wallis struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald.

That started Wald on the path that led him to *Sequential Analysis* [Wal47].

We'll formulate the problem using dynamic programming.

13.3 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** [Ber75].

A decision-maker can observe a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the sequence is not IID.

The observer has something to learn, namely, whether the observations are drawn from f_0 or from f_1.

The decision maker wants to decide which of the two distributions is generating outcomes.

We adopt a Bayesian formulation.

The decision maker begins with a prior probability

$$\pi_{-1} = \mathbb{P}\{f = f_0 \mid \text{ no observations}\} \in (0,1)$$

After observing $k+1$ observations $z_k, z_{k-1}, \ldots, z_0$, he updates his personal probability that the observations are described by distribution f_0 to

$$\pi_k = \mathbb{P}\{f = f_0 \mid z_k, z_{k-1}, \ldots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_0(z_{k+1})}{\pi_k f_0(z_{k+1}) + (1-\pi_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = \pi_k f_0(v) + (1-\pi_k) f_1(v),$$

which is a mixture of distributions f_0 and f_1, with the weight on f_0 being the posterior probability that $f = f_0$[1].

To illustrate such a distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters a and b is

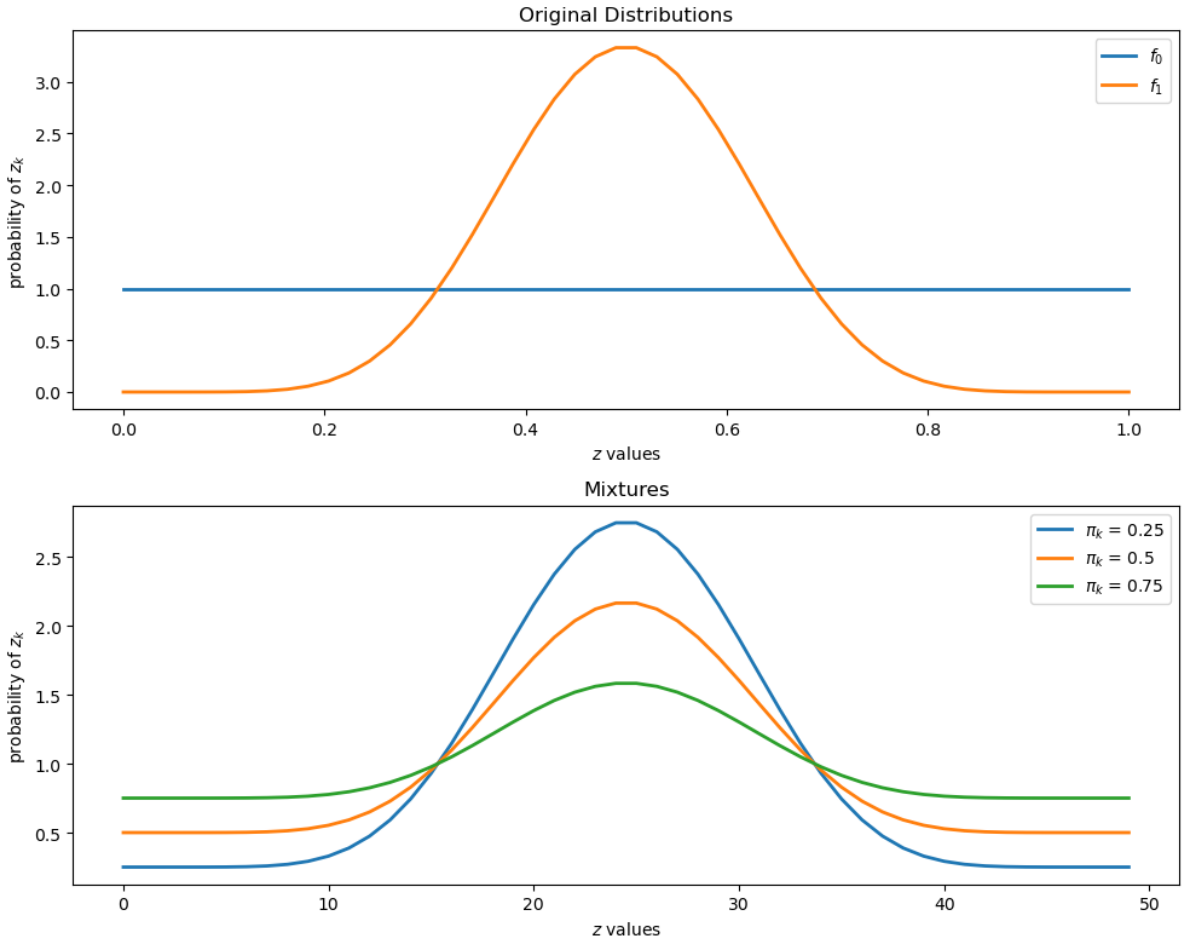
$$f(z; a, b) = \frac{\Gamma(a+b) z^{a-1} (1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

The next figure shows two beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities π_k

[1] The decision maker acts as if he believes that the sequence of random variables $[z_0, z_1, \ldots]$ is *exchangeable*. See Exchangeability and Bayesian Updating and [Kre88] chapter 11, for discussions of exchangeability.

```
@jit(nopython=True)
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)

f0 = lambda x: p(x, 1, 1)
f1 = lambda x: p(x, 9, 9)
grid = np.linspace(0, 1, 50)

fig, axes = plt.subplots(2, figsize=(10, 8))

axes[0].set_title("Original Distributions")
axes[0].plot(grid, f0(grid), lw=2, label="$f_0$")
axes[0].plot(grid, f1(grid), lw=2, label="$f_1$")

axes[1].set_title("Mixtures")
for π in 0.25, 0.5, 0.75:
    y = π * f0(grid) + (1 - π) * f1(grid)
    axes[1].plot(y, lw=2, label=f"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```



13.3.1 Losses and Costs

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision-maker can suffer three kinds of losses:

- A loss L_0 if he decides $f = f_0$ when actually $f = f_1$
- A loss L_1 if he decides $f = f_1$ when actually $f = f_0$
- A cost c if he postpones deciding and chooses instead to draw another z

13.3.2 Digression on Type I and Type II Errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then L_1 and L_0 are losses associated with two types of statistical errors

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

So when we treat $f = f_0$ as the null hypothesis

- We can think of L_1 as the loss associated with a type I error.
- We can think of L_0 as the loss associated with a type II error.

13.3.3 Intuition

Before proceeding, let's try to guess what an optimal decision rule might look like.

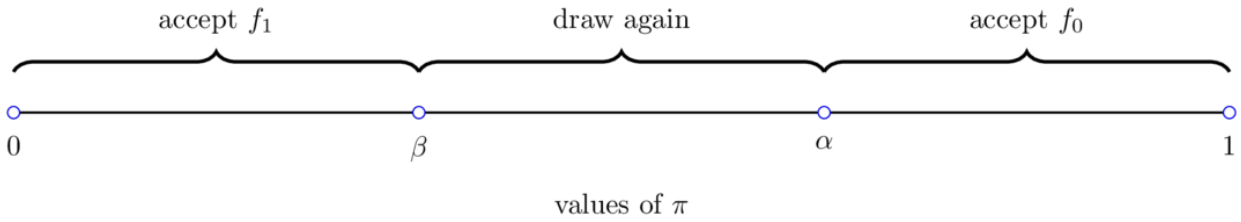
Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_0$.

If, on the other hand, π is close to 0, then $f = f_1$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we are confronted with more uncertainty.

This reasoning suggests a decision rule such as the one shown in the figure



As we'll see, this is indeed the correct form of the decision rule.

Our problem is to determine threshold values α, β that somehow depend on the parameters described above.

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on α or β.

13.3.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

With some thought, you will agree that J should satisfy the Bellman equation

$$J(\pi) = \min \{(1-\pi)L_0,\ \pi L_1,\ c + \mathbb{E}[J(\pi')]\} \tag{13.1}$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = \pi f_0(v) + (1-\pi) f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. Accept the hypothesis that $f = f_0$
2. Accept the hypothesis that $f = f_1$
3. Postpone deciding and draw again

We can represent the Bellman equation as

$$J(\pi) = \min\{(1-\pi)L_0,\ \pi L_1,\ h(\pi)\} \tag{13.2}$$

where $\pi \in [0,1]$ and

- $(1-\pi)L_0$ is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- πL_1 is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$; this is the continuation value; i.e., the expected cost associated with drawing one more z.

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0,1) \times (0,1)$ that satisfy

$$(1-\pi)L_0 < \min\{\pi L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq \alpha$$

and

$$\pi L_1 < \min\{(1-\pi)L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq \beta$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_0 \text{ if } \pi \geq \alpha \\ \text{accept } f = f_1 \text{ if } \pi \leq \beta \\ \text{draw another } z \text{ if } \beta \leq \pi \leq \alpha \end{aligned}$$

Our aim is to compute the cost function J, and from it the associated cutoffs α and β.

To make our computations manageable, using (13.2), we can write the continuation cost $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{(1-\pi')L_0, \pi' L_1, h(\pi')\} \\ &= c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \end{aligned} \tag{13.3}$$

The equality

$$h(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \tag{13.4}$$

is a **functional equation** in an unknown function h.

Using the functional equation, (13.4), for the continuation cost, we can back out optimal choices using the right side of (13.2).

This functional equation can be solved by taking an initial guess and iterating to find a fixed point.

Thus, we iterate with an operator Q, where

$$Qh(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz'$$

13.4 Implementation

First, we will construct a `jitclass` to store the parameters of the model

```
wf_data = [('a0', float64),          # Parameters of beta distributions
           ('b0', float64),
           ('a1', float64),
           ('b1', float64),
           ('c', float64),           # Cost of another draw
           ('π_grid_size', int64),
           ('L0', float64),          # Cost of selecting f0 when f1 is true
           ('L1', float64),          # Cost of selecting f1 when f0 is true
           ('π_grid', float64[:]),
           ('mc_size', int64),
           ('z0', float64[:]),
           ('z1', float64[:])]
```

```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=25,
                 L1=25,
                 π_grid_size=200,
                 mc_size=1000):

        self.a0, self.b0 = a0, b0
        self.a1, self.b1 = a1, b1
        self.c, self.π_grid_size = c, π_grid_size
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def f0_rvs(self):
        return np.random.beta(self.a0, self.b0)

    def f1_rvs(self):
        return np.random.beta(self.a1, self.b1)

    def κ(self, z, π):
```

(continues on next page)

(continued from previous page)

```
        """
        Updates π using Bayes' rule and the current observation z
        """

        f0, f1 = self.f0, self.f1

        π_f0, π_f1 = π * f0(z), (1 - π) * f1(z)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new
```

As in the optimal growth lecture, to approximate a continuous value function

- We iterate at a finite grid of possible values of π.
- When we evaluate $\mathbb{E}[J(\pi')]$ between grid points, we use linear interpolation.

We define the operator function `Q` below.

```
@jit(nopython=True, parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: interp(π_grid, h, p)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

To solve the key functional equation, we will iterate using `Q` to find the fixed point

```
@jit(nopython=True)
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation cost function
```

(continues on next page)

(continued from previous page)

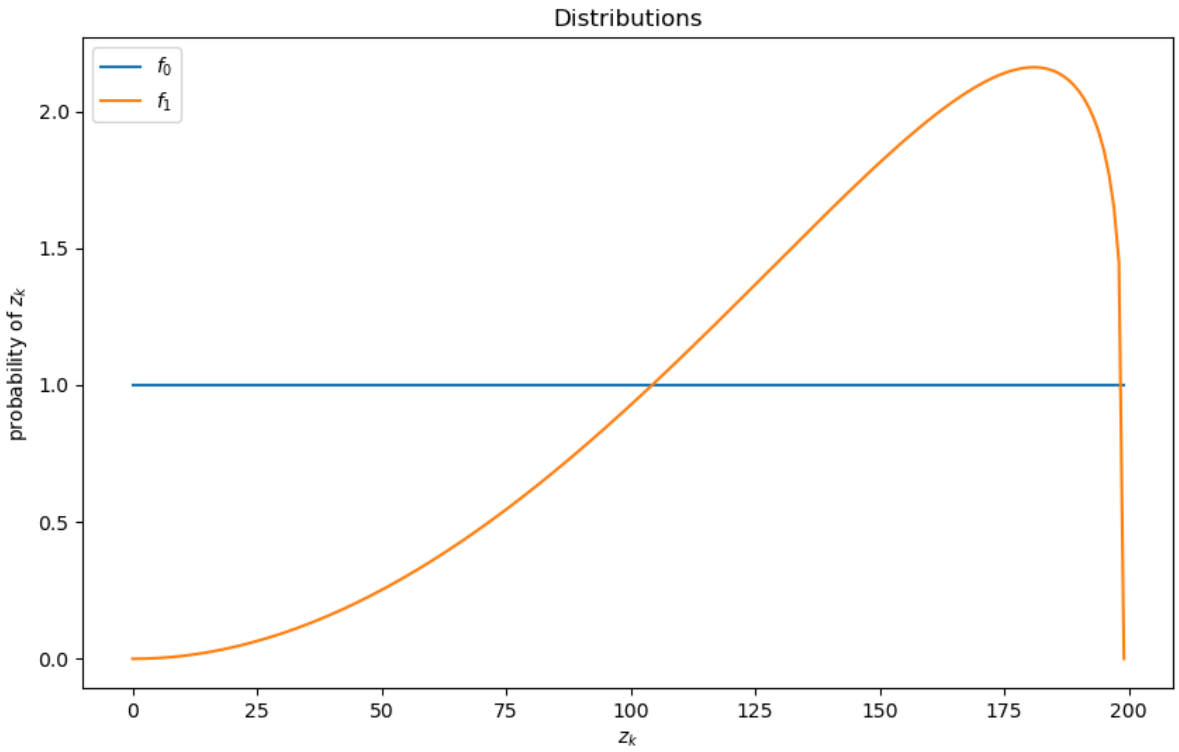
```
    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

13.5 Analysis

Let's inspect outcomes.

We will be using the default parameterization with distributions like so

```
wf = WaldFriedman()

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(wf.f0(wf.π_grid), label="$f_0$")
ax.plot(wf.f1(wf.π_grid), label="$f_1$")
ax.set(ylabel="probability of $z_k$", xlabel="$z_k$", title="Distributions")
ax.legend()

plt.show()
```



13.5.1 Value Function

To solve the model, we will call our `solve_model` function

```
h_star = solve_model(wf)     # Solve the model
```

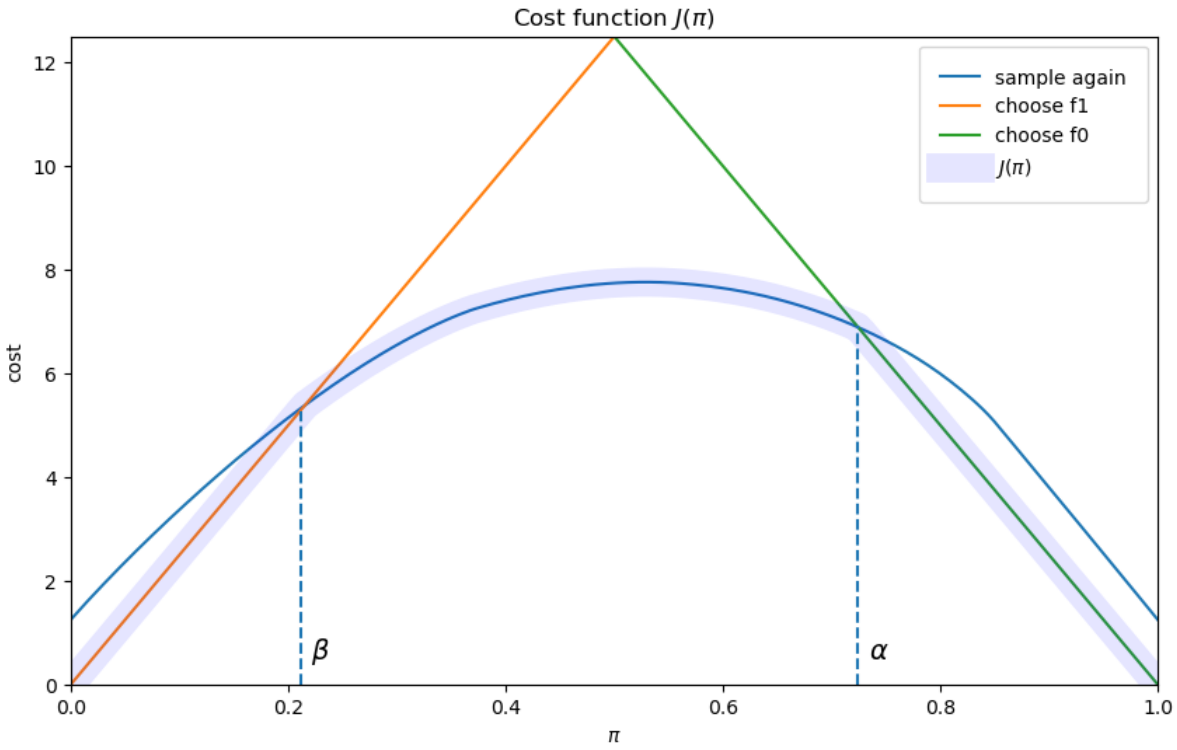
We will also set up a function to compute the cutoffs α and β and plot these on our cost function plot

```
@jit(nopython=True)
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation cost function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    β = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
```

(continues on next page)

(continued from previous page)

```
                - 1]
    α = π_grid[np.searchsorted(
                                np.minimum(h, payoff_f1) - payoff_f0,
                                1e-10)
                - 1]

    return (β, α)

β, α = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='sample again')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='$J(\pi)$')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Cost function $J(\pi)$")

plt.legend(borderpad=1.1)
plt.show()
```



The cost function J equals πL_1 for $\pi \leq \beta$, and $(1 - \pi)L_0$ for $\pi \geq \alpha$.

The slopes of the two linear pieces of the cost function $J(\pi)$ are determined by L_1 and $-L_0$.

The cost function J is smooth in the interior region, where the posterior probability assigned to f_0 is in the indecisive region $\pi \in (\beta, \alpha)$.

The decision-maker continues to sample until the probability that he attaches to model f_0 falls below β or above α.

13.5.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of **stopping times**, i.e., the number of draws of z_k required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case, the decision-maker is correct 80% of the time

```
def simulate(wf, true_dist, h_star, π_0=0.5):

    """
    This function takes an initial condition and simulates until it
    stops (when a decision is made)
    """

    f0, f1 = wf.f0, wf.f1
    f0_rvs, f1_rvs = wf.f0_rvs, wf.f1_rvs
    π_grid = wf.π_grid
    κ = wf.κ
```

(continues on next page)

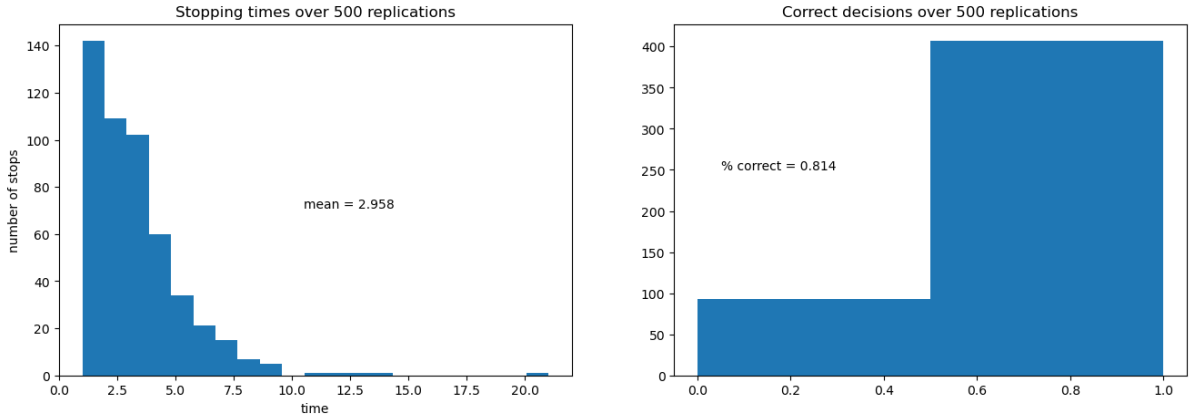
(continued from previous page)

```
    if true_dist == "f0":
        f, f_rvs = wf.f0, wf.f0_rvs
    elif true_dist == "f1":
        f, f_rvs = wf.f1, wf.f1_rvs

    # Find cutoffs
    β, α = find_cutoff_rule(wf, h_star)

    # Initialize a couple of useful variables
    decision_made = False
    π = π_0
    t = 0

    while decision_made is False:
        # Maybe should specify which distribution is correct one so that
        # the draws come from the "right" distribution
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < β:
            decision_made = True
            decision = 1
        elif π > α:
            decision_made = True
            decision = 0

    if true_dist == "f0":
        if decision == 0:
            correct = True
        else:
            correct = False

    elif true_dist == "f1":
        if decision == 1:
            correct = True
        else:
            correct = False

    return correct, π, t

def stopping_dist(wf, h_star, ndraws=250, true_dist="f0"):

    """
    Simulates repeatedly to get distributions of time needed to make a
    decision and how often they are correct
    """

    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, π, t = simulate(wf, true_dist, h_star)
        tdist[i] = t
        cdist[i] = correct
```

(continues on next page)

(continued from previous page)

```
    return cdist, tdist

def simulation_plot(wf):
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                   max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

    ax[1].hist(cdist.astype(int), bins=2)
    ax[1].set_title(f"Correct decisions over {ndraws} replications")
    ax[1].annotate(f"% correct = {np.mean(cdist)}",
                   xy=(0.05, ndraws / 2))

    plt.show()

simulation_plot(wf)
```



13.5.3 Comparative Statics

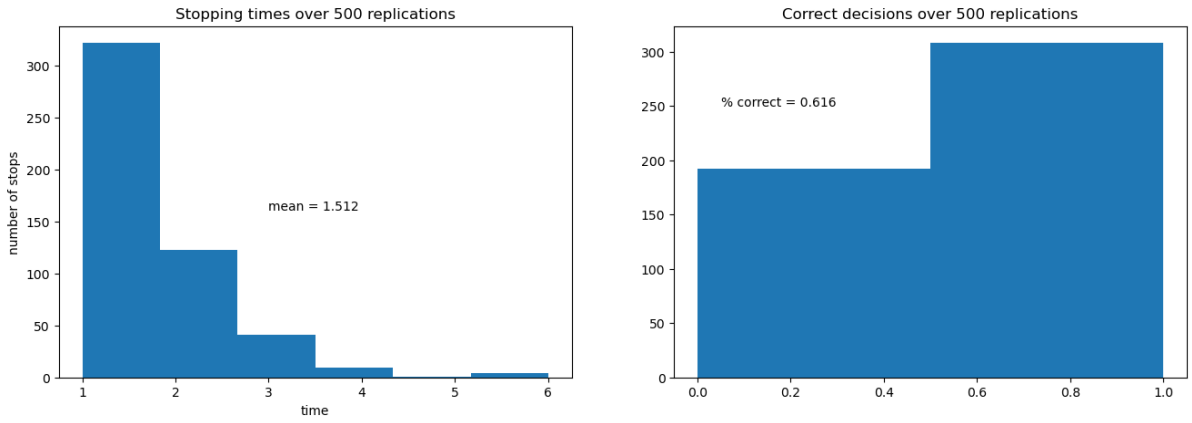
Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision-maker be correct more or less often?
- Will he make decisions sooner or later?

```
wf = WaldFriedman(c=2.5)
simulation_plot(wf)
```



Increased cost per draw has induced the decision-maker to take fewer draws before deciding.

Because he decides with fewer draws, the percentage of time he is correct drops.

This leads to him having a higher expected loss when he puts equal weight on both models.

13.5.4 A Notebook Implementation

To facilitate comparative statics, we provide a Jupyter notebook that generates the same plots, but with sliders.

With these sliders, you can adjust parameters and immediately observe

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.
- effects of different settings for the cost parameters L_0, L_1, c, the parameters of two beta distributions f_0 and f_1, and the number of points and linear functions m to use in the piece-wise continuous approximation to the value function.
- various simulations from f_0 and associated distributions of waiting times to making a decision.
- associated histograms of correct and incorrect decisions.

13.6 Comparison with Neyman-Pearson Formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captail Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We'll rely on Abraham Wald's [Wal47] elegant summary of Neyman-Pearson theory.

For our purposes, watch for there features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable.
- The parameters β and α characterize cut-off rules used to determine n as a random variable.

- Laws of large numbers make no appearances in the sequential construction.

In chapter 1 of **Sequential Analysis** [Wal47] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well-posed problem – usually, *something* means *a lot*)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas:

- A decision-maker wants to decide which of two distributions f_0, f_1 govern an IID random variable z.
- The null hypothesis H_0 is the statement that f_0 governs the data.
- The alternative hypothesis H_1 is the statement that f_1 governs the data.
- The problem is to devise and analyze a test of hypothesis H_0 against the alternative hypothesis H_1 on the basis of a sample of a fixed number n independent observations $z_1, z_2, \ldots, z_n$ of the random variable z.

To quote Abraham Wald,

> A test procedure leading to the acceptance or rejection of the [null] hypothesis in question is simply a rule specifying, for each possible sample of size n, whether the [null] hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size n into two mutually exclusive parts, say part 1 and part 2, together with the application of the rule that the [null] hypothesis be accepted if the observed sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size n which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

> As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies law of large numbers to interpret α and β:

> The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if $H_0$is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1 - \beta$ is called the *power* of the critical region.

Wald notes that

> one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is possible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

> Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \ldots, z_n)$ which satisfy the inequality
>
> $$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_0(z_n)} \geq k$$
>
> is a most powerful critical region for testing the hypothesis H_0 against the alternative hypothesis H_1. The term k on the right side is a constant chosen so that the region will have the required size α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

> A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H , (2) to reject the hypothesis H , (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation, one of the aforementioned decision is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations, one of the three decision is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

13.7 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- *this lecture* discusses the key concept of **exchangeability** that rationalizes statistical learning
- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* discusses the role of likelihood ratio processes in **Bayesian learning**
- *this lecture* returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

---

CHAPTER

FOURTEEN

EXCHANGEABILITY AND BAYESIAN UPDATING

Contents

14.1 Overview

This lecture studies learning via Bayes' Law.

We touch foundations of Bayesian statistical inference invented by Bruno DeFinetti [dF37].

The relevance of DeFinetti's work for economists is presented forcefully in chapter 11 of [Kre88] by David Kreps.

An example that we study in this lecture is a key component of this lecture that augments the classic job search model of McCall [McC70] by presenting an unemployed worker with a statistical inference problem.

Here we create graphs that illustrate the role that a likelihood ratio plays in Bayes' Law.

We'll use such graphs to provide insights into mechanics driving outcomes in this lecture about learning in an augmented McCall job search model.

Among other things, this lecture discusses connections between the statistical concepts of sequences of random variables that are

- independently and identically distributed
- exchangeable (also known as *conditionally* independently and identically distributed)

Understanding these concepts is essential for appreciating how Bayesian updating works.

You can read about exchangeability here.

Because another term for **exchangeable** is **conditionally independent**, we want to convey an answer to the question *conditional on what?*

We also tell why an assumption of independence precludes learning while an assumption of conditional independence makes learning possible.

Below, we'll often use

- W to denote a random variable
- w to denote a particular realization of a random variable W

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
```

14.2 Independently and Identically Distributed

We begin by looking at the notion of an **independently and identically distributed sequence** of random variables.

An independently and identically distributed sequence is often abbreviated as IID.

Two notions are involved

- **independence**
- **identically distributed**

A sequence $W_0, W_1, ...$ is **independently distributed** if the joint probability density of the sequence is the **product** of the densities of the components of the sequence.

The sequence $W_0, W_1, ...$ is **independently and identically distributed** (IID) if in addition the marginal density of W_t is the same for all $t = 0, 1, ...$

For example, let $p(W_0, W_1, ...)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, ...$

Then the joint density of the sequence $W_0, W_1, ...$ is IID if

$$p(W_0, W_1, ...) = p(W_0)p(W_1)\cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

14.2.1 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence is random variables is IID, past information provides no information about future realizations.

Therefore, there is **nothing to learn** from the past about the future.

To understand these statements, let the joint distribution of a sequence of random variables $\{W_t\}_{t=0}^T$ that is not necessarily IID be

$$p(W_T, W_{T-1}, \dots, W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$\begin{aligned} p(W_T, W_{T-1}, \dots, W_1, W_0) =& p(W_T|W_{T-1}, \dots, W_0)p(W_{T-1}|W_{T-2}, \dots, W_0)\cdots \\ & \cdots p(W_1|W_0)p(W_0) \end{aligned}$$

In general,

$$p(W_t|W_{t-1}, \dots, W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

But in the special IID case,

$$p(W_t|W_{t-1}, \dots, W_0) = p(W_t)$$

and partial history $W_{t-1}, \dots, W_0$ contains no information about the probability of W_t.

So in the IID case, there is **nothing to learn** about the densities of future random variables from past random variables.

But when the sequence is not IID, there is something to learn about the future from observations of past random variables.

We turn next to an instance of the general case in which the sequence is not IID.

Please watch for what can be learned from the past and when.

14.3 A Setting in Which Past Observations Are Informative

Let $\{W_t\}_{t=0}^\infty$ be a sequence of nonnegative scalar random variables with a joint probability distribution constructed as follows.

There are two distinct cumulative distribution functions F and G that have densities f and g, respectively, for a nonnegative scalar random variable W.

Before the start of time, say at time $t = -1$, "nature" once and for all selects **either** f **or** g.

Thereafter at each time $t \geq 0$, nature draws a random variable W_t from the selected distribution.

So the data are permanently generated as independently and identically distributed (IID) draws from **either** F **or** G.

We could say that *objectively*, meaning *after* nature has chosen either F or G, the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a partially informed decision maker who knows

- both F and G, but
- not the F or G that nature drew once-and-for-all at $t = -1$

So our decision maker does not know which of the two distributions nature selected.

The decision maker describes his ignorance with a **subjective probability** $\tilde{\pi}$ and reasons as if nature had selected F with probability $\tilde{\pi} \in (0, 1)$ and G with probability $1 - \tilde{\pi}$.

Thus, we assume that the decision maker

- **knows** both F and G
- **doesn't know** which of these two distributions that nature has drawn
- expresses his ignorance by **acting as if** or **thinking that** nature chose distribution F with probability $\tilde{\pi} \in (0, 1)$ and distribution G with probability $1 - \tilde{\pi}$
- at date $t \geq 0$ knows the partial history $w_t, w_{t-1}, \ldots, w_0$

To proceed, we want to know the decision maker's belief about the joint distribution of the partial history.

We'll discuss that next and in the process describe the concept of **exchangeability**.

14.4 Relationship Between IID and Exchangeable

Conditional on nature selecting F, the joint density of the sequence $W_0, W_1, \ldots$ is

$$f(W_0)f(W_1)\cdots$$

Conditional on nature selecting G, the joint density of the sequence $W_0, W_1, \ldots$ is

$$g(W_0)g(W_1)\cdots$$

Thus, **conditional on nature having selected** F, the sequence $W_0, W_1, \ldots$ is independently and identically distributed.

Furthermore, **conditional on nature having selected** G, the sequence $W_0, W_1, \ldots$ is also independently and identically distributed.

But what about the **unconditional distribution** of a partial history?

The unconditional distribution of $W_0, W_1, \ldots$ is evidently

$$h(W_0, W_1, \ldots) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1 - \tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{14.1}$$

Under the unconditional distribution $h(W_0, W_1, \ldots)$, the sequence $W_0, W_1, \ldots$ is **not** independently and identically distributed.

To verify this claim, it is sufficient to notice, for example, that

$$h(W_0, W_1) = \tilde{\pi}f(W_0)f(W_1) + (1 - \tilde{\pi})g(W_0)g(W_1) \neq (\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))(\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

Thus, the conditional distribution

$$h(W_1|W_0) \equiv \frac{h(W_0, W_1)}{(\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))} \neq (\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

This means that random variable W_0 contains information about random variable W_1.

So there is something to learn from the past about the future.

But what and how?

14.5 Exchangeability

While the sequence $W_0, W_1, ...$ is not IID, it can be verified that it is **exchangeable**, which means that the ``re-ordered" joint distributions $h(W_0, W_1)$ and $h(W_1, W_0)$ satisfy

$$h(W_0, W_1) = h(W_1, W_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for a sequence does not change when the positions in the sequence in which finitely many of random variables appear are altered.

Equation (14.1) represents our instance of an exchangeable joint density over a sequence of random variables as a **mixture** of two IID joint densities over a sequence of random variables.

For a Bayesian statistician, the mixing parameter $\tilde{\pi} \in (0, 1)$ has a special interpretation as a subjective **prior probability** that nature selected probability distribution F.

DeFinetti [dF37] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameter $\theta \in (0, 1)$ and mixing probability density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli parameter θ.

14.6 Bayes' Law

We noted above that in our example model there is something to learn about about the future from past data drawn from our particular instance of a process that is exchangeable but not IID.

But how can we learn?

And about what?

The answer to the *about what* question is $\tilde{\pi}$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *from a (subjective) joint distribution, compute an appropriate conditional distribution*.

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as a decision maker's **subjective probability** (also called a **personal probability**).

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, ..., w_0]$.

We let

$$\pi_t = \mathbb{P}\{q = f | w^t\}$$

where we adopt the convention

$$\pi_{-1} = \tilde{\pi}$$

The distribution of w_{t+1} conditional on w^t is then

$$\pi_t f + (1 - \pi_t) g.$$

Bayes' rule for updating π_{t+1} is

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1})} \tag{14.2}$$

Equation (14.2) follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}}$$

where

$$\mathbb{P}\{W = w\} = \sum_{a \in \{f,g\}} \mathbb{P}\{W = w \,|\, q = a\}\mathbb{P}\{q = a\}$$

14.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (14.2) with the aim of understanding how the **posterior** probability π_{t+1} is influenced by the **prior** probability π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (14.2) as

$$\pi_{t+1} = \frac{\pi_t f\left(w_{t+1}\right)}{\pi_t f\left(w_{t+1}\right) + (1-\pi_t) g\left(w_{t+1}\right)} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + (1-\pi_t)} = \frac{\pi_t l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + (1-\pi_t)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + (1-\pi_t)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{14.3}$$

Notice how the likelihood ratio and the prior interact to determine whether an observation w_{t+1} leads the decision maker to increase or decrease the subjective probability he/she attaches to distribution F.

When the likelihood ratio $l(w_{t+1})$ exceeds one, the observation w_{t+1} nudges the probability π put on distribution F upward, and when the likelihood ratio $l(w_{t+1})$ is less that one, the observation w_{t+1} nudges π downward.

Representation (14.3) is the foundation of some graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^{\infty}$ that are induced by Bayes' Law.

We'll plot $l\left(w\right)$ as a way to enlighten us about how learning – i.e., Bayesian updating of the probability π that nature has chosen distribution f – works.

To create the Python infrastructure to do our work for us, we construct a wrapper function that displays informative graphs given parameters of f and g.

```
@vectorize
def p(x, a, b):
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x ** (a-1) * (1 - x) ** (b-1)

def learning_example(F_a=1, F_b=1, G_a=3, G_b=1.2):
```

(continues on next page)

(continued from previous page)

```
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1

    x_grid = np.linspace(0, 1, 100)
    π_grid = np.linspace(1e-3, 1-1e-3, 100)

    w_max = 1
    w_grid = np.linspace(1e-12, w_max-1e-12, 100)

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

    ax1.plot(l(w_grid), w_grid, label='$l$', lw=2)
    ax1.vlines(1., 0., 1., linestyle="--")
    ax1.hlines(roots, 0., 2., linestyle="--")
    ax1.set_xlim([0., 2.])
    ax1.legend(loc=4)
    ax1.set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    ax2.plot(f(x_grid), x_grid, label='$f$', lw=2)
    ax2.plot(g(x_grid), x_grid, label='$g$', lw=2)
    ax2.vlines(1., 0., 1., linestyle="--")
    ax2.hlines(roots, 0., 2., linestyle="--")
    ax2.legend(loc=4)
    ax2.set(xlabel='$f(w), g(w)$', ylabel='$w$')

    area1 = quad(f, 0, roots[0])[0]
    area2 = quad(g, roots[0], roots[1])[0]
    area3 = quad(f, roots[1], 1)[0]

    ax2.text((f(0) + f(roots[0])) / 4, roots[0] / 2, f"{area1: .3g}")
    ax2.fill_between([0, 1], 0, roots[0], color='blue', alpha=0.15)
    ax2.text(np.mean(g(roots)) / 2, np.mean(roots), f"{area2: .3g}")
    w_roots = np.linspace(roots[0], roots[1], 20)
    ax2.fill_betweenx(w_roots, 0, g(w_roots), color='orange', alpha=0.15)
    ax2.text((f(roots[1]) + f(1)) / 4, (roots[1] + 1) / 2, f"{area3: .3g}")
    ax2.fill_between([0, 1], roots[1], 1, color='blue', alpha=0.15)

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)
```

(continues on next page)

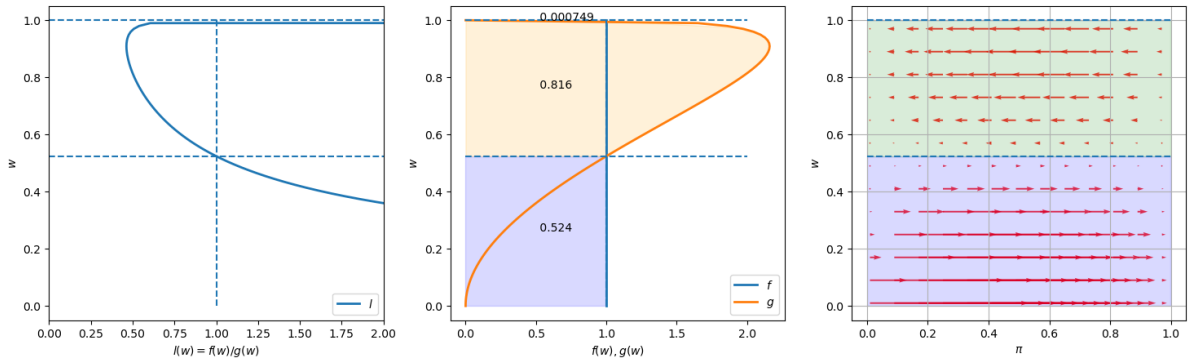
(continued from previous page)

```
    ΔW = np.zeros((len(W), len(Π)))
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = ax3.quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    ax3.fill_between(π_grid, 0, roots[0], color='blue', alpha=0.15)
    ax3.fill_between(π_grid, roots[0], roots[1], color='green', alpha=0.15)
    ax3.fill_between(π_grid, roots[1], w_max, color='blue', alpha=0.15)
    ax3.hlines(roots, 0., 1., linestyle="--")
    ax3.set(xlabel='$\pi$', ylabel='$w$')
    ax3.grid()

    plt.show()
```

Now we'll create a group of graphs that illustrate dynamics induced by Bayes' Law.

We'll begin with Python function default values of various objects, then change them in a subsequent example.

```
learning_example()
```



Please look at the three graphs above created for an instance in which f is a uniform distribution on $[0, 1]$ (i.e., a Beta distribution with parameters $F_a = 1, F_b = 1$), while g is a Beta distribution with the default parameter values $G_a = 3, G_b = 1.2$.

The graph on the left plots the likelihood ratio $l(w)$ as the absciassa axis against w as the ordinate.

The middle graph plots both $f(w)$ and $g(w)$ against w, with the horizontal dotted lines showing values of w at which the likelihood ratio equals 1.

The graph on the right plots arrows to the right that show when Bayes' Law makes π increase and arrows to the left that show when Bayes' Law make π decrease.

Lengths of the arrows show magnitudes of the force from Bayes' Law impelling π to change.

These lengths depend on both the prior probability π on the abscissa axis and the evidence in the form of the current draw of w on the ordinate axis.

The fractions in the colored areas of the middle graphs are probabilities under F and G, respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and toward 1 when F is the true distribution).

For example, in the above example, under true distribution F, π will be updated toward 0 if w falls into the interval $[0.524, 0.999]$, which occurs with probability $1 - .524 = .476$ under F.

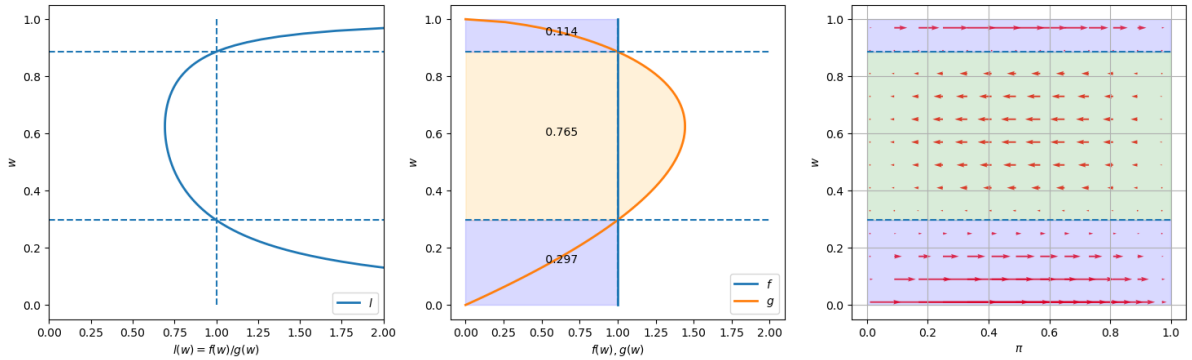
But this would occur with probability 0.816 if G were the true distribution.

The fraction 0.816 in the orange region is the integral of $g(w)$ over this interval.

Next we use our code to create graphs for another instance of our model.

We keep F the same as in the preceding instance, namely a uniform distribution, but now assume that G is a Beta distribution with parameters $G_a = 2, G_b = 1.6$.

```
learning_example(G_a=2, G_b=1.6)
```



Notice how the likelihood ratio, the middle graph, and the arrows compare with the previous instance of our example.

14.8 Appendix

14.8.1 Sample Paths of π_t

Now we'll have some fun by plotting multiple realizations of sample paths of π_t under two possible assumptions about nature's choice of distribution, namely

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes discussed in this lecture.

To proceed, we create some Python code.

```
def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    @njit
    def update(a, b, π):
        "Update π by drawing from beta distribution with parameters a and b"

        # Draw
        w = np.random.beta(a, b)
```

(continues on next page)

(continued from previous page)

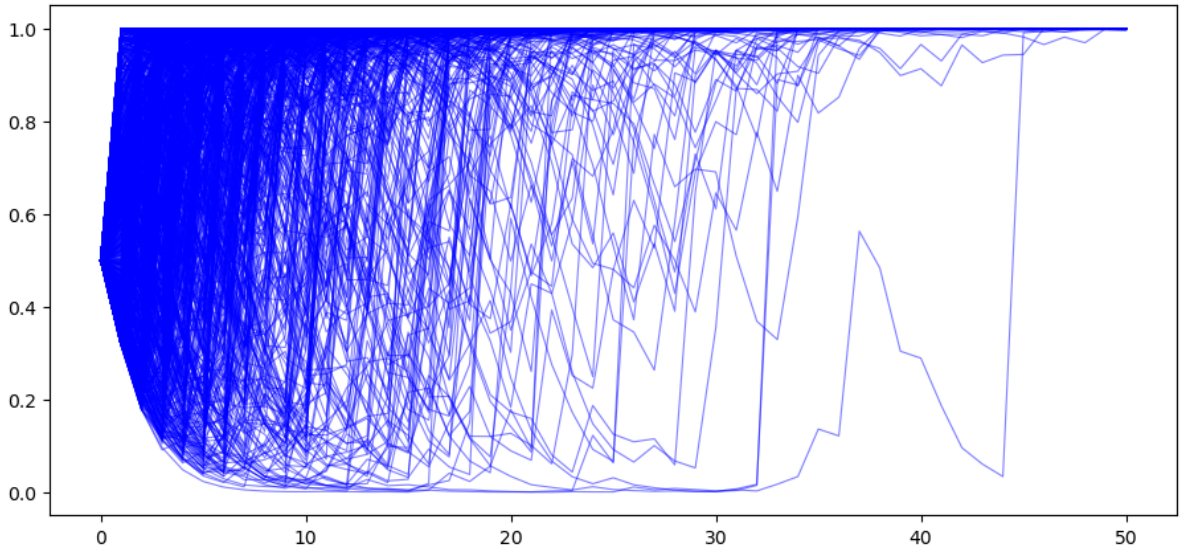
```
        # Update belief
        π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

        return π

    @njit
    def simulate_path(a, b, T=50):
        "Simulates a path of beliefs π with length T"

        π = np.empty(T+1)

        # initial condition
        π[0] = 0.5

        for t in range(1, T+1):
            π[t] = update(a, b, π[t-1])

        return π

    def simulate(a=1, b=1, T=50, N=200, display=True):
        "Simulates N paths of beliefs π with length T"

        π_paths = np.empty((N, T+1))
        if display:
            fig = plt.figure()

        for i in range(N):
            π_paths[i] = simulate_path(a=a, b=b, T=T)
            if display:
                plt.plot(range(T+1), π_paths[i], color='b', lw=0.8, alpha=0.5)

        if display:
            plt.show()

        return π_paths

    return simulate
```

```
simulate = function_factory()
```

We begin by generating N simulated $\{\pi_t\}$ paths with T periods when the sequence is truly IID draws from F. We set an initial prior $\pi_{-1} = .5$.

```
T = 50
```

```
# when nature selects F
π_paths_F = simulate(a=1, b=1, T=T, N=1000)
```

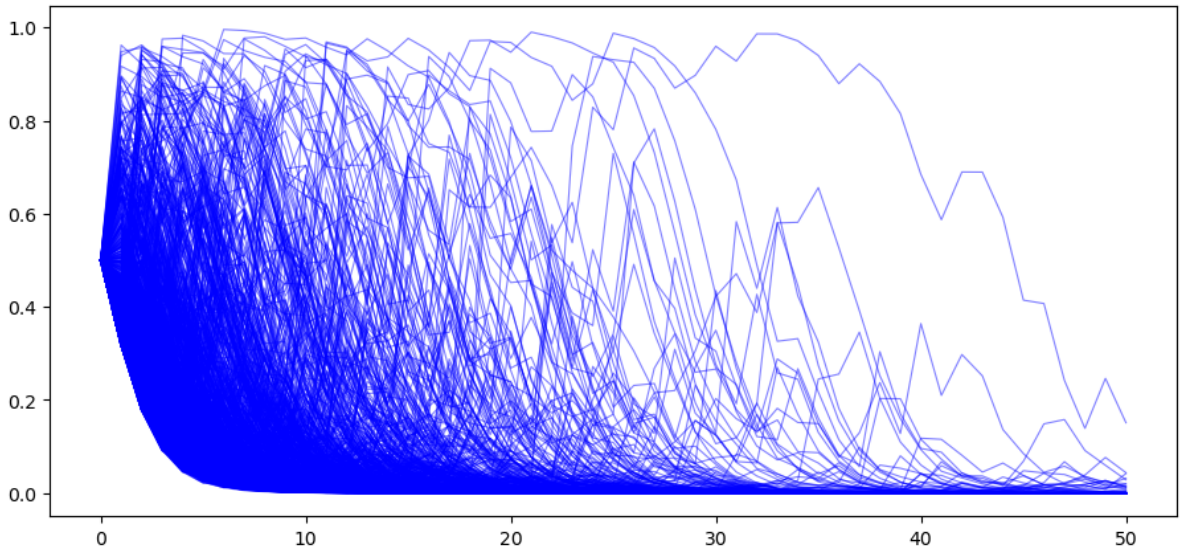


In the above example, for most paths $\pi_t \to 1$.

So Bayes' Law evidently eventually discovers the truth for most of our paths.

Next, we generate paths with T periods when the sequence is truly IID draws from G. Again, we set the initial prior $\pi_{-1} = .5$.

```
# when nature selects G
π_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```



In the above graph we observe that now most paths $\pi_t \to 0$.

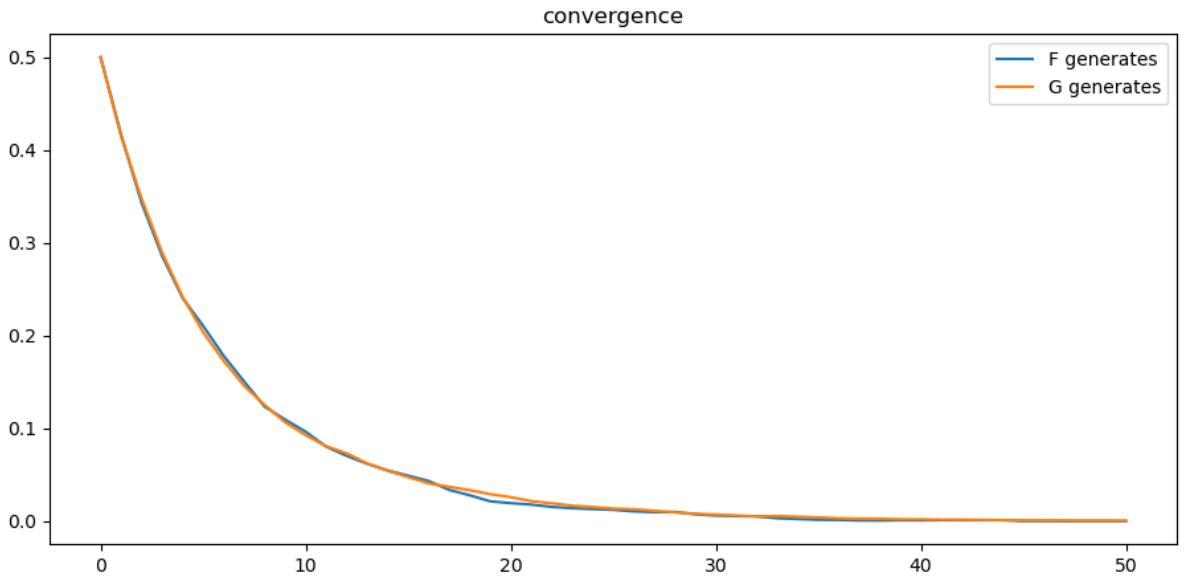
14.8.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of convergence of π_t to 0 when nature generates IID draws from G.

We do this by averaging across simulated paths of $\{\pi_t\}_{t=0}^T$.

Using N simulated π_t paths, we compute $1 - \sum_{i=1}^N \pi_{i,t}$ at each t when the data are generated as draws from F and compute $\sum_{i=1}^N \pi_{i,t}$ when the data are generated as draws from G.

```
plt.plot(range(T+1), 1 - np.mean(π_paths_F, 0), label='F generates')
plt.plot(range(T+1), np.mean(π_paths_G, 0), label='G generates')
plt.legend()
plt.title("convergence");
```



From the above graph, rates of convergence appear not to depend on whether F or G generates the data.

14.8.3 Graph of Ensemble Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$
\begin{aligned}
E\left[\frac{\pi_{t+1}}{\pi_t} \,\middle|\, q = a, \pi_t\right] &= E\left[\frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1-\pi_t)} \,\middle|\, q = a, \pi_t\right], \\
&= \int_0^1 \frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1-\pi_t)} a(w_{t+1})\, dw_{t+1}
\end{aligned}
$$

where $a = f, g$.

The following code approximates the integral above:

```
def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
```

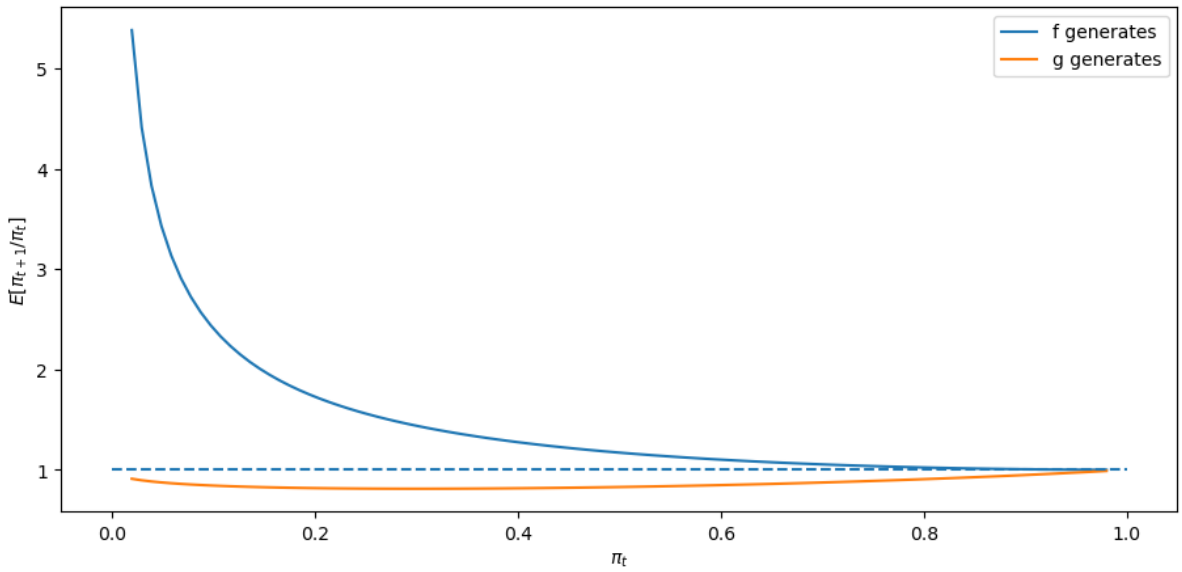
(continues on next page)

(continued from previous page)

```
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    l = lambda w: f(w) / g(w)
    integrand_f = lambda w, π: f(w) * l(w) / (π * l(w) + 1 - π)
    integrand_g = lambda w, π: g(w) * l(w) / (π * l(w) + 1 - π)

    π_grid = np.linspace(0.02, 0.98, 100)

    expected_rario = np.empty(len(π_grid))
    for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
        for i, π in enumerate(π_grid):
            expected_rario[i]= quad(inte, 0, 1, args=(π,))[0]
        plt.plot(π_grid, expected_rario, label=f"{q} generates")

    plt.hlines(1, 0, 1, linestyle="--")
    plt.xlabel("$π_t$")
    plt.ylabel("$E[\pi_{t+1}/\pi_t]$")
    plt.legend()

    plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio()
```

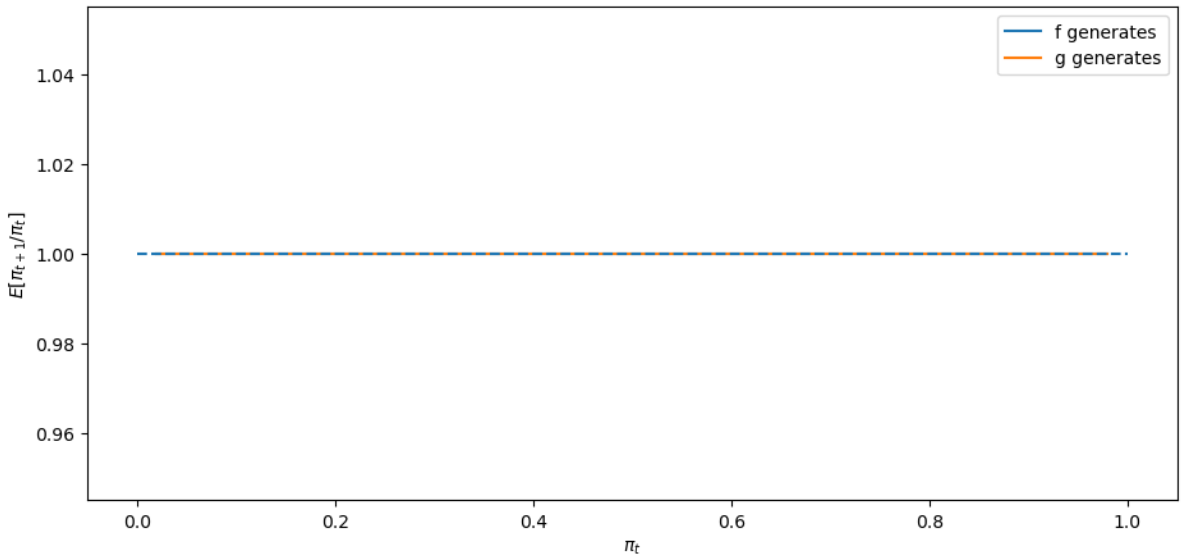


The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in whcih f and g are identical beta distributions, and $F_a = G_a = 3, F_b = G_b = 1.2$.

In a sense, here there is nothing to learn.

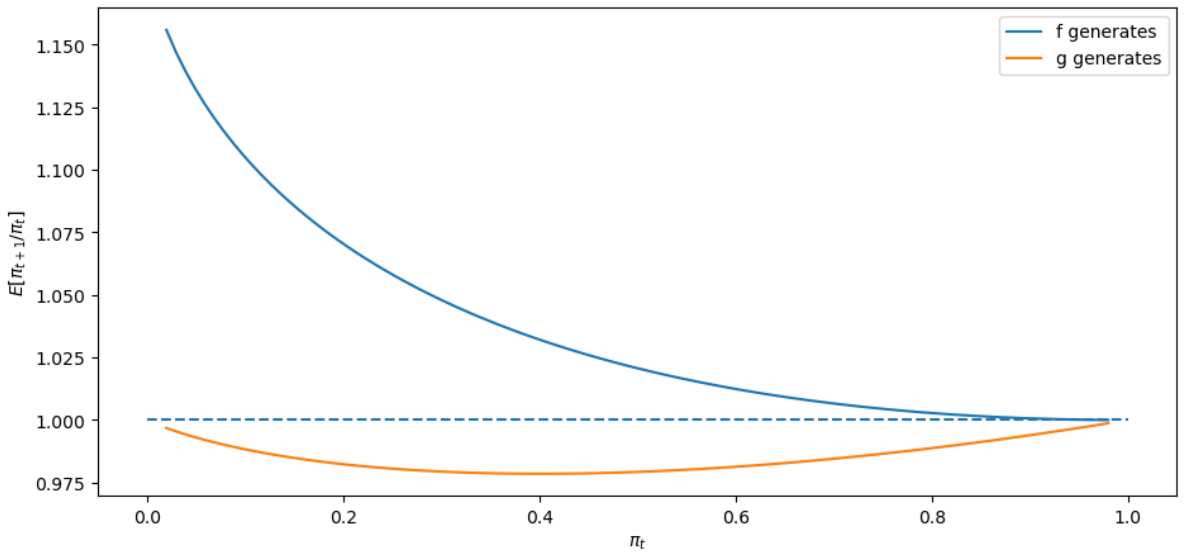
```
expected_ratio(F_a=3, F_b=1.2)
```



The above graph says that π_t is inert and remains at its initial value.

Finally, let's look at a case in which f and g are neither very different nor identical, in particular one in which $F_a = 2, F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)
```



14.9 Sequels

We'll apply and dig deeper into some of the ideas presented in this lecture:

- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* studies whether a World War II US Navy Captain's hunch that a (frequentist) decision rule that the Navy had told him to use was inferior to a sequential rule that Abraham Wald had not yet designed.

CHAPTER

FIFTEEN

LIKELIHOOD RATIO PROCESSES AND BAYESIAN LEARNING

15.1 Overview

This lecture describes the role that **likelihood ratio processes** play in **Bayesian learning**.

As in *this lecture*, we'll use a simple statistical setting from *this lecture*.

We'll focus on how a likelihood ratio process and a **prior** probability determine a **posterior** probability.

We'll derive a convenient recursion for today's posterior as a function of yesterday's posterior and today's multiplicative increment to a likelihood process.

We'll also present a useful generalization of that formula that represents today's posterior in terms of an initial prior and today's realization of the likelihood ratio process.

We'll study how, at least in our setting, a Bayesian eventually learns the probability distribution that generates the data, an outcome that rests on the asymptotic behavior of likelihood ratio processes studied in *this lecture*.

We'll also drill down into the psychology of our Bayesian learner and study dynamics under his subjective beliefs.

This lecture provides technical results that underly outcomes to be studied in this lecture and *this lecture* and *this lecture*.

We'll begin by loading some Python modules.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit, prange
from math import gamma
import pandas as pd

import seaborn as sns
colors = sns.color_palette()

@njit
def set_seed():
    np.random.seed(142857)
set_seed()
```

15.2 The Setting

We begin by reviewing the setting in *this lecture*, which we adopt here too.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from f or from g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g, but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define the key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^\infty$ is defined as

$$L(w^t) = \prod_{i=1}^t \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion* or *multiplicative decomposition*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [NP33].

We'll again deploy the following Python code from *this lecture* that evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from *some* probability distribution, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
```

(continues on next page)

(continued from previous page)

```
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

15.3 Likelihood Ratio Process and Bayes' Law

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f|w^t)$$

The likelihood ratio process is a principal actor in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{15.1}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (15.1)

```
@njit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (15.1) can be generalized by iterating on it and thereby deriving an expression for the time t posterior π_{t+1} as a function of the time 0 prior π_0 and the likelihood ratio process $L(w^{t+1})$ at time t.

To begin, notice that the updating rule

$$\pi_{t+1} = \frac{\pi_t \ell\left(w_{t+1}\right)}{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

implies

$$\begin{aligned}
\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}{\pi_t \ell\left(w_{t+1}\right)} \\
&= 1 - \frac{1}{\ell\left(w_{t+1}\right)} + \frac{1}{\ell\left(w_{t+1}\right)} \frac{1}{\pi_t}. \\
\Rightarrow \frac{1}{\pi_{t+1}} - 1 &= \frac{1}{\ell\left(w_{t+1}\right)} \left(\frac{1}{\pi_t} - 1\right).
\end{aligned}$$

Therefore

$$\frac{1}{\pi_{t+1}} - 1 = \frac{1}{\prod_{i=1}^{t+1} \ell\left(w_i\right)} \left(\frac{1}{\pi_0} - 1\right) = \frac{1}{L\left(w^{t+1}\right)} \left(\frac{1}{\pi_0} - 1\right).$$

Since $\pi_0 \in (0, 1)$ and $L\left(w^{t+1}\right) > 0$, we can verify that $\pi_{t+1} \in (0, 1)$.

After rearranging the preceding equation, we can express π_{t+1} as a function of $L\left(w^{t+1}\right)$, the likelihood ratio process at $t+1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{15.2}$$

Formula (15.2) generalizes formula (15.1).

Formula (15.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (15.2) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

Formula (15.2) is the foundation for the insight that, because of how the likelihood ratio process behaves as $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

To illustrate this insight, below we will plot graphs showing **one** simulated path of the likelihood ratio process L_t along with two paths of π_t that are associated with the *same* realization of the likelihood ratio process but *different* initial prior probabilities π_0.

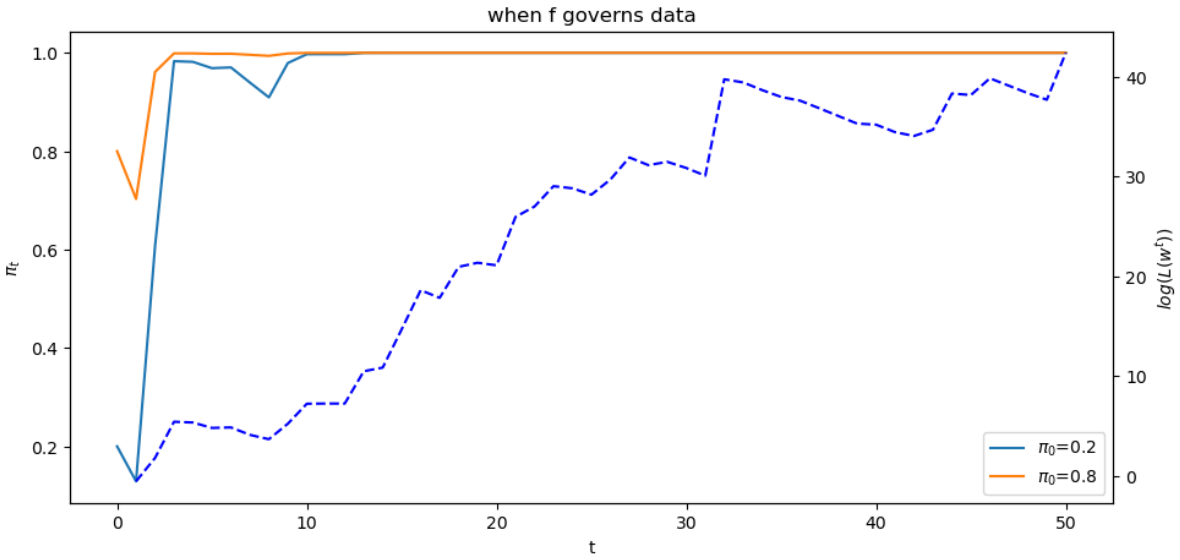
First, we tell Python two values of π_0.

```
π1, π2 = 0.2, 0.8
```

Next we generate paths of the likelihood ratio process L_t and the posterior π_t for a history of IID draws from density f.

```
T = l_arr_f.shape[1]
π_seq_f = np.empty((2, T+1))
π_seq_f[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_f[i, t+1] = update(π_seq_f[i, t], l_arr_f[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_f[i, :], label=f"$\pi_0$={π_seq_f[i, 0]}")

ax1.set_ylabel("$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when f governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_f[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



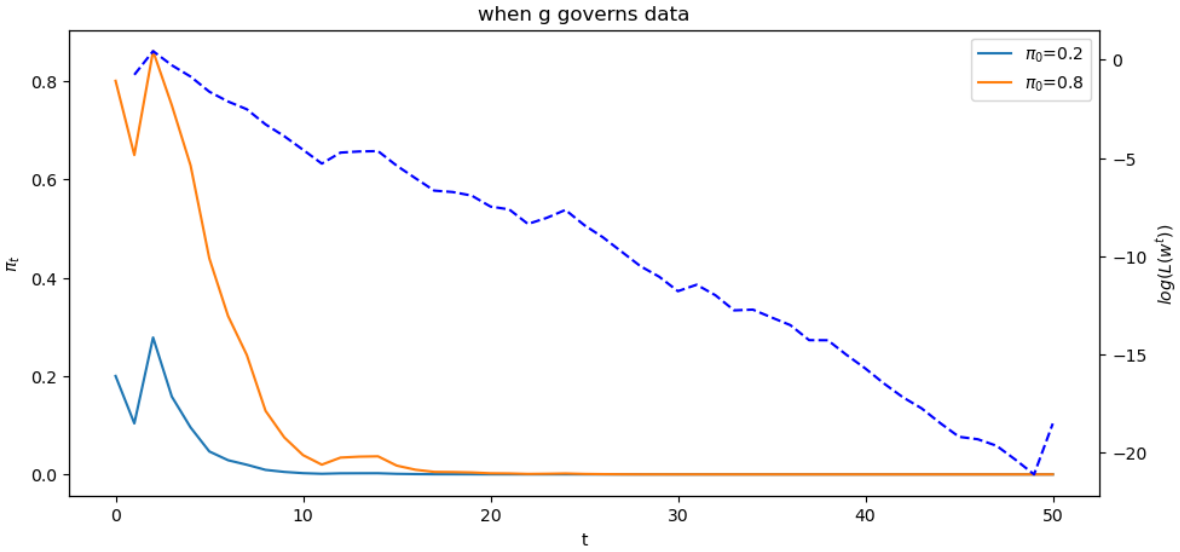
The dotted line in the graph above records the logarithm of the likelihood ratio process $\log L(w^t)$.

Please note that there are two different scales on the y axis.

Now let's study what happens when the history consists of IID draws from density g

```
T = l_arr_g.shape[1]
π_seq_g = np.empty((2, T+1))
π_seq_g[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_g[i, t+1] = update(π_seq_g[i, t], l_arr_g[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_g[i, :], label=f"$\pi_0$={π_seq_g[i, 0]}")

ax1.set_ylabel("$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when g governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_g[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



Below we offer Python code that verifies that nature chose permanently to draw from density f.

```
π_seq = np.empty((2, T+1))
π_seq[:, 0] = π1, π2

for i in range(2):
    πL = π_seq[i, 0] * l_seq_f[0, :]
    π_seq[i, 1:] = πL / (πL + 1 - π_seq[i, 0])
```

```
np.abs(π_seq - π_seq_f).max() < 1e-10
```

```
True
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (15.2) for a Bayesian's posteior probabilty that nature has drawn history w^t as repeated draws from density g.

15.4 Behavior of posterior probability $\{\pi_t\}$ under the subjective probability distribution

We'll end this lecture by briefly studying what our Baysian learner expects to learn under the subjective beliefs π_t cranked out by Bayes' law.

This will provide us with some perspective on our application of Bayes's law as a theory of learning.

As we shall see, at each time t, the Bayesian learner knows that he will be surprised.

But he expects that new information will not lead him to change his beliefs.

And it won't on average under his subjective beliefs.

We'll continue with our setting in which a McCall worker knows that successive draws of his wage are drawn from either F or G, but does not know which of these two distributions nature has drawn once-and-for-all before time 0.

We'll review and reiterate and rearrange some formulas that we have encountered above and in associated lectures.

The worker's initial beliefs induce a joint probability distribution over a potentially infinite sequence of draws $w_0, w_1, \ldots$.

Bayes' law is simply an application of laws of probability to compute the conditional distribution of the tth draw w_t conditional on $[w_0, \ldots, w_{t-1}]$.

After our worker puts a subjective probability π_{-1} on nature having selected distribution F, we have in effect assumes from the start that the decision maker **knows** the joint distribution for the process $\{w_t\}_{t=0}$.

We assume that the worker also knows the laws of probability theory.

A respectable view is that Bayes' law is less a theory of learning than a statement about the consequences of information inflows for a decision maker who thinks he knows the truth (i.e., a joint probability distribution) from the beginning.

15.4.1 Mechanical details again

At time 0 **before** drawing a wage offer, the worker attaches probability $\pi_{-1} \in (0, 1)$ to the distribution being F.

Before drawing a wage at time 0, the worker thus believes that the density of w_0 is

$$h(w_0; \pi_{-1}) = \pi_{-1} f(w_0) + (1 - \pi_{-1}) g(w_0).$$

Let $a \in \{f, g\}$ be an index that indicates whether nature chose permanently to draw from distribution f or from distribution g.

After drawing w_0, the worker uses Bayes' law to deduce that the posterior probability $\pi_0 = \text{Prob} a = f | w_0$ that the density is $f(w)$ is

$$\pi_0 = \frac{\pi_{-1} f(w_0)}{\pi_{-1} f(w_0) + (1 - \pi_{-1}) g(w_0)}.$$

More generally, after making the tth draw and having observed $w_t, w_{t-1}, \ldots, w_0$, the worker believes that the probability that w_{t+1} is being drawn from distribution F is

$$\pi_t = \pi_t(w_t | \pi_{t-1}) \equiv \frac{\pi_{t-1} f(w_t)/g(w_t)}{\pi_{t-1} f(w_t)/g(w_t) + (1 - \pi_{t-1})} \tag{15.3}$$

or

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}}$$

and that the density of w_{t+1} conditional on $w_t, w_{t-1}, \ldots, w_0$ is

$$h(w_{t+1}; \pi_t) = \pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1}).$$

Notice that

$$\begin{aligned} E(\pi_t | \pi_{t-1}) &= \int \Big[\frac{\pi_{t-1} f(w)}{\pi_{t-1} f(w) + (1 - \pi_{t-1}) g(w)} \Big] \Big[\pi_{t-1} f(w) + (1 - \pi_{t-1}) g(w) \Big] dw \\ &= \pi_{t-1} \int f(w) dw \\ &= \pi_{t-1}, \end{aligned}$$

so that the process π_t is a **martingale**.

Indeed, it is a **bounded martingale** because each π_t, being a probability, is between 0 and 1.

In the first line in the above string of equalities, the term in the first set of brackets is just π_t as a function of w_t, while the term in the second set of brackets is the density of w_t conditional on $w_{t-1}, \ldots, w_0$ or equivalently conditional on the *sufficient statistic* π_{t-1} for $w_{t-1}, \ldots, w_0$.

Notice that here we are computing $E(\pi_t | \pi_{t-1})$ under the **subjective** density described in the second term in brackets.

Because $\{\pi_t\}$ is a bounded martingale sequence, it follows from the **martingale convergence theorem** that π_t converges almost surely to a random variable in $[0, 1]$.

Practically, this means that probability one is attached to sample paths $\{\pi_t\}_{t=0}^{\infty}$ that converge.

According to the theorem, it different sample paths can converge to different limiting values.

Thus, let $\{\pi_t(\omega)\}_{t=0}^{\infty}$ denote a particular sample path indexed by a particular $\omega \in \Omega$.

We can think of nature as drawing an $\omega \in \Omega$ from a probability distribution ProbΩ and then generating a single realization (or *simulation*) $\{\pi_t(\omega)\}_{t=0}^{\infty}$ of the process.

The limit points of $\{\pi_t(\omega)\}_{t=0}^{\infty}$ as $t \to +\infty$ are realizations of a random variable that is swept out as we sample ω from Ω and construct repeated draws of $\{\pi_t(\omega)\}_{t=0}^{\infty}$.

By staring at law of motion (15.1) or (15.3) , we can figure out some things about the probability distribution of the limit points

$$\pi_\infty(\omega) = \lim_{t \to +\infty} \pi_t(\omega).$$

Evidently, since the likelihood ratio $\ell(w_t)$ differs from 1 when $f \neq g$, as we have assumed, the only possible fixed points of (15.3) are

$$\pi_\infty(\omega) = 1$$

and

$$\pi_\infty(\omega) = 0$$

Thus, for some realizations, $\lim_{\to +\infty} \pi_t(\omega) = 1$ while for other realizations, $\lim_{\to +\infty} \pi_t(\omega) = 0$.

Now let's remember that $\{\pi_t\}_{t=0}^{\infty}$ is a martingale and apply the law of iterated expectations.

The law of iterated expectations implies

$$E_t \pi_{t+j} = \pi_t$$

and in particular

$$E_{-1} \pi_{t+j} = \pi_{-1}.$$

Applying the above formula to π_∞, we obtain

$$E_{-1}\pi_\infty(\omega) = \pi_{-1}$$

where the mathematical expectation E_{-1} here is taken with respect to the probability measure $\text{Prob}(\Omega)$.

Since the only two values that $\pi_\infty(\omega)$ can take are 1 and 0, we know that for some $\lambda \in [0,1]$

$$\text{Prob}\Big(\pi_\infty(\omega) = 1\Big) = \lambda, \quad \text{Prob}\Big(\pi_\infty(\omega) = 0\Big) = 1 - \lambda$$

and consequently that

$$E_{-1}\pi_\infty(\omega) = \lambda \cdot 1 + (1-\lambda)\cdot 0 = \lambda$$

Combining this equation with equation (20), we deduce that the probability that $\text{Prob}(\Omega)$ attaches to $\pi_\infty(\omega)$ being 1 must be π_{-1}.

Thus, under the worker's subjective distribution, π_{-1} of the sample paths of $\{\pi_t\}$ will converge pointwise to 1 and $1-\pi_{-1}$ of the sample paths will converge pointwise to 0.

15.4.2 Some simulations

Let's watch the martingale convergence theorem at work in some simulations of our learning model under the worker's subjective distribution.

Let us simulate $\{\pi_t\}_{t=0}^T$, $\{w_t\}_{t=0}^T$ paths where for each $t \geq 0$, w_t is drawn from the subjective distribution

$$\pi_{t-1} f(w_t) + (1-\pi_{t-1})\, g(w_t)$$

We'll plot a large sample of paths.

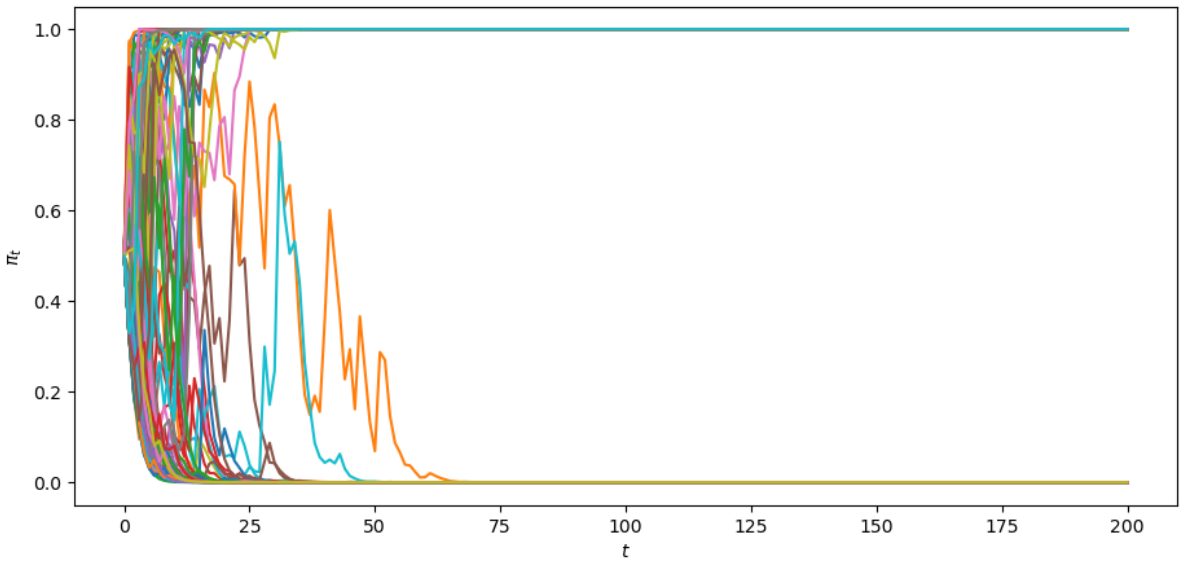
```
@njit
def martingale_simulate(π0, N=5000, T=200):

    π_path = np.empty((N,T+1))
    w_path = np.empty((N,T))
    π_path[:,0] = π0

    for n in range(N):
        π = π0
        for t in range(T):
            # draw w
            if np.random.rand() <= π:
                w = np.random.beta(F_a, F_b)
            else:
                w = np.random.beta(G_a, G_b)
            π = π*f(w)/g(w)/(π*f(w)/g(w) + 1 - π)
            π_path[n,t+1] = π
            w_path[n,t] = w

    return π_path, w_path

def fraction_0_1(π0, N, T, decimals):

    π_path, w_path = martingale_simulate(π0, N=N, T=T)
    values, counts = np.unique(np.round(π_path[:,-1], decimals=decimals), return_
↪counts=True)
```

(continues on next page)

(continued from previous page)

```
    return values, counts

def create_table(π0s, N=10000, T=500, decimals=2):

    outcomes = []
    for π0 in π0s:
        values, counts = fraction_0_1(π0, N=N, T=T, decimals=decimals)
        freq = counts/N
        outcomes.append(dict(zip(values, freq)))
    table = pd.DataFrame(outcomes).sort_index(axis=1).fillna(0)
    table.index = π0s
    return table



# simulate
T = 200
π0 = .5

π_path, w_path = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for i in range(100):
    ax.plot(range(T+1), π_path[i, :])

ax.set_xlabel('$t$')
ax.set_ylabel('$\pi_t$')
plt.show()
```



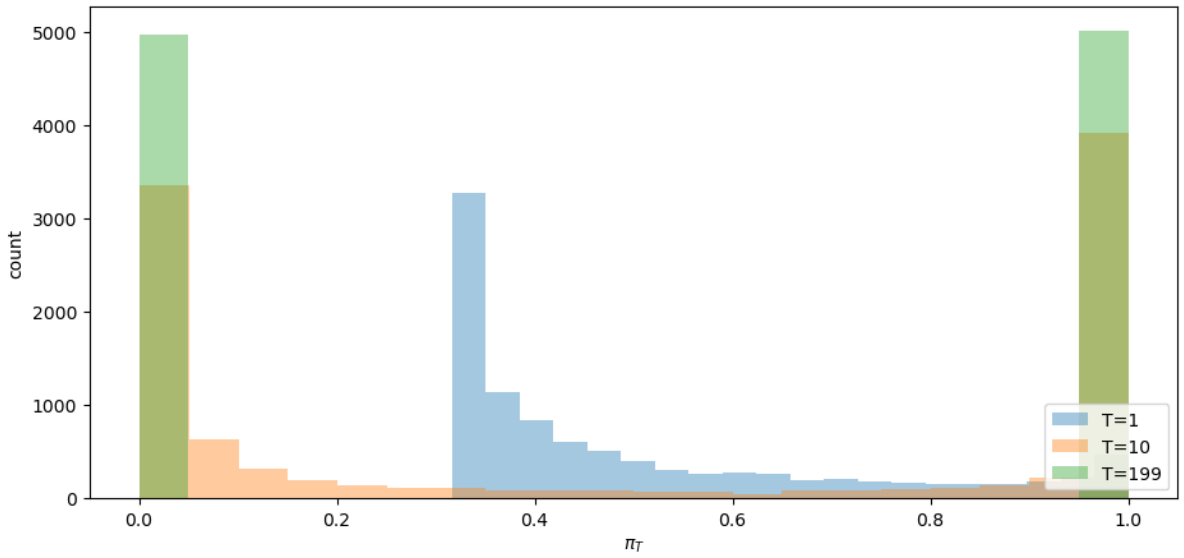
The above graph indicates that

- each of paths converges
- some of the paths converge to 1
- some of the paths converge to 0

- none of the paths converge to a limit point not equal to 0 or 1

Convergence actually occurs pretty fast, as the following graph of the cross-ensemble distribution of π_t for various small t's indicates.

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel('$\pi_T$')
ax.legend(loc='lower right')
plt.show()
```



Evidently, by $t = 199$, π_t has converged to either 0 or 1.

The fraction of paths that have converged to 1 is .5

The fractions of paths that have converged to 0 is also .5.

Does the fraction .5 ring a bell?

Yes, it does: it equals the value of $\pi_0 = .5$ that we used to generate each sequence in the ensemble.

So let's change π_0 to .3 and watch what happens to the distribution of the ensemble of π_t's for various t's.

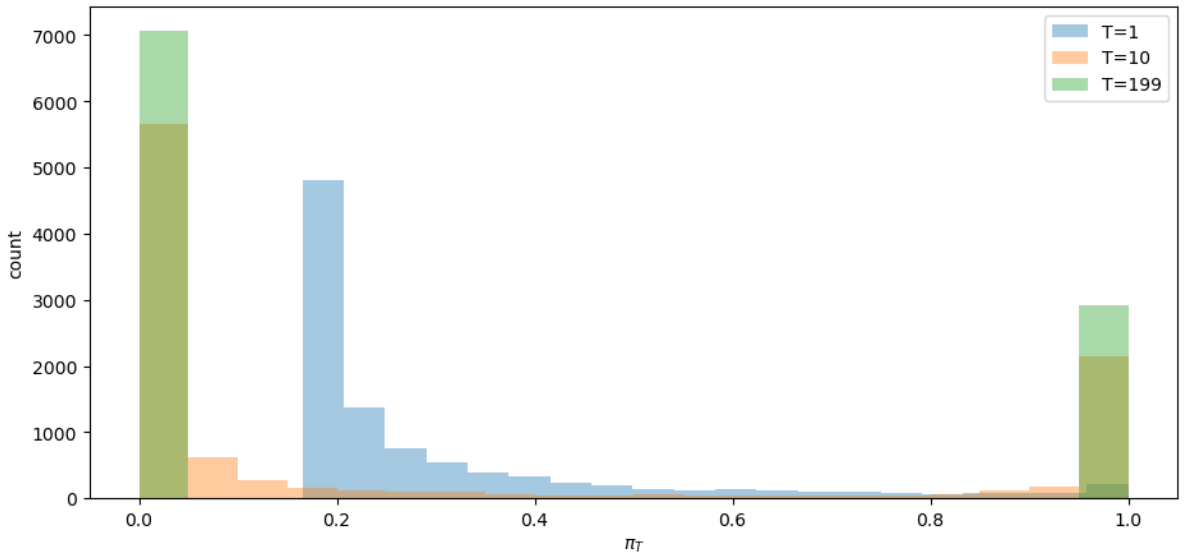
```
# simulate
T = 200
π0 = .3

π_path3, w_path3 = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path3[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel('$\pi_T$')
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc='upper right')
plt.show()
```

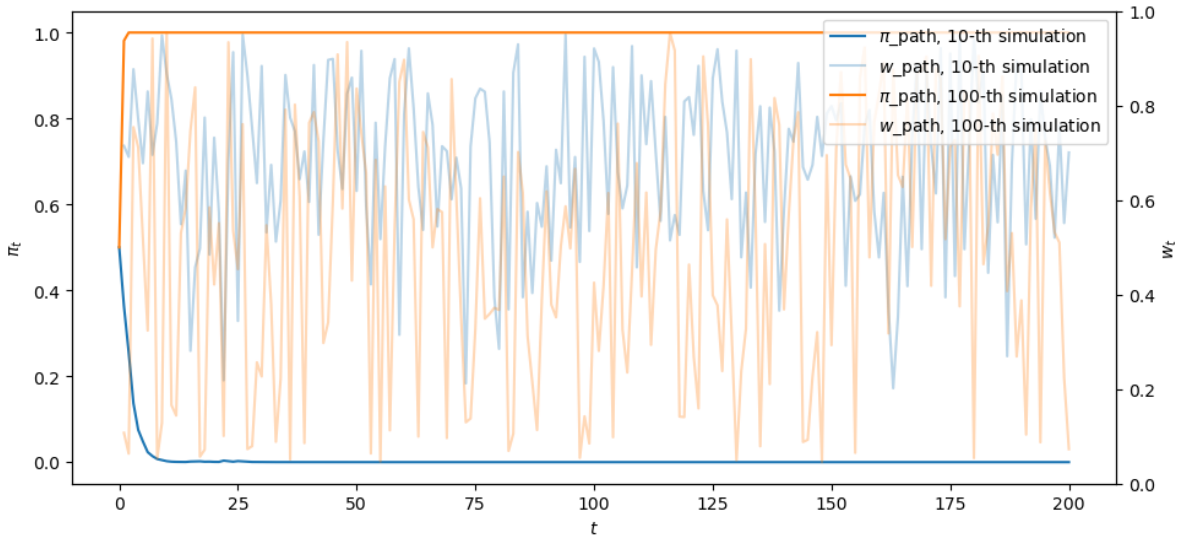


For the preceding ensemble that assumed $\pi_0 = .5$, the following graph shows two paths of w_t's and the π_t sequences that gave rise to them.

Notice that one of the paths involves systematically higher w_t's, outcomes that push π_t upward.

The luck of the draw early in a simulation push the subjective distribution to draw from F more frequently along a sample path, and this pushes π_t toward 0.

```
fig, ax = plt.subplots()
for i, j in enumerate([10, 100]):
    ax.plot(range(T+1), π_path[j,:], color=colors[i], label=f'$\pi$_path, {j}-th␣
↪simulation')
    ax.plot(range(1,T+1), w_path[j,:], color=colors[i], label=f'$w$_path, {j}-th␣
↪simulation', alpha=0.3)

ax.legend(loc='upper right')
ax.set_xlabel('$t$')
ax.set_ylabel('$\pi_t$')
ax2 = ax.twinx()
ax2.set_ylabel("$w_t$")
plt.show()
```



15.5 Initial Prior is Verified by Paths Drawn from Subjective Conditional Densities

Now let's use our Python code to generate a table that checks out our earlier claims about the probability distribution of the pointwise limits $\pi_\infty(\omega)$.

We'll use our simulations to generate a histogram of this distribution.

In the following table, the left column in bold face reports an assumed value of π_{-1}.

The second column reports the fraction of $N = 10000$ simulations for which π_t had converged to 0 at the terminal date $T = 500$ for each simulation.

The third column reports the fraction of $N = 10000$ simulations for which π_t had converged to 1 as the terminal date $T = 500$ for each simulation.

```
# create table
table = create_table(list(np.linspace(0,1,11)), N=10000, T=500)
table
```

	0.0	1.0
0.0	1.0000	0.0000
0.1	0.8929	0.1071
0.2	0.7994	0.2006
0.3	0.7014	0.2986
0.4	0.5939	0.4061
0.5	0.5038	0.4962
0.6	0.3982	0.6018
0.7	0.3092	0.6908
0.8	0.1963	0.8037
0.9	0.0963	0.9037
1.0	0.0000	1.0000

The fraction of simulations for which π_t had converged to 1 is indeed always close to π_{-1}, as anticipated.

15.6 Drilling Down a Little Bit

To understand how the local dynamics of π_t behaves, it is enlightening to consult the variance of π_t conditional on π_{t-1}.

Under the subjective distribution this conditional variance is defined as

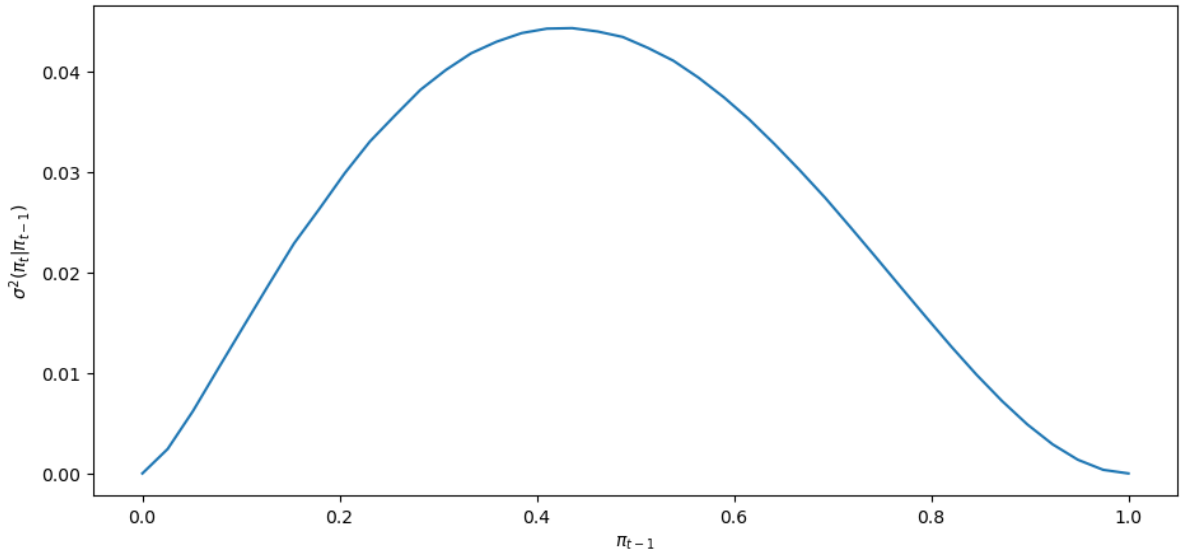
$$\sigma^2(\pi_t|\pi_{t-1}) = \int \Big[\frac{\pi_{t-1}f(w)}{\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)} - \pi_{t-1}\Big]^2 \Big[\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)\Big] dw$$

We can use a Monte Carlo simulation to approximate this conditional variance.

We approximate it for a grid of points $\pi_{t-1} \in [0, 1]$.

Then we'll plot it.

```
@njit
def compute_cond_var(pi, mc_size=int(1e6)):
    # create monte carlo draws
    mc_draws = np.zeros(mc_size)

    for i in prange(mc_size):
        if np.random.rand() <= pi:
            mc_draws[i] = np.random.beta(F_a, F_b)
        else:
            mc_draws[i] = np.random.beta(G_a, G_b)

    dev = pi*f(mc_draws)/(pi*f(mc_draws) + (1-pi)*g(mc_draws)) - pi
    return np.mean(dev**2)

pi_array = np.linspace(0, 1, 40)
cond_var_array = []

for pi in pi_array:
    cond_var_array.append(compute_cond_var(pi))

fig, ax = plt.subplots()
ax.plot(pi_array, cond_var_array)
ax.set_xlabel('$\pi_{t-1}$')
ax.set_ylabel('$\sigma^{2}(\pi_{t}\\vert \pi_{t-1})$')
plt.show()
```



The shape of the the conditional variance as a function of π_{t-1} is informative about the behavior of sample paths of $\{\pi_t\}$.

Notice how the conditional variance approaches 0 for π_{t-1} near either 0 or 1.

The conditional variance is nearly zero only when the agent is almost sure that w_t is drawn from F, or is almost sure it is drawn from G.

15.7 Sequels

This lecture has been devoted to building some useful infrastructure that will help us understand inferences that are the foundations of results described in this lecture and *this lecture* and *this lecture*.

CHAPTER

SIXTEEN

INCORRECT MODELS

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install numpyro jax
```

16.1 Overview

This is a sequel to *this quantecon lecture*.

We discuss two ways to create compound lottery and their consequences.

A compound lottery can be said to create a *mixture distribution*.

Our two ways of constructing a compound lottery will differ in their **timing**.

- in one, mixing between two possible probability distributions will occur once and all at the beginning of time
- in the other, mixing between the same two possible possible probability distributions will occur each period

The statistical setting is close but not identical to the problem studied in that quantecon lecture.

In that lecture, there were two i.i.d. processes that could possibly govern successive draws of a non-negative random variable W.

Nature decided once and for all whether to make a sequence of IID draws from either f or from g.

That lecture studied an agent who knew both f and g but did not know which distribution nature chose at time -1.

The agent represented that ignorance by assuming that nature had chosen f or g by flipping an unfair coin that put probability π_{-1} on probability distribution f.

That assumption allowed the agent to construct a subjective joint probability distribution over the random sequence $\{W_t\}_{t=0}^{\infty}$.

We studied how the agent would then use the laws of conditional probability and an observed history $w^t = \{w_s\}_{t=0}^{t}$ to form

$$\pi_t = E[\text{nature chose distribution} f | w^t], \quad t = 0, 1, 2, ...$$

However, in the setting of this lecture, that rule imputes to the agent an incorrect model.

The reason is that now the wage sequence is actually described by a different statistical model.

Thus, we change the *quantecon lecture* specification in the following way.

Now, **each period** $t \geq 0$, nature flips a possibly unfair coin that comes up f with probability α and g with probability $1 - \alpha$.

Thus, naturally perpetually draws from the **mixture distribution** with c.d.f.

$$H(w) = \alpha F(w) + (1-\alpha)G(w), \quad \alpha \in (0,1)$$

We'll study two agents who try to learn about the wage process, but who use different statistical models.

Both types of agent know f and g but neither knows α.

Our first type of agent erroneously thinks that at time -1 nature once and for all chose f or g and thereafter permanently draws from that distribution.

Our second type of agent knows, correctly, that nature mixes f and g with mixing probability $\alpha \in (0,1)$ each period, though the agent doesn't know the mixing parameter.

Our first type of agent applies the learning algorithm described in *this quantecon lecture*.

In the context of the statistical model that prevailed in that lecture, that was a good learning algorithm and it enabled the Bayesian learner eventually to learn the distribution that nature had drawn at time -1.

This is because the agent's statistical model was *correct* in the sense of being aligned with the data generating process.

But in the present context, our type 1 decision maker's model is incorrect because the model h that actually generates the data is neither f nor g and so is beyond the support of the models that the agent thinks are possible.

Nevertheless, we'll see that our first type of agent muddles through and eventually learns something interesting and useful, even though it is not *true*.

Instead, it turn out that our type 1 agent who is armed with a wrong statistical model ends up learning whichever probability distribution, f or g, is in a special sense *closest* to the h that actually generates the data.

We'll tell the sense in which it is closest.

Our second type of agent understands that nature mixes between f and g each period with a fixed mixing probability α.

But the agent doesn't know α.

The agent sets out to learn α using Bayes' law applied to his model.

His model is correct in the sense that it includes the actual data generating process h as a possible distribution.

In this lecture, we'll learn about

- how nature can *mix* between two distributions f and g to create a new distribution h.
- The Kullback-Leibler statistical divergence https://en.wikipedia.org/wiki/Kullback–Leibler_divergence that governs statistical learning under an incorrect statistical model
- A useful Python function `numpy.searchsorted` that, in conjunction with a uniform random number generator, can be used to sample from an arbitrary distribution

As usual, we'll start by importing some Python tools.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import vectorize, njit
from math import gamma
import pandas as pd
import scipy.stats as sp
from scipy.integrate import quad

import seaborn as sns
colors = sns.color_palette()
```

(continues on next page)

(continued from previous page)

```
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

import jax.numpy as jnp
from jax import random

np.random.seed(142857)

@njit
def set_seed():
    np.random.seed(142857)
set_seed()
```

Let's use Python to generate two beta distributions

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
@njit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

16.2 Sampling from Compound Lottery H

We implement two methods to draw samples from our mixture model $\alpha F + (1-\alpha)G$.

We'll generate samples using each of them and verify that they match well.

Here is pseudo code for a direct "method 1" for drawing from our compound lottery:

- Step one:
  - use the numpy.random.choice function to flip an unfair coin that selects distribution F with prob α and G with prob $1-\alpha$
- Step two:
  - draw from either F or G, as determined by the coin flip.
- Step three:
  - put the first two steps in a big loop and do them for each realization of w

Our second method uses a uniform distribution and the following fact that we also described and used in the quantecon lecture https://python.quantecon.org/prob_matrix.html:

- If a random variable X has c.d.f. $F(X)$, then a random variable $F^{-1}(U)$ also has c.d.f. $F(x)$, where U is a uniform random variable on $[0,1]$.

In other words, if $X \sim F(x)$ we can generate a random sample from F by drawing a random sample from a uniform distribution on $[0,1]$ and computing $F^{-1}(U)$.

We'll use this fact in conjunction with the `numpy.searchsorted` command to sample from H directly.

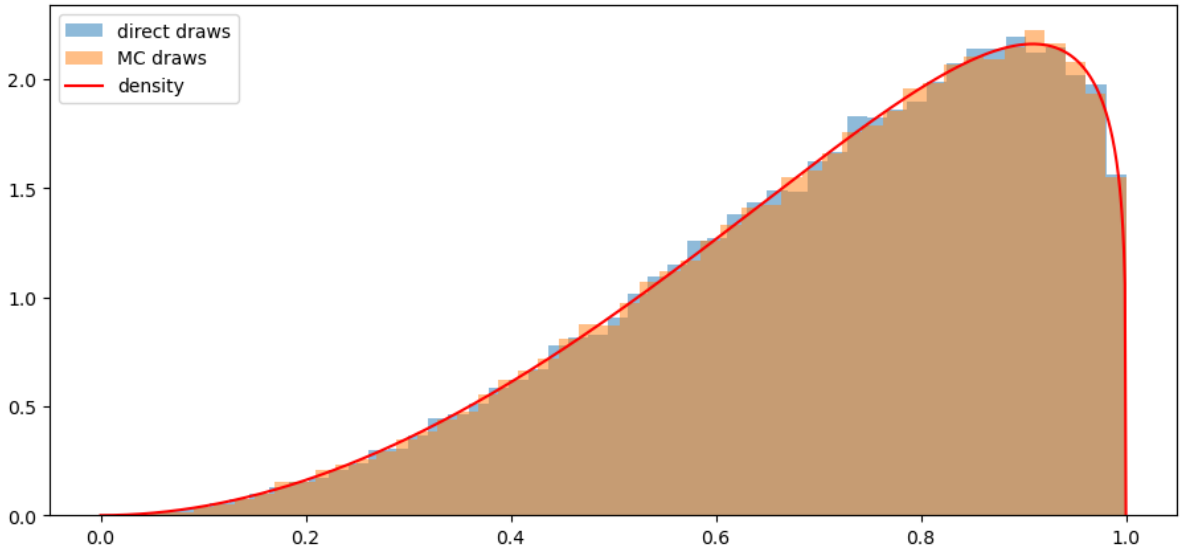
See https://numpy.org/doc/stable/reference/generated/numpy.searchsorted.html for the `searchsorted` function.

See the Mr. P Solver video on Monte Carlo simulation to see other applications of this powerful trick.

In the Python code below, we'll use both of our methods and confirm that each of them does a good job of sampling from our target mixture distribution.

```
@njit
def draw_lottery(p, N):
    "Draw from the compound lottery directly."

    draws = []
    for i in range(0, N):
        if np.random.rand()<=p:
            draws.append(np.random.beta(F_a, F_b))
        else:
            draws.append(np.random.beta(G_a, G_b))

    return np.array(draws)

def draw_lottery_MC(p, N):
    "Draw from the compound lottery using the Monte Carlo trick."

    xs = np.linspace(1e-8,1-(1e-8),10000)
    CDF = p*sp.beta.cdf(xs, F_a, F_b) + (1-p)*sp.beta.cdf(xs, G_a, G_b)

    Us = np.random.rand(N)
    draws = xs[np.searchsorted(CDF[:-1], Us)]
    return draws
```

```
# verify
N = 100000
α = 0.0

sample1 = draw_lottery(α, N)
sample2 = draw_lottery_MC(α, N)

# plot draws and density function
plt.hist(sample1, 50, density=True, alpha=0.5, label='direct draws')
plt.hist(sample2, 50, density=True, alpha=0.5, label='MC draws')

xs = np.linspace(0,1,1000)
plt.plot(xs, α*f(xs)+(1-α)*g(xs), color='red', label='density')

plt.legend()
plt.show()
```




```
# %%timeit    # compare speed
# sample1 = draw_lottery(α, N=int(1e6))
```

```
# %%timeit
# sample2 = draw_lottery_MC(α, N=int(1e6))
```

Note: With numba acceleration the first method is actually only slightly slower than the second when we generated 1,000,000 samples.

16.3 Type 1 Agent

We'll now study what our type 1 agent learns

Remember that our type 1 agent uses the wrong statistical model, thinking that nature mixed between f and g once and for all at time -1.

The type 1 agent thus uses the learning algorithm studied in *this quantecon lecture*.

We'll briefly review that learning algorithm now.

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f|w^t)$$

The likelihood ratio process plays a principal role in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{16.1}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (16.1)

```
@njit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (16.1) can be generalized by iterating on it and thereby deriving an expression for the time t posterior π_{t+1} as a function of the time 0 prior π_0 and the likelihood ratio process $L(w^{t+1})$ at time t.

To begin, notice that the updating rule

$$\pi_{t+1} = \frac{\pi_t \ell\left(w_{t+1}\right)}{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

implies

$$\begin{aligned}
\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell\left(w_{t+1}\right) + \left(1 - \pi_t\right)}{\pi_t \ell\left(w_{t+1}\right)} \\
&= 1 - \frac{1}{\ell\left(w_{t+1}\right)} + \frac{1}{\ell\left(w_{t+1}\right)} \frac{1}{\pi_t}.
\end{aligned}$$

$$\Rightarrow \frac{1}{\pi_{t+1}} - 1 = \frac{1}{\ell\left(w_{t+1}\right)} \left(\frac{1}{\pi_t} - 1\right).$$

Therefore

$$\frac{1}{\pi_{t+1}} - 1 = \frac{1}{\prod_{i=1}^{t+1} \ell\left(w_i\right)} \left(\frac{1}{\pi_0} - 1\right) = \frac{1}{L\left(w^{t+1}\right)} \left(\frac{1}{\pi_0} - 1\right).$$

Since $\pi_0 \in (0,1)$ and $L\left(w^{t+1}\right) > 0$, we can verify that $\pi_{t+1} \in (0,1)$.

After rearranging the preceding equation, we can express π_{t+1} as a function of $L\left(w^{t+1}\right)$, the likelihood ratio process at $t+1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{16.2}$$

Formula (16.2) generalizes formula (16.1).

Formula (16.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

16.4 What a type 1 Agent Learns when Mixture H Generates Data

We now study what happens when the mixture distribution $h;\alpha$ truly generated the data each period.

A submartingale or supermartingale continues to describe π_t

It raises its ugly head and causes π_t to converge either to 0 or to 1.

This is true even though in truth nature always mixes between f and g.

After verifying that claim about possible limit points of π_t sequences, we'll drill down and study what fundamental force determines the limiting value of π_t.

Let's set a value of α and then watch how π_t evolves.

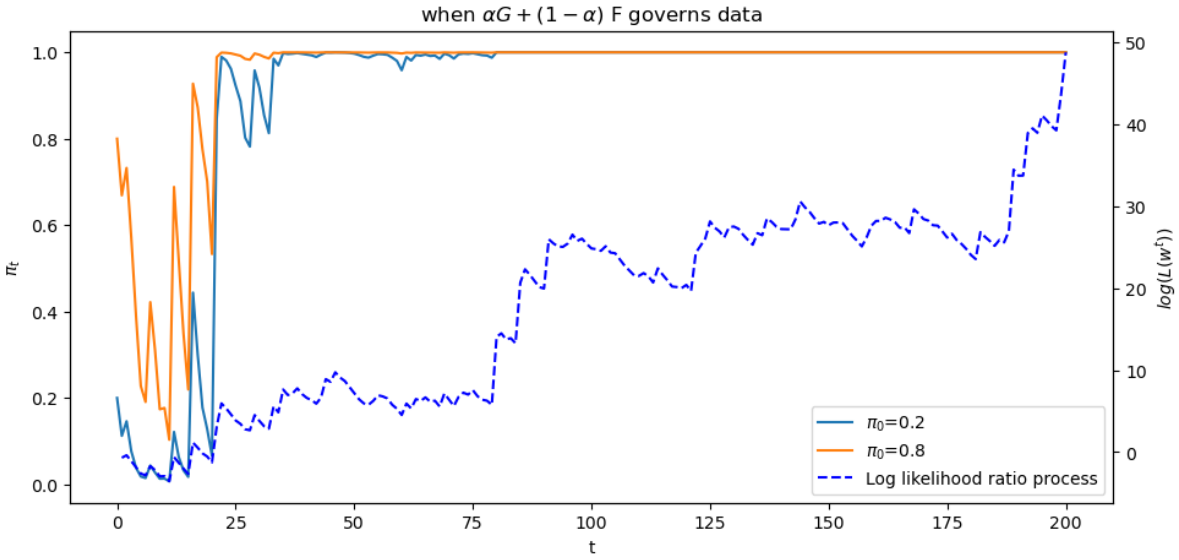
```
def simulate_mixed(α, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix, when the true density is mixed h;a
    """

    w_s = draw_lottery(α, N*T).reshape(N, T)
    l_arr = f(w_s) / g(w_s)

    return l_arr

def plot_π_seq(α, π1=0.2, π2=0.8, T=200):
    """
    Compute and plot π_seq and the log likelihood ratio process
    when the mixed distribution governs the data.
    """

    l_arr_mixed = simulate_mixed(α, T=T, N=50)
    l_seq_mixed = np.cumprod(l_arr_mixed, axis=1)

    T = l_arr_mixed.shape[1]
    π_seq_mixed = np.empty((2, T+1))
    π_seq_mixed[:, 0] = π1, π2

    for t in range(T):
        for i in range(2):
            π_seq_mixed[i, t+1] = update(π_seq_mixed[i, t], l_arr_mixed[0, t])

    # plot
    fig, ax1 = plt.subplots()
    for i in range(2):
```

(continues on next page)

(continued from previous page)

```
        ax1.plot(range(T+1), π_seq_mixed[i, :], label=f"$\pi_0$={π_seq_mixed[i, 0]}")

    ax1.plot(np.nan, np.nan,  '--', color='b', label='Log likelihood ratio process')
    ax1.set_ylabel("$\pi_t$")
    ax1.set_xlabel("t")
    ax1.legend()
    ax1.set_title("when $\\alpha G + (1-\\alpha)$ F governs data")

    ax2 = ax1.twinx()
    ax2.plot(range(1, T+1), np.log(l_seq_mixed[0, :]), '--', color='b')
    ax2.set_ylabel("$log(L(w^{t}))$")

    plt.show()
```

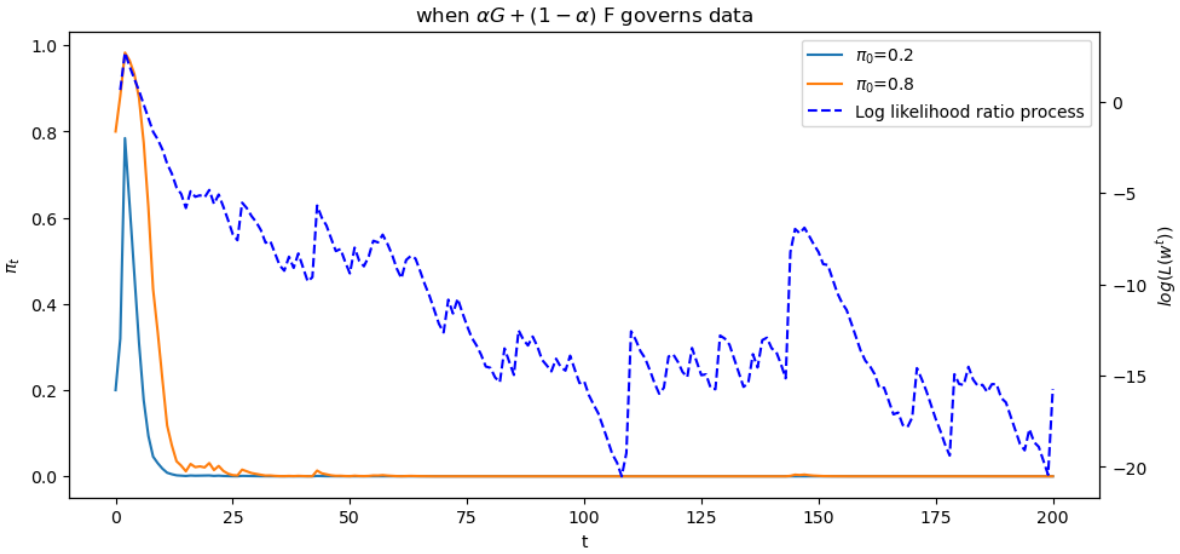
```
plot_π_seq(α = 0.6)
```



The above graph shows a sample path of the log likelihood ratio process as the blue dotted line, together with sample paths of π_t that start from two distinct initial conditions.

Let's see what happens when we change α

```
plot_π_seq(α = 0.2)
```



Evidently, α is having a big effect on the destination of π_t as $t \to +\infty$

16.5 Kullback-Leibler Divergence Governs Limit of π_t

To understand what determines whether the limit point of π_t is 0 or 1 and how the answer depends on the true value of the mixing probability $\alpha \in (0, 1)$ that generates

$$h(w) \equiv h(w|\alpha) = \alpha f(w) + (1 - \alpha) g(w)$$

we shall compute the following two Kullback-Leibler divergences

$$KL_g(\alpha) = \int \log\left(\frac{g(w)}{h(w)}\right) h(w) dw$$

and

$$KL_f(\alpha) = \int \log\left(\frac{f(w)}{h(w)}\right) h(w) dw$$

We shall plot both of these functions against α as we use α to vary $h(w) = h(w|\alpha)$.

The limit of π_t is determined by

$$\min_{f,g} \{KL_g, KL_f\}$$

The only possible limits are 0 and 1.

As $\to +\infty$, π_t goes to one if and only if $KL_f < KL_g$

```
@vectorize
def KL_g(α):
    "Compute the KL divergence between g and h."
    err = 1e-8                          # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
```

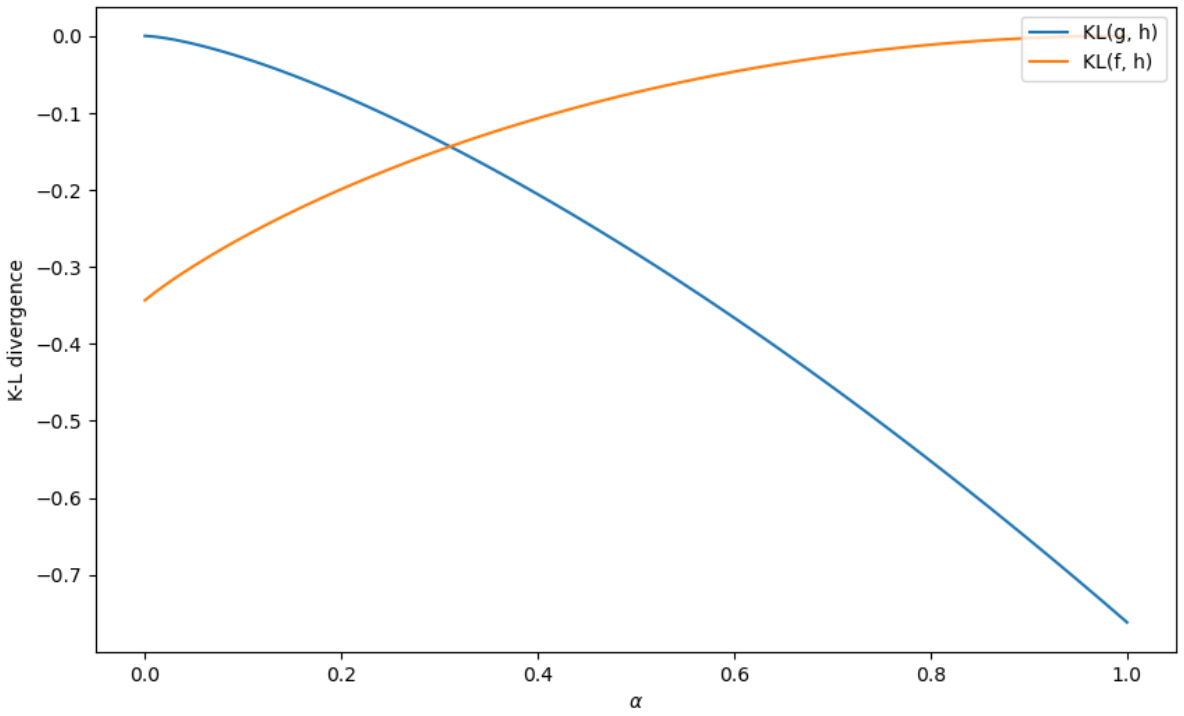
(continues on next page)

(continued from previous page)

```
    return np.sum(np.log(gs/hs)*hs)/10000

@vectorize
def KL_f(α):
    "Compute the KL divergence between f and h."
    err = 1e-8                          # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
    return np.sum(np.log(fs/hs)*hs)/10000


# compute KL using quad in Scipy
def KL_g_quad(α):
    "Compute the KL divergence between g and h using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: np.log(g(x)/h(x))*h(x), 0, 1)[0]

def KL_f_quad(α):
    "Compute the KL divergence between f and h using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: np.log(f(x)/h(x))*h(x), 0, 1)[0]

# vectorize
KL_g_quad_v = np.vectorize(KL_g_quad)
KL_f_quad_v = np.vectorize(KL_f_quad)


# Let us find the limit point
def π_lim(α, T=5000, π_0=0.4):
    "Find limit of π sequence."
    π_seq = np.zeros(T+1)
    π_seq[0] = π_0
    l_arr = simulate_mixed(α, T, N=1)[0]

    for t in range(T):
        π_seq[t+1] = update(π_seq[t], l_arr[t])
    return π_seq[-1]

π_lim_v = np.vectorize(π_lim)
```

Let us first plot the KL divergences $KL_g(\alpha), KL_f(\alpha)$ for each α.

```
α_arr = np.linspace(0, 1, 100)
KL_g_arr = KL_g(α_arr)
KL_f_arr = KL_f(α_arr)

fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
ax.set_ylabel('K-L divergence')
ax.set_xlabel(r'$\alpha$')

ax.legend(loc='upper right')
plt.show()
```




```
# # using Scipy to compute KL divergence

# α_arr = np.linspace(0, 1, 100)
# KL_g_arr = KL_g_quad_v(α_arr)
# KL_f_arr = KL_f_quad_v(α_arr)

# fig, ax = plt.subplots(1, figsize=[10, 6])

# ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
# ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
# ax.set_ylabel('K-L divergence')

# ax.legend(loc='upper right')
# plt.show()
```

Let's compute an α for which the KL divergence between h and g is the same as that between h and f.

```
# where KL_f = KL_g
α_arr[np.argmin(np.abs(KL_g_arr-KL_f_arr))]
```

```
0.31313131313131315
```

We can compute and plot the convergence point π_∞ for each α to verify that the convergence is indeed governed by the KL divergence.

The blue circles show the limiting values of π_t that simulations discover for different values of α recorded on the x axis.

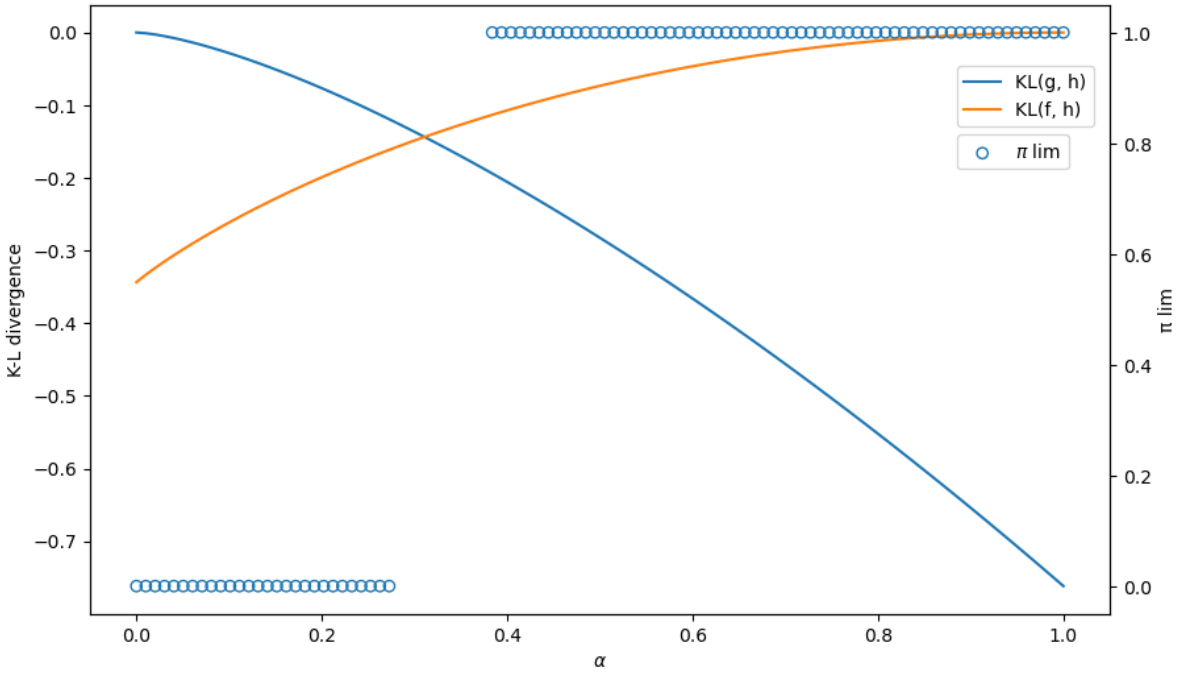
Thus, the graph below confirms how a minimum KL divergence governs what our type 1 agent eventually learns.

```
α_arr_x = α_arr[(α_arr<0.28)|(α_arr>0.38)]
π_lim_arr = π_lim_v(α_arr_x)
```

(continues on next page)

(continued from previous page)

```
# plot
fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(g, h)')
ax.plot(α_arr, KL_f_arr, label='KL(f, h)')
ax.set_ylabel('K-L divergence')
ax.set_xlabel(r'$\alpha$')

# plot KL
ax2 = ax.twinx()
# plot limit point
ax2.scatter(α_arr_x, π_lim_arr, facecolors='none', edgecolors='tab:blue', label='$\pi
↪$ lim')
ax2.set_ylabel('π lim')

ax.legend(loc=[0.85, 0.8])
ax2.legend(loc=[0.85, 0.73])
plt.show()
```



Evidently, our type 1 learner who applies Bayes' law to his misspecified set of statistical models eventually learns an approximating model that is as close as possible to the true model, as measured by its Kullback-Leibler divergence.

16.6 Type 2 Agent

We now describe how our type 2 agent formulates his learning problem and what he eventually learns.

Our type 2 agent understands the correct statistical model but acknowledges does not know α.

We apply Bayes law to deduce an algorithm for learning α under the assumption that the agent knows that

$$h(w) = h(w|\alpha)$$

but does not know α.

We'll assume that the person starts out with a prior probabilty $\pi_0(\alpha)$ on $\alpha \in (0,1)$ where the prior has one of the forms that we deployed in *this quantecon lecture*.

We'll fire up `numpyro` and apply it to the present situation.

Bayes' law now takes the form

$$\pi_{t+1}(\alpha) = \frac{h(w_{t+1}|\alpha)\pi_t(\alpha)}{\int h(w_{t+1}|\hat{\alpha})\pi_t(\hat{\alpha})d\hat{\alpha}}$$

We'll use numpyro to approximate this equation.

We'll create graphs of the posterior $\pi_t(\alpha)$ as $t \to +\infty$ corresponding to ones presented in the quantecon lecture https://python.quantecon.org/bayes_nonconj.html.

We anticipate that a posterior distribution will collapse around the true α as $t \to +\infty$.

Let us try a uniform prior first.

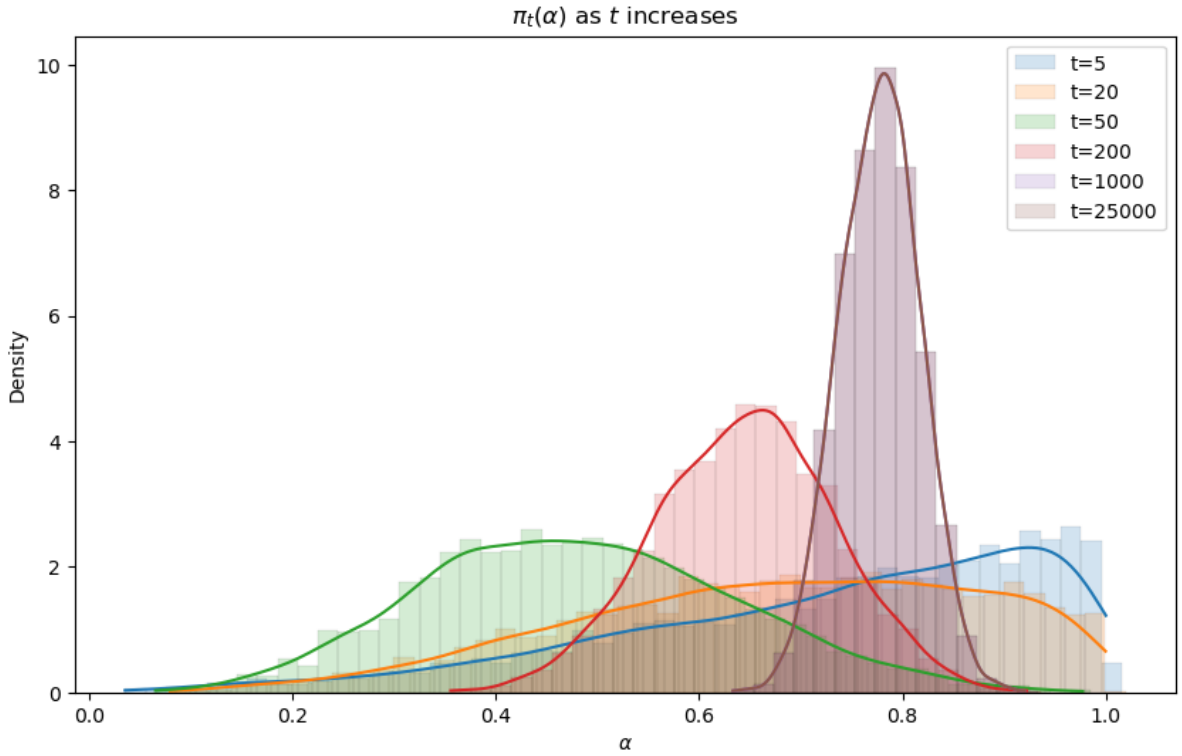
We use the `Mixture` class in Numpyro to construct the likelihood function.

```
α = 0.8

# simulate data with true α
data = draw_lottery(α, 1000)
sizes = [5, 20, 50, 200, 1000, 25000]

def model(w):
    α = numpyro.sample('α', dist.Uniform(low=0.0, high=1.0))

    y_samp = numpyro.sample('w',
        dist.Mixture(dist.Categorical(jnp.array([α, 1-α])), [dist.Beta(F_a, F_b),␣
↪dist.Beta(G_a, G_b)]), obs=w)

def MCMC_run(ws):
    "Compute posterior using MCMC with observed ws"

    kernal = NUTS(model)
    mcmc = MCMC(kernal, num_samples=5000, num_warmup=1000, progress_bar=False)

    mcmc.run(rng_key=random.PRNGKey(142857), w=jnp.array(ws))
    sample = mcmc.get_samples()
    return sample['α']
```

The following code generates the graph below that displays Bayesian posteriors for α at various history lengths.

```
fig, ax = plt.subplots(figsize=(10, 6))

for i in range(len(sizes)):
    sample = MCMC_run(data[:sizes[i]])
    sns.histplot(
        data=sample, kde=True, stat='density', alpha=0.2, ax=ax,
        color=colors[i], binwidth=0.02, linewidth=0.05, label=f't={sizes[i]}'
    )
ax.set_title('$\pi_t(\\alpha)$ as $t$ increases')
ax.legend()
ax.set_xlabel('$\\alpha$')
plt.show()
```

```
CUDA backend failed to initialize: Found CUDA version 12010, but JAX was built␣
↪against version 12020, which is newer. The copy of CUDA that is installed must␣
↪be at least as new as the version against which JAX was built. (Set TF_CPP_MIN_
↪LOG_LEVEL=0 and rerun for more info.)
```



Evidently, the Bayesian posterior narrows in on the true value $\alpha = .8$ of the mixing parameter as the length of a history of observations grows.

16.7 Concluding Remarks

Our type 1 person deploys an incorrect statistical model.

He believes that either f or g generated the w process, but just doesn't know which one.

That is wrong because nature is actually mixing each period with mixing probability α.

Our type 1 agent eventually believes that either f or g generated the w sequence, the outcome being determined by the model, either f or g, whose KL divergence relative to h is smaller.

Our type 2 agent has a different statistical model, one that is correctly specified.

He knows the parametric form of the statistical model but not the mixing parameter α.

He knows that he does not know it.

But by using Bayes' law in conjunction with his statistical model and a history of data, he eventually acquires a more and more accurate inference about α.

This little laboratory exhibits some important general principles that govern outcomes of Bayesian learning of misspecified models.

Thus, the following situation prevails quite generally in empirical work.

A scientist approaches the data with a manifold S of statistical models $s(X|\theta)$, where s is a probability distribution over a random vector X, $\theta \in \Theta$ is a vector of parameters, and Θ indexes the manifold of models.

The scientist with observations that he interprests as realizations x of the random vector X wants to solve an **inverse problem** of somehow *inverting* $s(x|\theta)$ to infer θ from x.

But the scientist's model is misspecified, being only an approximation to an unknown model h that nature uses to generate X.

If the scientist uses Bayes' law or a related likelihood-based method to infer θ, it occurs quite generally that for large sample sizes the inverse problem infers a θ that minimizes the KL divergence of the scientist's model s relative to nature's model h.

CHAPTER

SEVENTEEN

BAYESIAN VERSUS FREQUENTIST DECISION RULES

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install interpolation
```

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
from scipy.optimize import minimize
```

```
/opt/conda/envs/quantecon/lib/python3.11/site-packages/numba/core/decorators.
↪py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated. Please see␣
↪the documentation at: https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-generated-jit for more information and advice on␣
↪a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

17.1 Overview

This lecture follows up on ideas presented in the following lectures:

- *A Problem that Stumped Milton Friedman*
- *Exchangeability and Bayesian Updating*
- *Likelihood Ratio Processes*

In *A Problem that Stumped Milton Friedman* we described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had instructed the Captain to use a decision rule for quality control that the Captain suspected could be dominated by a better rule.

(The Navy had ordered the Captain to use an instance of a **frequentist decision rule**.)

Milton Friedman recognized the Captain's conjecture as posing a challenging statistical problem that he and other members of the US Government's Statistical Research Group at Columbia University proceeded to try to solve.

One of the members of the group, the great mathematician Abraham Wald, soon solved the problem.

A good way to formulate the problem is to use some ideas from Bayesian statistics that we describe in this lecture *Exchangeability and Bayesian Updating* and in this lecture *Likelihood Ratio Processes*, which describes the link between Bayesian updating and likelihood ratio processes.

The present lecture uses Python to generate simulations that evaluate expected losses under **frequentist** and **Bayesian** decision rules for an instance of the Navy Captain's decision problem.

The simulations validate the Navy Captain's hunch that there is a better rule than the one the Navy had ordered him to use.

17.2 Setup

To formalize the problem of the Navy Captain whose questions posed the problem that Milton Friedman and Allan Wallis handed over to Abraham Wald, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z from a probability distribution that he does not completely understand. He knows that two probability distributions are possible, f_0 and f_1, and that which ever distribution it is remains fixed over time. The decision maker believes that before the beginning of time, nature once and for all selected either f_0 or f_1 and that the probability that it selected f_0 is probability π^*.
- The decision maker observes a sample $\{z_i\}_{i=0}^t$ from the the distribution chosen by nature.

The decision maker wants to decide which distribution actually governs Z and is worried by two types of errors and the losses that they impose on him.

- a loss $\bar{L}_1$ from a **type I error** that occurs when he decides that $f = f_1$ when actually $f = f_0$
- a loss $\bar{L}_0$ from a **type II error** that occurs when he decides that $f = f_0$ when actually $f = f_1$

The decision maker pays a cost c for drawing another z

We mainly borrow parameters from the quantecon lecture *A Problem that Stumped Milton Friedman* except that we increase both $\bar{L}_0$ and $\bar{L}_1$ from 25 to 100 to encourage the frequentist Navy Captain to take more draws before deciding.

We set the cost c of taking one more draw at 1.25.

We set the probability distributions f_0 and f_1 to be beta distributions with $a_0 = b_0 = 1$, $a_1 = 3$, and $b_1 = 1.2$, respectively.

Below is some Python code that sets up these objects.

```
@njit
def p(x, a, b):
    "Beta distribution."

    r = gamma(a + b) / (gamma(a) * gamma(b))

    return r * x**(a-1) * (1 - x)**(b-1)
```

We start with defining a `jitclass` that stores parameters and functions we need to solve problems for both the Bayesian and frequentist Navy Captains.

```
wf_data = [
    ('c', float64),           # unemployment compensation
    ('a0', float64),          # parameters of beta distribution
    ('b0', float64),
    ('a1', float64),
    ('b1', float64),
    ('L0', float64),          # cost of selecting f0 when f1 is true
    ('L1', float64),          # cost of selecting f1 when f0 is true
    ('π_grid', float64[:]),   # grid of beliefs π
    ('π_grid_size', int64),
    ('mc_size', int64),       # size of Monto Carlo simulation
    ('z0', float64[:]),       # sequence of random values
    ('z1', float64[:])        # sequence of random values
]
```

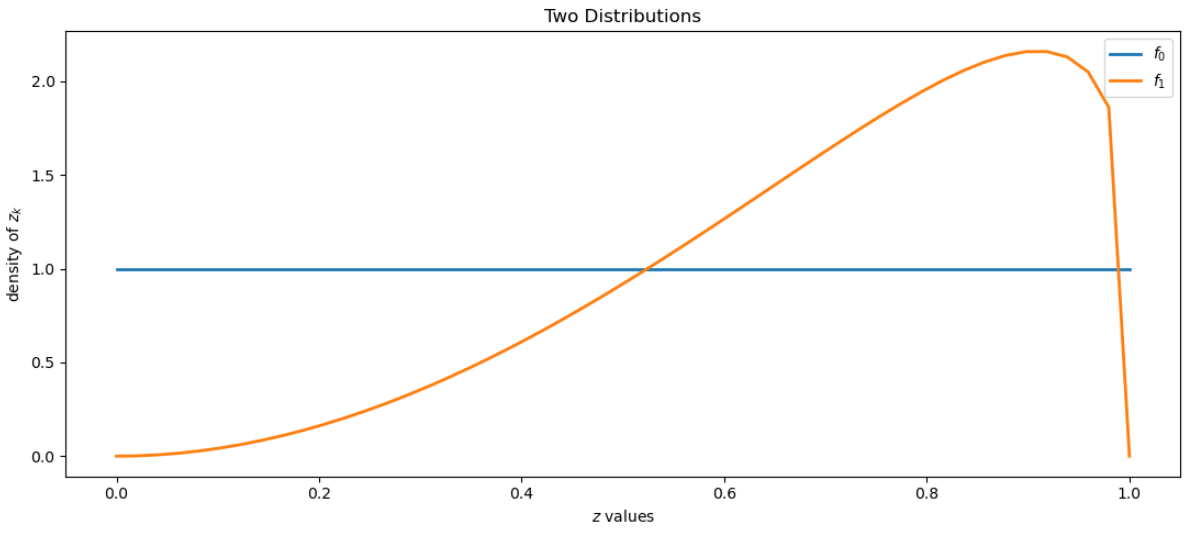
```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=100,
                 L1=100,
                 π_grid_size=200,
                 mc_size=1000):

        self.c, self.π_grid_size = c, π_grid_size
        self.a0, self.b0, self.a1, self.b1 = a0, b0, a1, b1
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):
```

(continues on next page)

(continued from previous page)

```
        return p(x, self.a1, self.b1)

    def κ(self, z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """

        a0, b0, a1, b1 = self.a0, self.b0, self.a1, self.b1

        π_f0, π_f1 = π * p(z, a0, b0), (1 - π) * p(z, a1, b1)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new
```

```
wf = WaldFriedman()

grid = np.linspace(0, 1, 50)

plt.figure()

plt.title("Two Distributions")
plt.plot(grid, wf.f0(grid), lw=2, label="$f_0$")
plt.plot(grid, wf.f1(grid), lw=2, label="$f_1$")

plt.legend()
plt.xlabel("$z$ values")
plt.ylabel("density of $z_k$")

plt.tight_layout()
plt.show()
```



Above, we plot the two possible probability densities f_0 and f_1

17.3 Frequentist Decision Rule

The Navy told the Captain to use a frequentist decision rule.

In particular, it gave him a decision rule that the Navy had designed by using frequentist statistical theory to minimize an expected loss function.

That decision rule is characterized by a sample size t and a cutoff d associated with a likelihood ratio.

Let $L\left(z^t\right) = \prod_{i=0}^{t} \frac{f_0(z_i)}{f_1(z_i)}$ be the likelihood ratio associated with observing the sequence $\{z_i\}_{i=0}^{t}$.

The decision rule associated with a sample size t is:

- decide that f_0 is the distribution if the likelihood ratio is greater than d

To understand how that rule was engineered, let null and alternative hypotheses be

- null: H_0: $f = f_0$,
- alternative H_1: $f = f_1$.

Given sample size t and cutoff d, under the model described above, the mathematical expectation of total loss is

$$\bar{V}_{fre}\left(t, d\right) = ct + \pi^* PFA \times \bar{L}_1 + \left(1 - \pi^*\right)\left(1 - PD\right) \times \bar{L}_0 \tag{17.1}$$

$$\begin{aligned} \text{where} \quad PFA &= \Pr\left\{L\left(z^t\right) < d \mid q = f_0\right\} \\ PD &= \Pr\left\{L\left(z^t\right) < d \mid q = f_1\right\} \end{aligned}$$

Here

- PFA denotes the probability of a **false alarm**, i.e., rejecting H_0 when it is true
- PD denotes the probability of a **detection error**, i.e., not rejecting H_0 when H_1 is true

For a given sample size t, the pairs (PFA, PD) lie on a **receiver operating characteristic curve** and can be uniquely pinned down by choosing d.

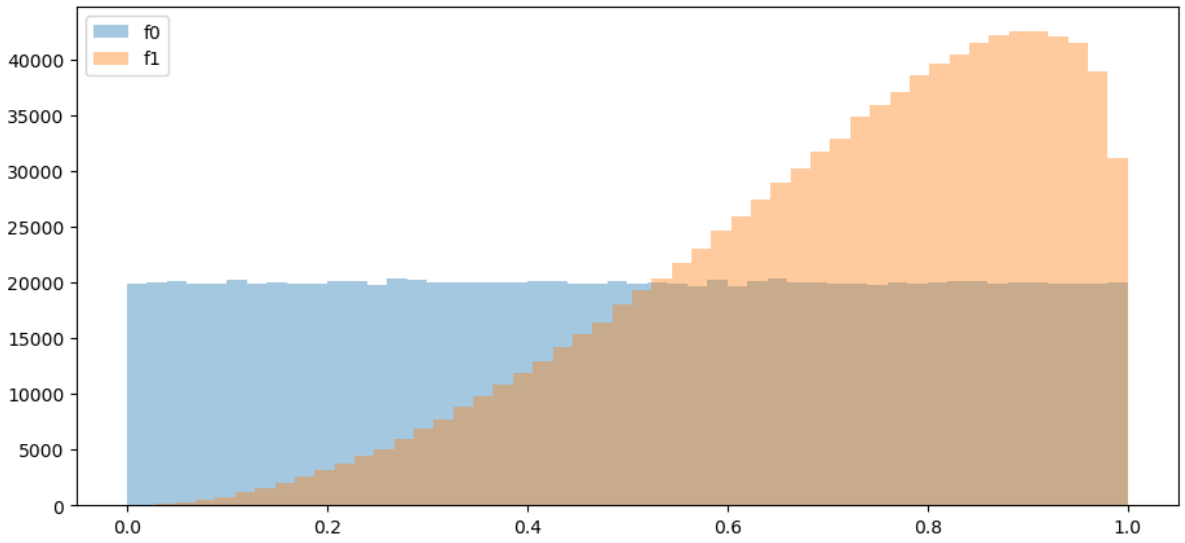
To see some receiver operating characteristic curves, please see this lecture *Likelihood Ratio Processes*.

To solve for $\bar{V}_{fre}\left(t, d\right)$ numerically, we first simulate sequences of z when either f_0 or f_1 generates data.

```
N = 10000
T = 100
```

```
z0_arr = np.random.beta(wf.a0, wf.b0, (N, T))
z1_arr = np.random.beta(wf.a1, wf.b1, (N, T))
```

```
plt.hist(z0_arr.flatten(), bins=50, alpha=0.4, label='f0')
plt.hist(z1_arr.flatten(), bins=50, alpha=0.4, label='f1')
plt.legend()
plt.show()
```



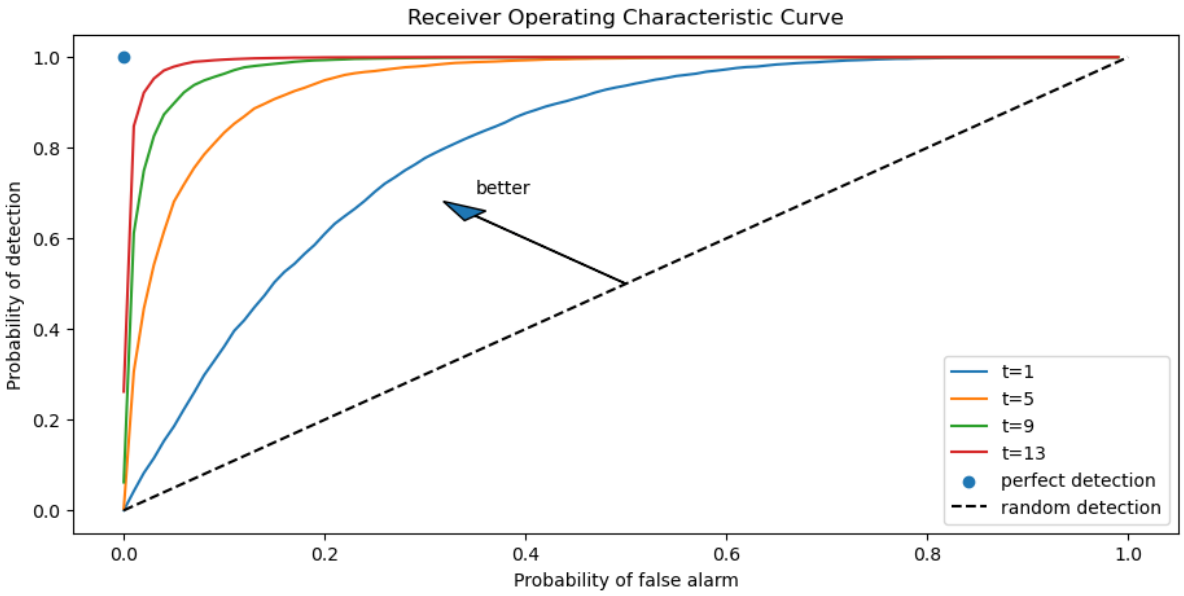
We can compute sequences of likelihood ratios using simulated samples.

```
l = lambda z: wf.f0(z) / wf.f1(z)
```

```
l0_arr = l(z0_arr)
l1_arr = l(z1_arr)

L0_arr = np.cumprod(l0_arr, 1)
L1_arr = np.cumprod(l1_arr, 1)
```

With an empirical distribution of likelihood ratios in hand, we can draw **receiver operating characteristic curves** by enumerating (PFA, PD) pairs given each sample size t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(L0_arr[:, t], PFA)
    PD = [np.sum(L1_arr[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
plt.show()
```



Our frequentist minimizes the expected total loss presented in equation (17.1) by choosing (t, d).

Doing that delivers an expected loss

$$\bar{V}_{fre} = \min_{t,d} \bar{V}_{fre}(t, d).$$

We first consider the case in which $\pi^* = \Pr\{\text{nature selects } f_0\} = 0.5$.

We can solve the minimization problem in two steps.

First, we fix t and find the optimal cutoff d and consequently the minimal $\bar{V}_{fre}(t)$.

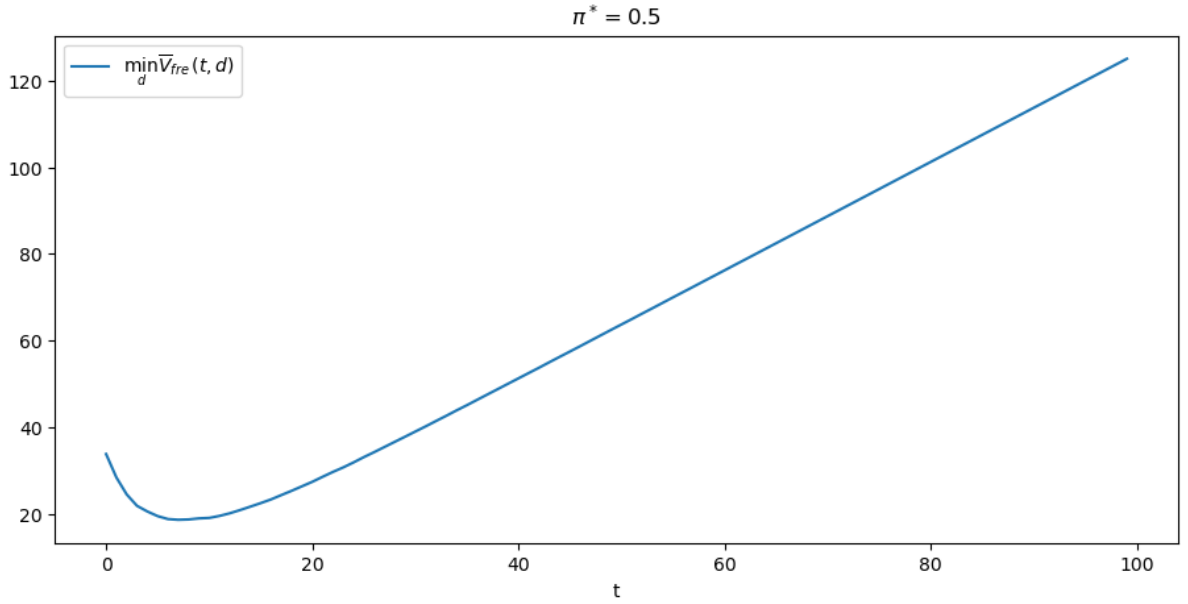
Here is Python code that does that and then plots a useful graph.

```
@njit
def V_fre_d_t(d, t, L0_arr, L1_arr, π_star, wf):

    N = L0_arr.shape[0]

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    V = π_star * PFA *wf. L1 + (1 - π_star) * (1 - PD) * wf.L0

    return V
```

```
def V_fre_t(t, L0_arr, L1_arr, π_star, wf):

    res = minimize(V_fre_d_t, 1, args=(t, L0_arr, L1_arr, π_star, wf), method='Nelder-
↪Mead')
    V = res.fun
    d = res.x

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    return V, PFA, PD
```

```
def compute_V_fre(L0_arr, L1_arr, π_star, wf):

    T = L0_arr.shape[1]

    V_fre_arr = np.empty(T)
    PFA_arr = np.empty(T)
    PD_arr = np.empty(T)

    for t in range(1, T+1):
        V, PFA, PD = V_fre_t(t, L0_arr, L1_arr, π_star, wf)
        V_fre_arr[t-1] = wf.c * t + V
        PFA_arr[t-1] = PFA
        PD_arr[t-1] = PD

    return V_fre_arr, PFA_arr, PD_arr
```

```
π_star = 0.5
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

plt.plot(range(T), V_fre_arr, label='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()
```




```
t_optimal = np.argmin(V_fre_arr) + 1
```

```
msg = f"The above graph indicates that minimizing over t tells the frequentist to␣
↪draw {t_optimal} observations and then decide."
print(msg)
```

```
The above graph indicates that minimizing over t tells the frequentist to draw 8␣
↪observations and then decide.
```

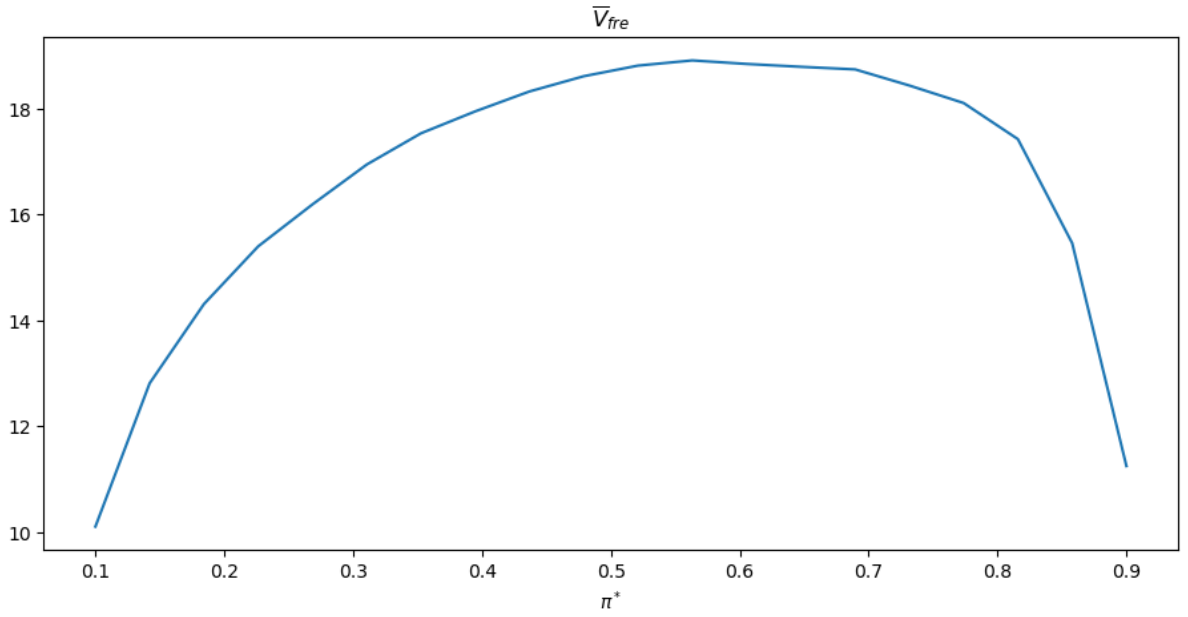
(continues on next page)

(continued from previous page)

Let's now change the value of π^* and watch how the decision rule changes.

```
n_π = 20
π_star_arr = np.linspace(0.1, 0.9, n_π)

V_fre_bar_arr = np.empty(n_π)
t_optimal_arr = np.empty(n_π)
PFA_optimal_arr = np.empty(n_π)
PD_optimal_arr = np.empty(n_π)

for i, π_star in enumerate(π_star_arr):
    V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)
    t_idx = np.argmin(V_fre_arr)

    V_fre_bar_arr[i] = V_fre_arr[t_idx]
    t_optimal_arr[i] = t_idx + 1
    PFA_optimal_arr[i] = PFA_arr[t_idx]
    PD_optimal_arr[i] = PD_arr[t_idx]
```

```
plt.plot(π_star_arr, V_fre_bar_arr)
plt.xlabel('$\pi^*$')
plt.title('$\overline{V}_{fre}$')

plt.show()
```



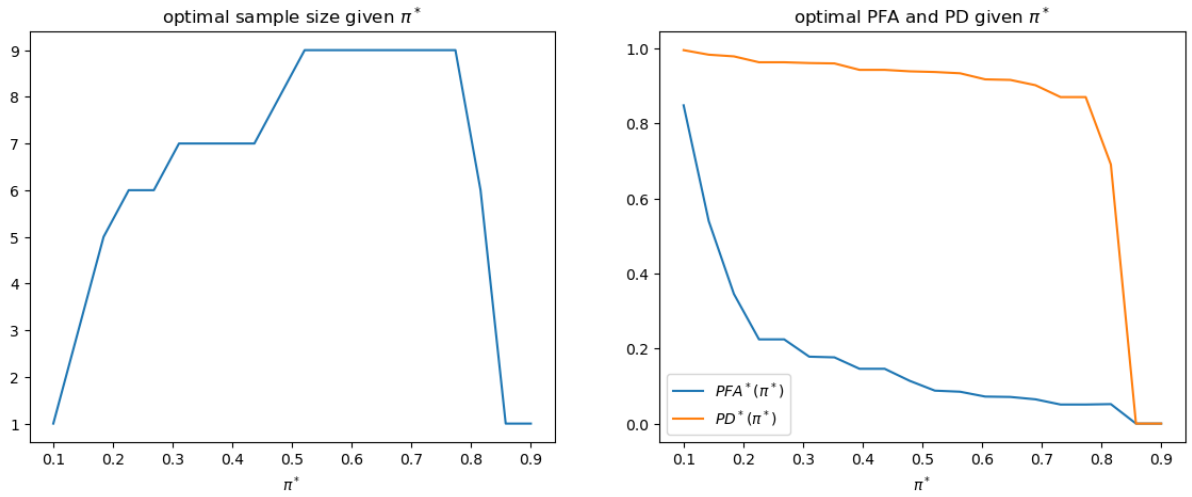
The following shows how optimal sample size t and targeted (PFA, PD) change as π^* varies.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, t_optimal_arr)
axs[0].set_xlabel('$\pi^*$')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_title('optimal sample size given $\pi^*$')

axs[1].plot(π_star_arr, PFA_optimal_arr, label='$PFA^*(\pi^*)$')
axs[1].plot(π_star_arr, PD_optimal_arr, label='$PD^*(\pi^*)$')
axs[1].set_xlabel('$\pi^*$')
axs[1].legend()
axs[1].set_title('optimal PFA and PD given $\pi^*$')

plt.show()
```



17.4 Bayesian Decision Rule

In *A Problem that Stumped Milton Friedman*, we learned how Abraham Wald confirmed the Navy Captain's hunch that there is a better decision rule.

We presented a Bayesian procedure that instructed the Captain to makes decisions by comparing his current Bayesian posterior probability π with two cutoff probabilities called α and β.

To proceed, we borrow some Python code from the quantecon lecture *A Problem that Stumped Milton Friedman* that computes α and β.

```
@njit(parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: interp(π_grid, h, p)

    for i in prange(len(π_grid)):
        π = π_grid[i]
```

(continues on next page)

(continued from previous page)

```
        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

```
@njit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

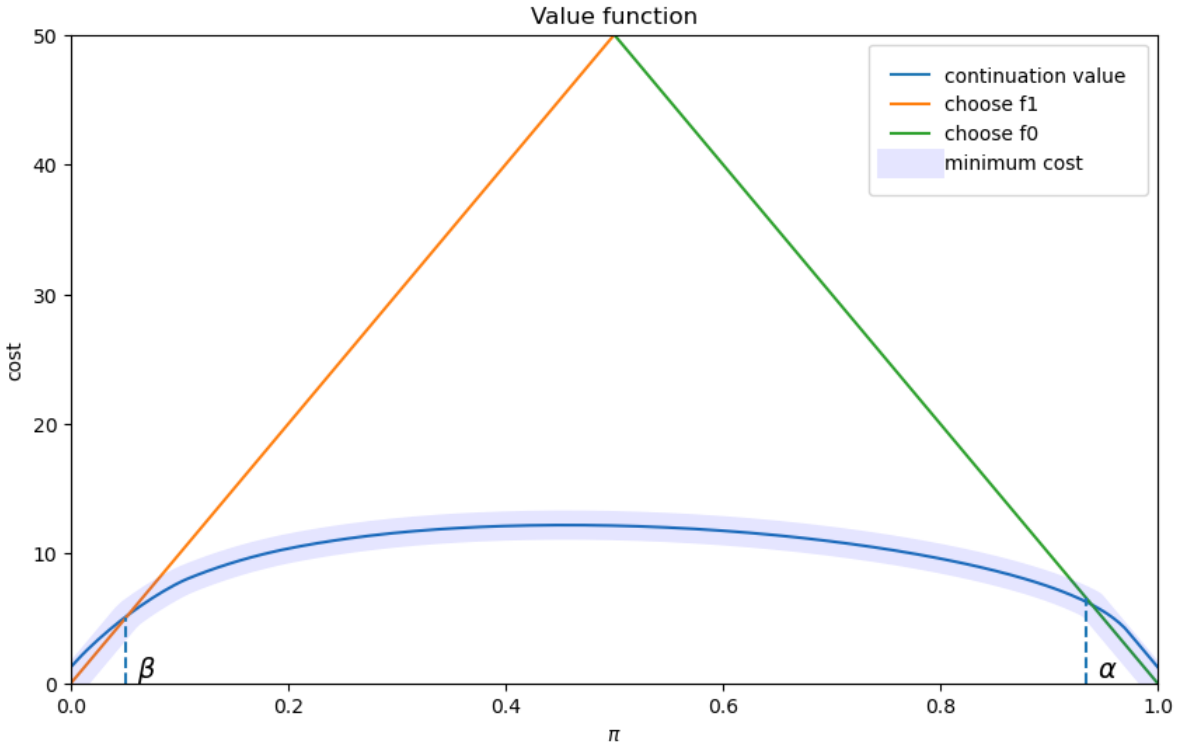
```
h_star = solve_model(wf)
```

```
@njit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1
```

(continues on next page)

(continued from previous page)

```
    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    β = π_grid[np.searchsorted(
                                payoff_f1 - np.minimum(h, payoff_f0),
                                1e-10)
               - 1]
    α = π_grid[np.searchsorted(
                                np.minimum(h, payoff_f1) - payoff_f0,
                                1e-10)
               - 1]

    return (β, α)

β, α = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```



The above figure portrays the value function plotted against the decision maker's Bayesian posterior.

It also shows the probabilities α and β.

The Bayesian decision rule is:

- accept H_0 if $\pi \geq \alpha$
- accept H_1 if $\pi \leq \beta$
- delay deciding and draw another z if $\beta \leq \pi \leq \alpha$

We can calculate two "objective" loss functions under this situation conditioning on knowing for sure that nature has selected f_0, in the first case, or f_1, in the second case.

1. under f_0,

$$V^0(\pi) = \begin{cases} 0 & \text{if } \alpha \leq \pi, \\ c + EV^0(\pi') & \text{if } \beta \leq \pi < \alpha, \\ \bar{L}_1 & \text{if } \pi < \beta. \end{cases}$$

2. under f_1

$$V^1(\pi) = \begin{cases} \bar{L}_0 & \text{if } \alpha \leq \pi, \\ c + EV^1(\pi') & \text{if } \beta \leq \pi < \alpha, \\ 0 & \text{if } \pi < \beta. \end{cases}$$

where $\pi' = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$.

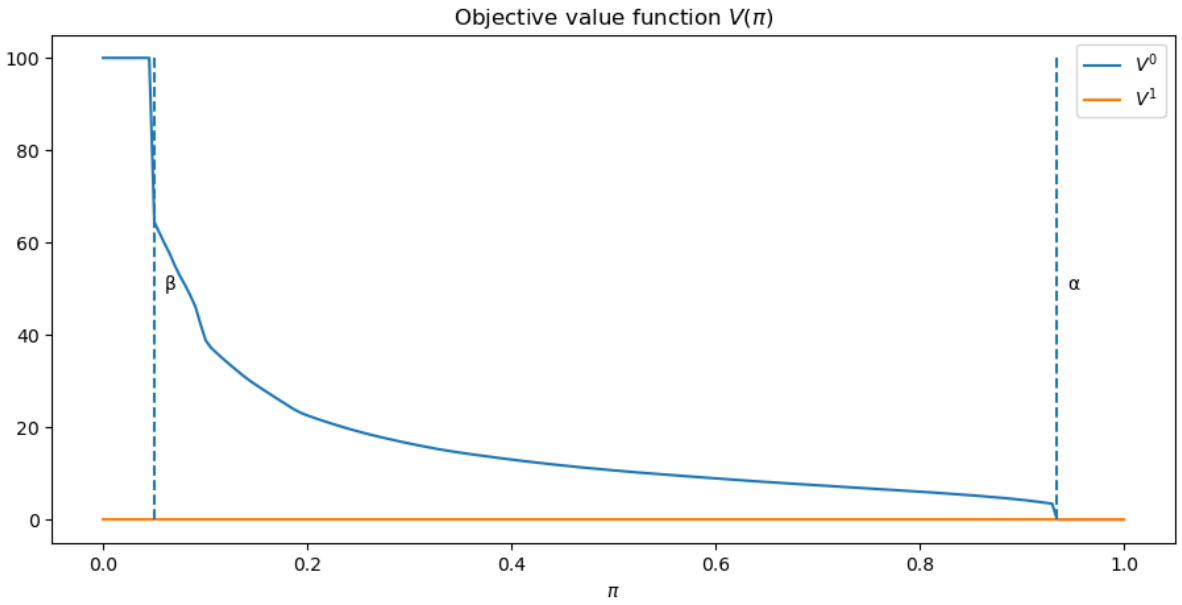
Given a prior probability π_0, the expected loss for the Bayesian is

$$\bar{V}_{Bayes}(\pi_0) = \pi^* V^0(\pi_0) + (1 - \pi^*) V^1(\pi_0).$$

Below we write some Python code that computes $V^0(\pi)$ and $V^1(\pi)$ numerically.

```
@njit(parallel=True)
def V_q(wf, flag):
    V = np.zeros(wf.π_grid_size)
    if flag == 0:
        z_arr = wf.z0
        V[wf.π_grid < β] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.π_grid >= α] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(β <= wf.π_grid) & (wf.π_grid < α)] = 0

        for i in prange(len(wf.π_grid)):
            π = wf.π_grid[i]

            if π >= α or π < β:
                continue

            for j in prange(len(z_arr)):
                π_next = wf.κ(z_arr[j], π)
                V[i] += wf.c + interp(wf.π_grid, V_old, π_next)

            V[i] /= wf.mc_size

        if np.abs(V - V_old).max() < 1e-5:
            break

    return V
```

```
V0 = V_q(wf, 0)
V1 = V_q(wf, 1)

plt.plot(wf.π_grid, V0, label='$V^0$')
plt.plot(wf.π_grid, V1, label='$V^1$')
plt.vlines(β, 0, wf.L0, linestyle='--')
plt.text(β+0.01, wf.L0/2, 'β')
plt.vlines(α, 0, wf.L0, linestyle='--')
plt.text(α+0.01, wf.L0/2, 'α')
plt.xlabel('$\pi$')
plt.title('Objective value function $V(\pi)$')
plt.legend()
plt.show()
```



Given an assumed value for $\pi^* = \Pr\{\text{nature selects } f_0\}$, we can then compute $\bar{V}_{Bayes}(\pi_0)$.

We can then determine an initial Bayesian prior π_0^* that minimizes this objective concept of expected loss.

The figure 9 below plots four cases corresponding to $\pi^* = 0.25, 0.3, 0.5, 0.7$.

We observe that in each case π_0^* equals π^*.

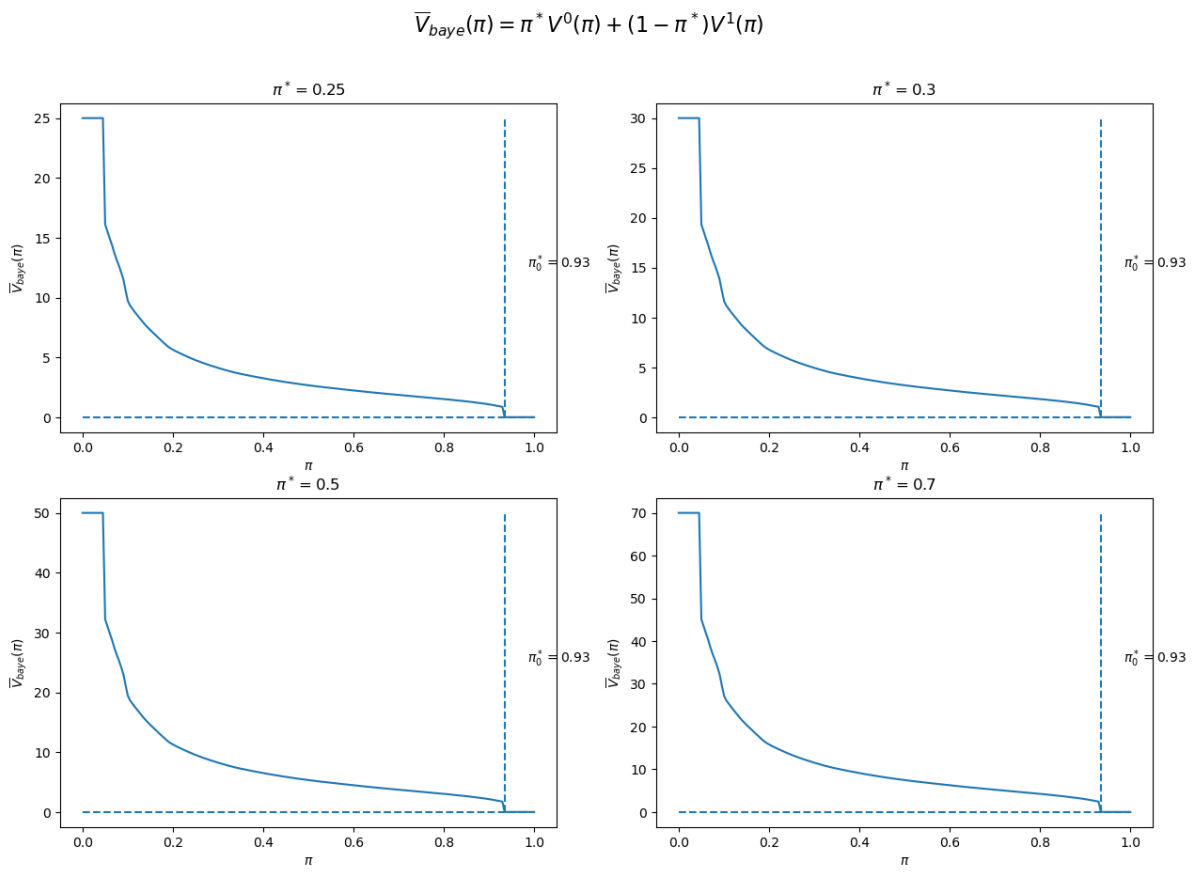
```
def compute_V_baye_bar(π_star, V0, V1, wf):

    V_baye = π_star * V0 + (1 - π_star) * V1
    π_idx = np.argmin(V_baye)
    π_optimal = wf.π_grid[π_idx]
    V_baye_bar = V_baye[π_idx]

    return V_baye, π_optimal, V_baye_bar
```

```
π_star_arr = [0.25, 0.3, 0.5, 0.7]

fig, axs = plt.subplots(2, 2, figsize=(15, 10))

for i, π_star in enumerate(π_star_arr):
    row_i = i // 2
    col_i = i % 2

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    axs[row_i, col_i].plot(wf.π_grid, V_baye)
    axs[row_i, col_i].hlines(V_baye_bar, 0, 1, linestyle='--')
    axs[row_i, col_i].vlines(π_optimal, V_baye_bar, V_baye.max(), linestyle='--')
    axs[row_i, col_i].text(π_optimal+0.05, (V_baye_bar + V_baye.max()) / 2,
                           '${\pi_0^*}=$'+f'{π_optimal:0.2f}')
    axs[row_i, col_i].set_xlabel('$\pi$')
    axs[row_i, col_i].set_ylabel('$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title('$\pi^*=$' + f'{π_star}')
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('$\overline{V}_{baye}(\pi)=\pi^*V^0(\pi) + (1-\pi^*)V^1(\pi)$',␣
 ↪fontsize=16)
plt.show()
```



This pattern of outcomes holds more generally.

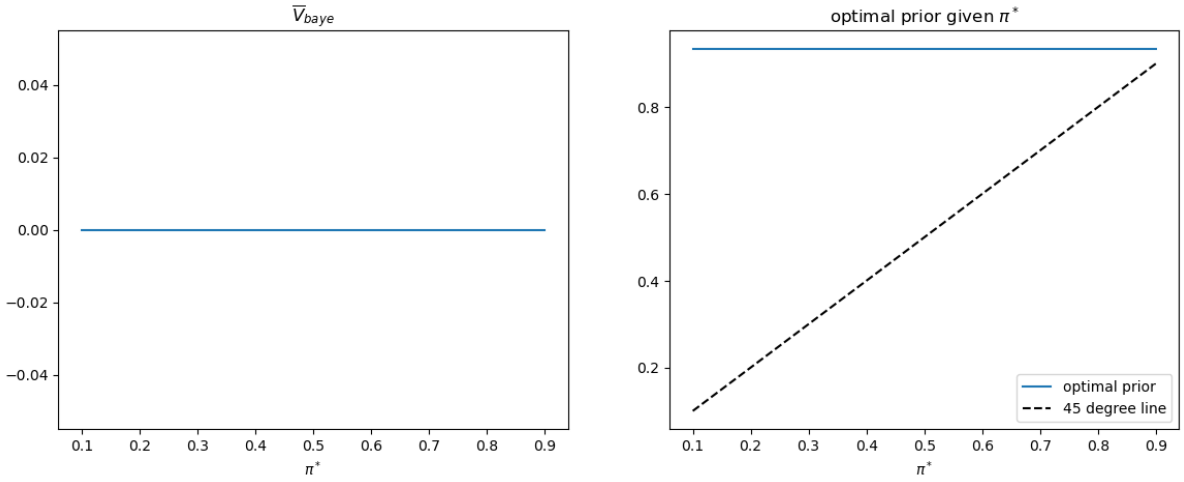
Thus, the following Python code generates the associated graph that verifies the equality of π_0^* to π^* holds for all π^*.

```
π_star_arr = np.linspace(0.1, 0.9, n_π)
V_baye_bar_arr = np.empty_like(π_star_arr)
π_optimal_arr = np.empty_like(π_star_arr)

for i, π_star in enumerate(π_star_arr):

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    V_baye_bar_arr[i] = V_baye_bar
    π_optimal_arr[i] = π_optimal

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_baye_bar_arr)
axs[0].set_xlabel('$\pi^*$')
axs[0].set_title('$\overline{V}_{baye}$')

axs[1].plot(π_star_arr, π_optimal_arr, label='optimal prior')
```

(continues on next page)

(continued from previous page)

```
axs[1].plot([π_star_arr.min(), π_star_arr.max()],
            [π_star_arr.min(), π_star_arr.max()],
            c='k', linestyle='--', label='45 degree line')
axs[1].set_xlabel('$\pi^*$')
axs[1].set_title('optimal prior given $\pi^*$')
axs[1].legend()

plt.show()
```



17.5 Was the Navy Captain's Hunch Correct?

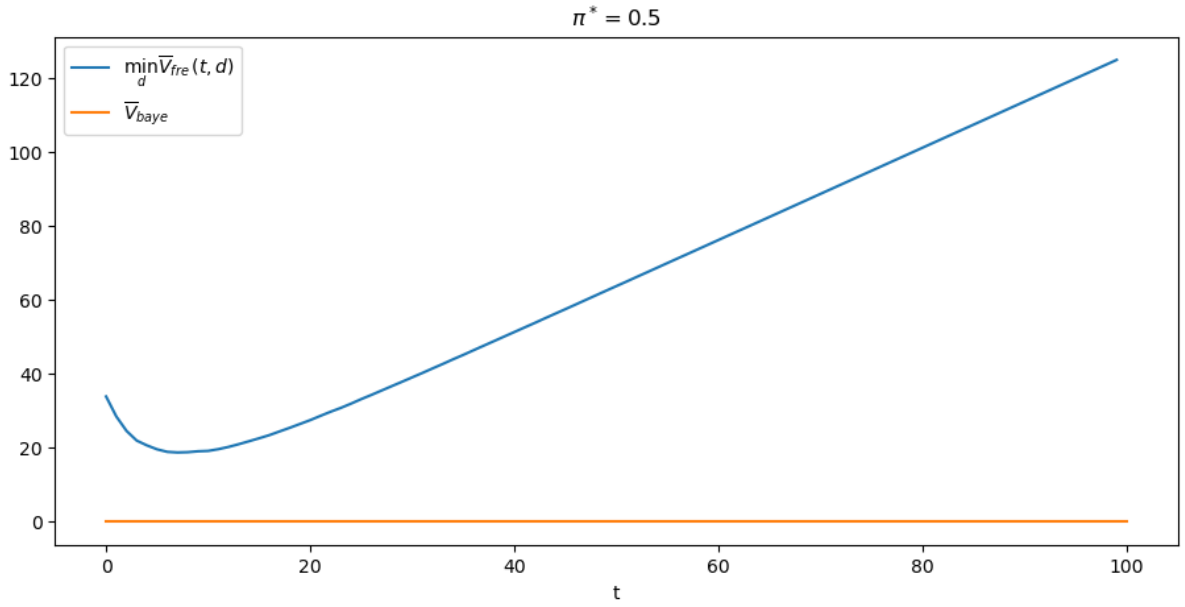
We now compare average (i.e., frequentist) losses obtained by the frequentist and Bayesian decision rules.

As a starting point, let's compare average loss functions when $\pi^* = 0.5$.

```
π_star = 0.5
```

```
# frequentist
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

# bayesian
V_baye = π_star * V0 + π_star * V1
V_baye_bar = V_baye.min()
```

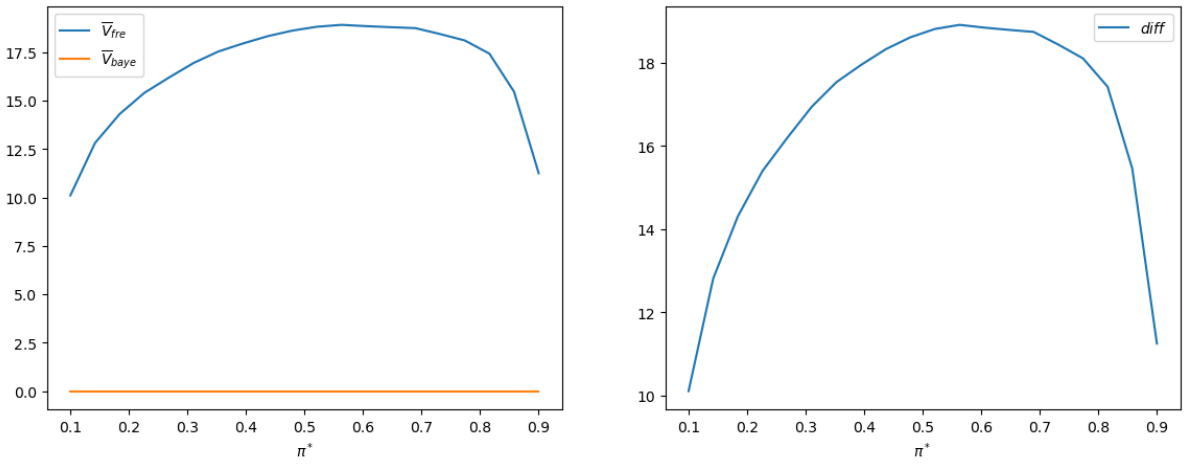
```
plt.plot(range(T), V_fre_arr, label='$\min_{d} \overline{V}_{fre}(t,d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label='$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()
```



Evidently, there is no sample size t at which the frequentist decision rule attains a lower loss function than does the Bayesian rule.

Furthermore, the following graph indicates that the Bayesian decision rule does better on average for all values of π^*.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_fre_bar_arr, label='$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label='$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel('$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='$diff$')
axs[1].legend()
axs[1].set_xlabel('$\pi^*$')

plt.show()
```



The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

17.6 More Details

We can provide more insights by focusing on the case in which $\pi^* = 0.5 = \pi_0$.

```
π_star = 0.5
```

Recall that when $\pi^* = 0.5$, the frequentist decision rule sets a sample size `t_optimal` **ex ante**.

For our parameter settings, we can compute its value:

```
t_optimal
```

```
8
```

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

```
t_idx = t_optimal - 1
```

17.7 Distribution of Bayesian Decision Rule's Time to Decide

By using simulations, we compute the frequency distribution of time to deciding for the Bayesian decision rule and compare that time to the frequentist rule's fixed t.

The following Python code creates a graph that shows the frequency distribution of Bayesian times to decide of Bayesian decision maker, conditional on distribution $q = f_0$ or $q = f_1$ generating the data.

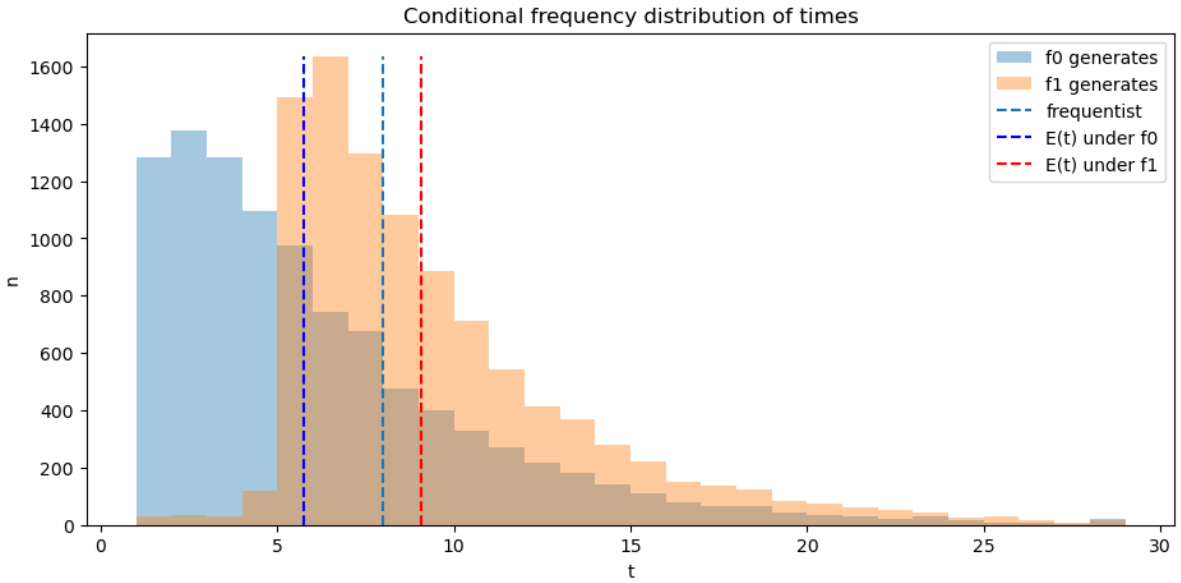
The blue and red dotted lines show averages for the Bayesian decision rule, while the black dotted line shows the frequentist optimal sample size t.

On average the Bayesian rule decides **earlier** than the frequentist rule when $q = f_0$ and **later** when $q = f_1$.

```
@njit(parallel=True)
def check_results(L_arr, α, β, flag, π0):

    N, T = L_arr.shape

    time_arr = np.empty(N)
    correctness = np.empty(N)

    π_arr = π0 * L_arr / (π0 * L_arr + 1 - π0)

    for i in prange(N):
        for t in range(T):
            if (π_arr[i, t] < β) or (π_arr[i, t] > α):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > α) or (flag == 1 and π_
↪arr[i, t] < β)
                break

    return time_arr, correctness
```

```
time_arr0, correctness0 = check_results(L0_arr, α, β, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, α, β, 1, π_star)

# unconditional distribution
time_arr_u = np.concatenate((time_arr0, time_arr1))
correctness_u = np.concatenate((correctness0, correctness1))
```

```
n1 = plt.hist(time_arr0, bins=range(1, 30), alpha=0.4, label='f0 generates')[0]
n2 = plt.hist(time_arr1, bins=range(1, 30), alpha=0.4, label='f1 generates')[0]
plt.vlines(t_optimal, 0, max(n1.max(), n2.max()), linestyle='--', label='frequentist')
plt.vlines(np.mean(time_arr0), 0, max(n1.max(), n2.max()),
           linestyle='--', color='b', label='E(t) under f0')
plt.vlines(np.mean(time_arr1), 0, max(n1.max(), n2.max()),
           linestyle='--', color='r', label='E(t) under f1')
plt.legend();

plt.xlabel('t')
plt.ylabel('n')
plt.title('Conditional frequency distribution of times')

plt.show()
```



Later we'll figure out how these distributions ultimately affect objective expected values under the two decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can easily compute the updated beliefs at any time t using the one-to-one mapping from L_t to π_t given π_0 described in this lecture *Likelihood Ratio Processes*.

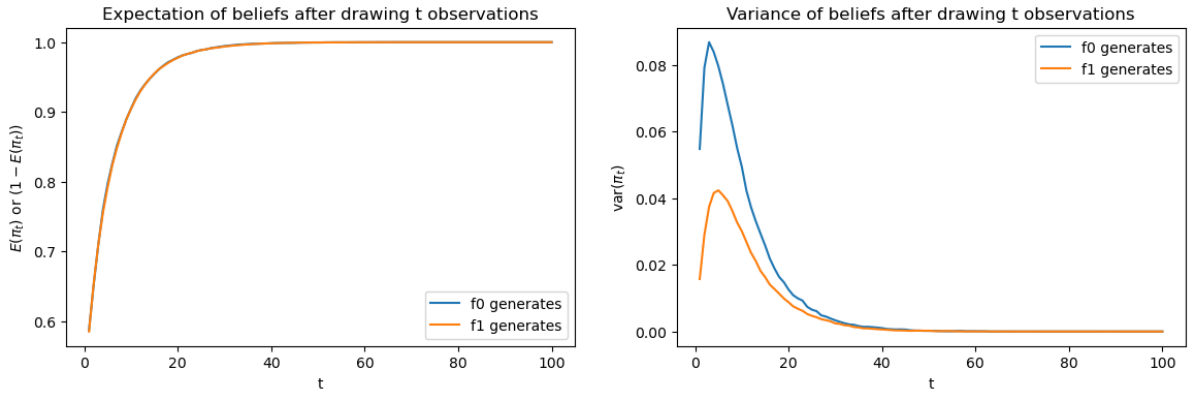
```
π0_arr = π_star * L0_arr / (π_star * L0_arr + 1 - π_star)
π1_arr = π_star * L1_arr / (π_star * L1_arr + 1 - π_star)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 4))

axs[0].plot(np.arange(1, π0_arr.shape[1]+1), np.mean(π0_arr, 0), label='f0 generates')
```

(continues on next page)

(continued from previous page)

```
axs[0].plot(np.arange(1, π1_arr.shape[1]+1), 1 - np.mean(π1_arr, 0), label='f1␣
↪generates')
axs[0].set_xlabel('t')
axs[0].set_ylabel('$E(\pi_t)$ or ($1 - E(\pi_t)$)')
axs[0].set_title('Expectation of beliefs after drawing t observations')
axs[0].legend()

axs[1].plot(np.arange(1, π0_arr.shape[1]+1), np.var(π0_arr, 0), label='f0 generates')
axs[1].plot(np.arange(1, π1_arr.shape[1]+1), np.var(π1_arr, 0), label='f1 generates')
axs[1].set_xlabel('t')
axs[1].set_ylabel('var($\pi_t$)')
axs[1].set_title('Variance of beliefs after drawing t observations')
axs[1].legend()

plt.show()
```



The above figures compare averages and variances of updated Bayesian posteriors after t draws.

The left graph compares $E(\pi_t)$ under f_0 to $1 - E(\pi_t)$ under f_1: they lie on top of each other.

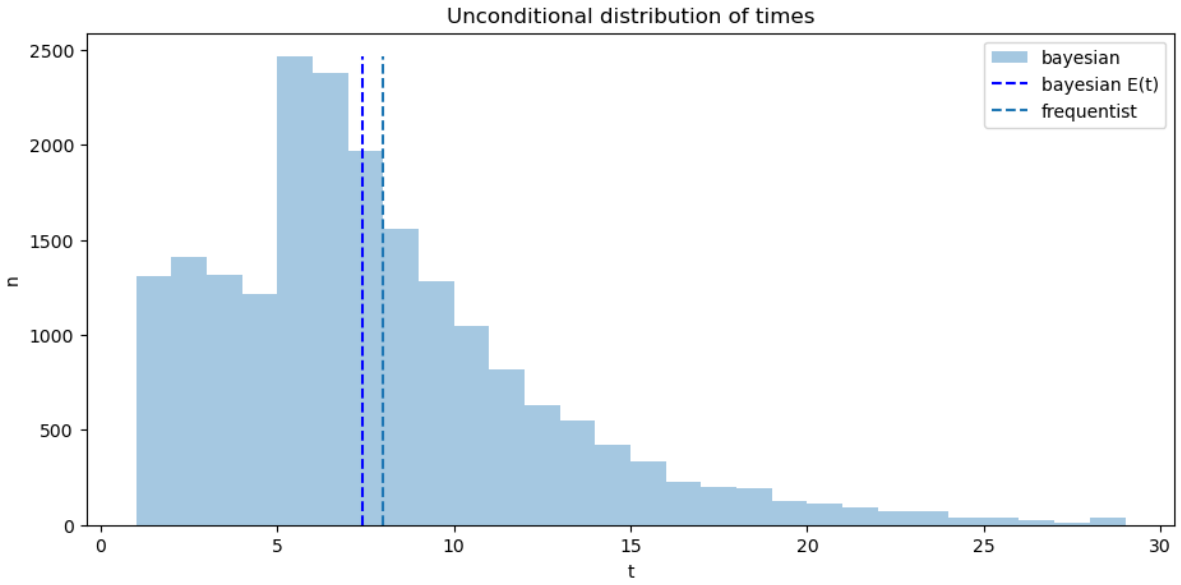
However, as the right hand size graph shows, there is significant difference in variances when t is small: the variance is lower under f_1.

The difference in variances is the reason that the Bayesian decision maker waits longer to decide when f_1 generates the data.

The code below plots outcomes of constructing an unconditional distribution by simply pooling the simulated data across the two possible distributions f_0 and f_1.

The pooled distribution describes a sense in which on average the Bayesian decides earlier, an outcome that seems at least partly to confirm the Navy Captain's hunch.

```
n = plt.hist(time_arr_u, bins=range(1, 30), alpha=0.4, label='bayesian')[0]
plt.vlines(np.mean(time_arr_u), 0, n.max(), linestyle='--',
           color='b', label='bayesian E(t)')
plt.vlines(t_optimal, 0, n.max(), linestyle='--', label='frequentist')
plt.legend()

plt.xlabel('t')
plt.ylabel('n')
plt.title('Unconditional distribution of times')

plt.show()
```



17.8 Probability of Making Correct Decision

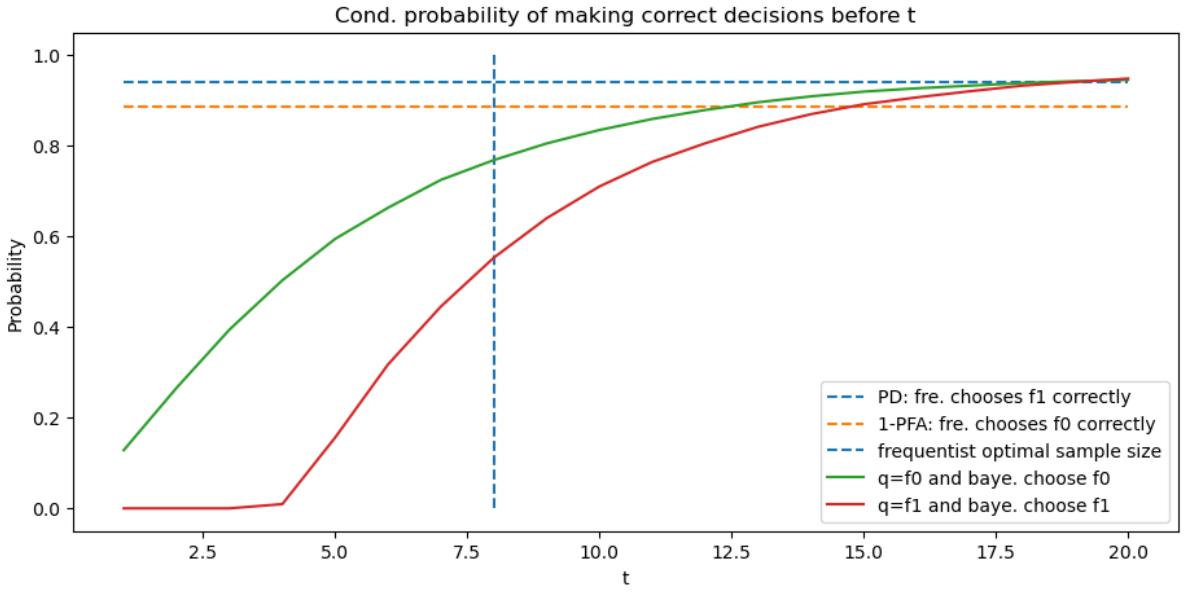
Now we use simulations to compute the fraction of samples in which the Bayesian and the frequentist decision rules decide correctly.

For the frequentist rule, the probability of making the correct decision under f_1 is the optimal probability of detection given t that we defined earlier, and similarly it equals 1 minus the optimal probability of a false alarm under f_0.

Below we plot these two probabilities for the frequentist rule, along with the conditional probabilities that the Bayesian rule decides before t *and* that the decision is correct.

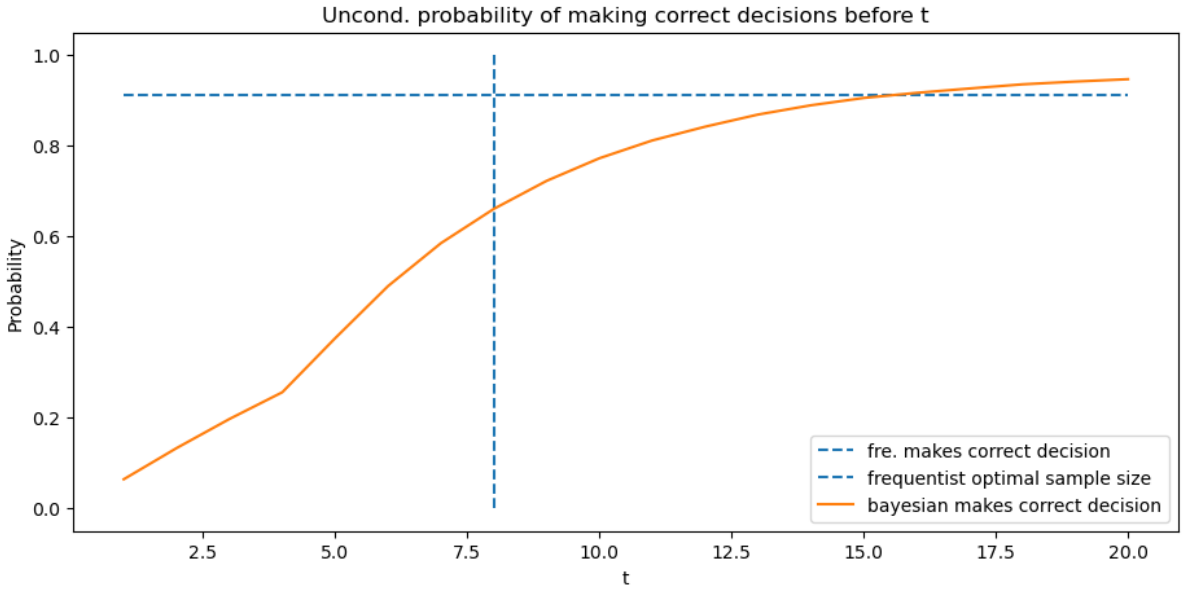
```
# optimal PFA and PD of frequentist with optimal sample size
V, PFA, PD = V_fre_t(t_optimal, L0_arr, L1_arr, π_star, wf)
```

```
plt.plot([1, 20], [PD, PD], linestyle='--', label='PD: fre. chooses f1 correctly')
plt.plot([1, 20], [1-PFA, 1-PFA], linestyle='--', label='1-PFA: fre. chooses f0␣
↪correctly')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr0.size
T_arr = np.arange(1, 21)
plt.plot(T_arr, [np.sum(correctness0[time_arr0 <= t] == 1) / N for t in T_arr],
         label='q=f0 and baye. choose f0')
plt.plot(T_arr, [np.sum(correctness1[time_arr1 <= t] == 1) / N for t in T_arr],
         label='q=f1 and baye. choose f1')
plt.legend(loc=4)

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Cond. probability of making correct decisions before t')

plt.show()
```



By averaging using π^*, we also plot the unconditional distribution.

```
plt.plot([1, 20], [(PD + 1 - PFA) / 2, (PD + 1 - PFA) / 2],
         linestyle='--', label='fre. makes correct decision')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr_u.size
plt.plot(T_arr, [np.sum(correctness_u[time_arr_u <= t] == 1) / N for t in T_arr],
         label="bayesian makes correct decision")
plt.legend()

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Uncond. probability of making correct decisions before t')

plt.show()
```



17.9 Distribution of Likelihood Ratios at Frequentist's t

Next we use simulations to construct distributions of likelihood ratios after t draws.

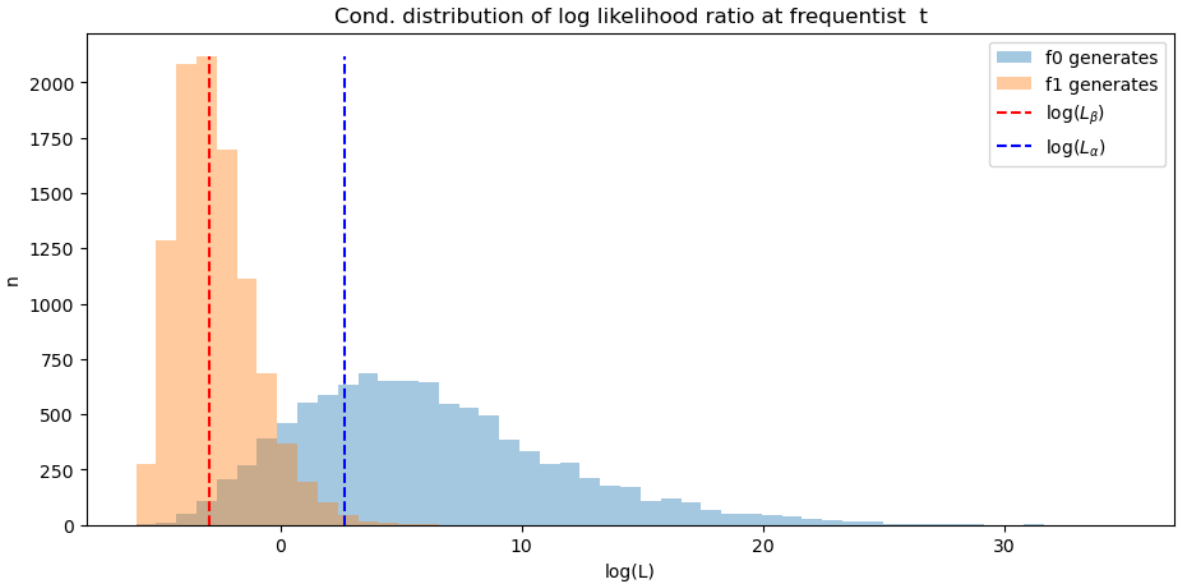
To serve as useful reference points, we also show likelihood ratios that correspond to the Bayesian cutoffs α and β.

In order to exhibit the distribution more clearly, we report logarithms of likelihood ratios.

The graphs below reports two distributions, one conditional on f_0 generating the data, the other conditional on f_1 generating the data.

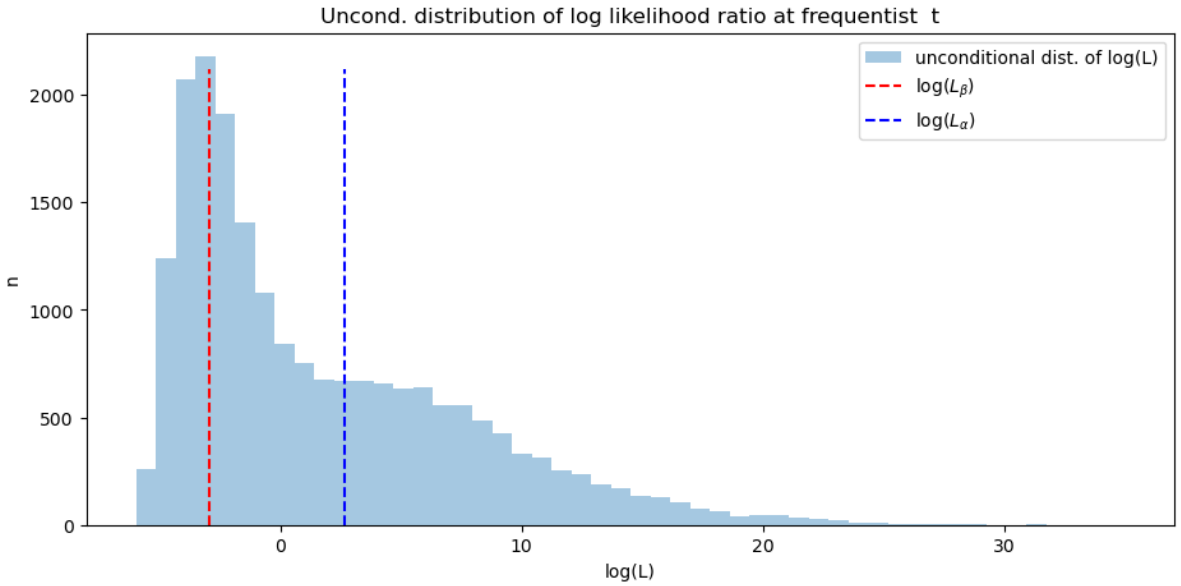
```
Lα = (1 - π_star) *  α / (π_star - π_star * α)
Lβ = (1 - π_star) *  β / (π_star - π_star * β)
```

```
L_min = min(L0_arr[:, t_idx].min(), L1_arr[:, t_idx].min())
L_max = max(L0_arr[:, t_idx].max(), L1_arr[:, t_idx].max())
bin_range = np.linspace(np.log(L_min), np.log(L_max), 50)
n0 = plt.hist(np.log(L0_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f0 generates
 ↪')[0]
n1 = plt.hist(np.log(L1_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f1 generates
 ↪')[0]

plt.vlines(np.log(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_α$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Cond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



The next graph plots the unconditional distribution of Bayesian times to decide, constructed as earlier by pooling the two conditional distributions.

```
plt.hist(np.log(np.concatenate([L0_arr[:, t_idx], L1_arr[:, t_idx]])),
         bins=50, alpha=0.4, label='unconditional dist. of log(L)')
plt.vlines(np.log(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
 ↪'log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
 ↪'log($L_α$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



Part III

Applications of Statistics

CHAPTER

EIGHTEEN

MULTIVARIATE HYPERGEOMETRIC DISTRIBUTION

Contents

18.1 Overview

This lecture describes how an administrator deployed a **multivariate hypergeometric distribution** in order to access the fairness of a procedure for awarding research grants.

In the lecture we'll learn about

- properties of the multivariate hypergeometric distribution
- first and second moments of a multivariate hypergeometric distribution
- using a Monte Carlo simulation of a multivariate normal distribution to evaluate the quality of a normal approximation
- the administrator's problem and why the multivariate hypergeometric distribution is the right tool

18.2 The Administrator's Problem

An administrator in charge of allocating research grants is in the following situation.

To help us forget details that are none of our business here and to protect the anonymity of the administrator and the subjects, we call research proposals **balls** and continents of residence of authors of a proposal a **color**.

There are K_i balls (proposals) of color i.

There are c distinct colors (continents of residence).

Thus, $i = 1, 2, \ldots, c$

So there is a total of $N = \sum_{i=1}^{c} K_i$ balls.

All N of these balls are placed in an urn.

Then n balls are drawn randomly.

The selection procedure is supposed to be **color blind** meaning that **ball quality**, a random variable that is supposed to be independent of **ball color**, governs whether a ball is drawn.

Thus, the selection procedure is supposed randomly to draw n balls from the urn.

The n balls drawn represent successful proposals and are awarded research funds.

The remaining $N - n$ balls receive no research funds.

18.2.1 Details of the Awards Procedure Under Study

Let k_i be the number of balls of color i that are drawn.

Things have to add up so $\sum_{i=1}^{c} k_i = n$.

Under the hypothesis that the selection process judges proposals on their quality and that quality is independent of continent of the author's continent of residence, the administrator views the outcome of the selection procedure as a random vector

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_c \end{pmatrix}.$$

To evaluate whether the selection procedure is **color blind** the administrator wants to study whether the particular realization of X drawn can plausibly be said to be a random draw from the probability distribution that is implied by the **color blind** hypothesis.

The appropriate probability distribution is the one described here.

Let's now instantiate the administrator's problem, while continuing to use the colored balls metaphor.

The administrator has an urn with $N = 238$ balls.

157 balls are blue, 11 balls are green, 46 balls are yellow, and 24 balls are black.

So $(K_1, K_2, K_3, K_4) = (157, 11, 46, 24)$ and $c = 4$.

15 balls are drawn without replacement.

So $n = 15$.

The administrator wants to know the probability distribution of outcomes

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_4 \end{pmatrix}.$$

In particular, he wants to know whether a particular outcome - in the form of a 4×1 vector of integers recording the numbers of blue, green, yellow, and black balls, respectively, - contains evidence against the hypothesis that the selection process is *fair*, which here means *color blind* and truly are random draws without replacement from the population of N balls.

The right tool for the administrator's job is the **multivariate hypergeometric distribution**.

18.2.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import njit, prange
```

To recapitulate, we assume there are in total c types of objects in an urn.

If there are K_i type i object in the urn and we take n draws at random without replacement, then the numbers of type i objects in the sample $(k_1, k_2, \ldots, k_c)$ has the multivariate hypergeometric distribution.

Note again that $N = \sum_{i=1}^{c} K_i$ is the total number of objects in the urn and $n = \sum_{i=1}^{c} k_i$.

Notation

We use the following notation for **binomial coefficients**: $\binom{m}{q} = \frac{m!}{(m-q)!}$.

The multivariate hypergeometric distribution has the following properties:

Probability mass function:

$$\Pr\{X_i = k_i \ \forall i\} = \frac{\prod_{i=1}^{c} \binom{K_i}{k_i}}{\binom{N}{n}}$$

Mean:

$$\mathrm{E}(X_i) = n\frac{K_i}{N}$$

Variances and covariances:

$$\mathrm{Var}(X_i) = n\frac{N-n}{N-1}\ \frac{K_i}{N}\left(1 - \frac{K_i}{N}\right)$$

$$\mathrm{Cov}(X_i, X_j) = -n\frac{N-n}{N-1}\ \frac{K_i}{N}\frac{K_j}{N}$$

To do our work for us, we'll write an `Urn` class.

```
class Urn:

    def __init__(self, K_arr):
        """
        Initialization given the number of each type i object in the urn.

        Parameters
        ----------
        K_arr: ndarray(int)
            number of each type i object.
        """

        self.K_arr = np.array(K_arr)
        self.N = np.sum(K_arr)
        self.c = len(K_arr)
```

(continues on next page)

(continued from previous page)

```
    def pmf(self, k_arr):
        """
        Probability mass function.

        Parameters
        ----------
        k_arr: ndarray(int)
            number of observed successes of each object.
        """

        K_arr, N = self.K_arr, self.N

        k_arr = np.atleast_2d(k_arr)
        n = np.sum(k_arr, 1)

        num = np.prod(comb(K_arr, k_arr), 1)
        denom = comb(N, n)

        pr = num / denom

        return pr

    def moments(self, n):
        """
        Compute the mean and variance-covariance matrix for
        multivariate hypergeometric distribution.

        Parameters
        ----------
        n: int
            number of draws.
        """

        K_arr, N, c = self.K_arr, self.N, self.c

        # mean
        μ = n * K_arr / N

        # variance-covariance matrix
        Σ = np.full((c, c), n * (N - n) / (N - 1) / N ** 2)
        for i in range(c-1):
            Σ[i, i] *= K_arr[i] * (N - K_arr[i])
            for j in range(i+1, c):
                Σ[i, j] *= - K_arr[i] * K_arr[j]
                Σ[j, i] = Σ[i, j]

        Σ[-1, -1] *= K_arr[-1] * (N - K_arr[-1])

        return μ, Σ

    def simulate(self, n, size=1, seed=None):
        """
        Simulate a sample from multivariate hypergeometric
        distribution where at each draw we take n objects
        from the urn without replacement.
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ----------
        n: int
            number of objects for each draw.
        size: int(optional)
            sample size.
        seed: int(optional)
            random seed.
        """

        K_arr = self.K_arr

        gen = np.random.Generator(np.random.PCG64(seed))
        sample = gen.multivariate_hypergeometric(K_arr, n, size=size)

        return sample
```

18.3 Usage

18.3.1 First example

Apply this to an example from wiki:

Suppose there are 5 black, 10 white, and 15 red marbles in an urn. If six marbles are chosen without replacement, the probability that exactly two of each color are chosen is

$$P(2 \text{ black}, 2 \text{ white}, 2 \text{ red}) = \frac{\binom{5}{2}\binom{10}{2}\binom{15}{2}}{\binom{30}{6}} = 0.079575596816976$$

```
# construct the urn
K_arr = [5, 10, 15]
urn = Urn(K_arr)
```

Now use the Urn Class method `pmf` to compute the probability of the outcome $X = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$

```
k_arr = [2, 2, 2] # array of number of observed successes
urn.pmf(k_arr)
```

```
array([0.0795756])
```

We can use the code to compute probabilities of a list of possible outcomes by constructing a 2-dimensional array `k_arr` and `pmf` will return an array of probabilities for observing each case.

```
k_arr = [[2, 2, 2], [1, 3, 2]]
urn.pmf(k_arr)
```

```
array([0.0795756, 0.1061008])
```

Now let's compute the mean vector and variance-covariance matrix.

```
n = 6
μ, Σ = urn.moments(n)
```

```
μ
```

```
array([1., 2., 3.])
```

```
Σ
```

```
array([[ 0.68965517, -0.27586207, -0.4137931 ],
       [-0.27586207,  1.10344828, -0.82758621],
       [-0.4137931 , -0.82758621,  1.24137931]])
```

18.3.2 Back to The Administrator's Problem

Now let's turn to the grant administrator's problem.

Here the array of numbers of i objects in the urn is $(157, 11, 46, 24)$.

```
K_arr = [157, 11, 46, 24]
urn = Urn(K_arr)
```

Let's compute the probability of the outcome $(10, 1, 4, 0)$.

```
k_arr = [10, 1, 4, 0]
urn.pmf(k_arr)
```

```
array([0.01547738])
```

We can compute probabilities of three possible outcomes by constructing a 3-dimensional arrays `k_arr` and utilizing the method `pmf` of the `Urn` class.

```
k_arr = [[5, 5, 4 ,1], [10, 1, 2, 2], [13, 0, 2, 0]]
urn.pmf(k_arr)
```

```
array([6.21412534e-06, 2.70935969e-02, 1.61839976e-02])
```

Now let's compute the mean and variance-covariance matrix of X when $n = 6$.

```
n = 6 # number of draws
μ, Σ = urn.moments(n)
```

```
# mean
μ
```

```
array([3.95798319, 0.27731092, 1.15966387, 0.60504202])
```

```
# variance-covariance matrix
Σ
```

```
array([[ 1.31862604, -0.17907267, -0.74884935, -0.39070401],
       [-0.17907267,  0.25891399, -0.05246715, -0.02737417],
       [-0.74884935, -0.05246715,  0.91579029, -0.11447379],
       [-0.39070401, -0.02737417, -0.11447379,  0.53255196]])
```

We can simulate a large sample and verify that sample means and covariances closely approximate the population means and covariances.

```
size = 10_000_000
sample = urn.simulate(n, size=size)
```

```
# mean
np.mean(sample, 0)
```

```
array([3.9572732, 0.2773802, 1.1604012, 0.6049454])
```

```
# variance covariance matrix
np.cov(sample.T)
```

```
array([[ 1.31879095, -0.17919355, -0.74891794, -0.39067946],
       [-0.17919355,  0.25886705, -0.05246242, -0.02721108],
       [-0.74891794, -0.05246242,  0.91595635, -0.11457598],
       [-0.39067946, -0.02721108, -0.11457598,  0.53246652]])
```

Evidently, the sample means and covariances approximate their population counterparts well.

18.3.3 Quality of Normal Approximation

To judge the quality of a multivariate normal approximation to the multivariate hypergeometric distribution, we draw a large sample from a multivariate normal distribution with the mean vector and covariance matrix for the corresponding multivariate hypergeometric distribution and compare the simulated distribution with the population multivariate hypergeometric distribution.

```
sample_normal = np.random.multivariate_normal(μ, Σ, size=size)
```

```
def bivariate_normal(x, y, μ, Σ, i, j):

    μ_x, μ_y = μ[i], μ[j]
    σ_x, σ_y = np.sqrt(Σ[i, i]), np.sqrt(Σ[j, j])
    σ_xy = Σ[i, j]

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)
```

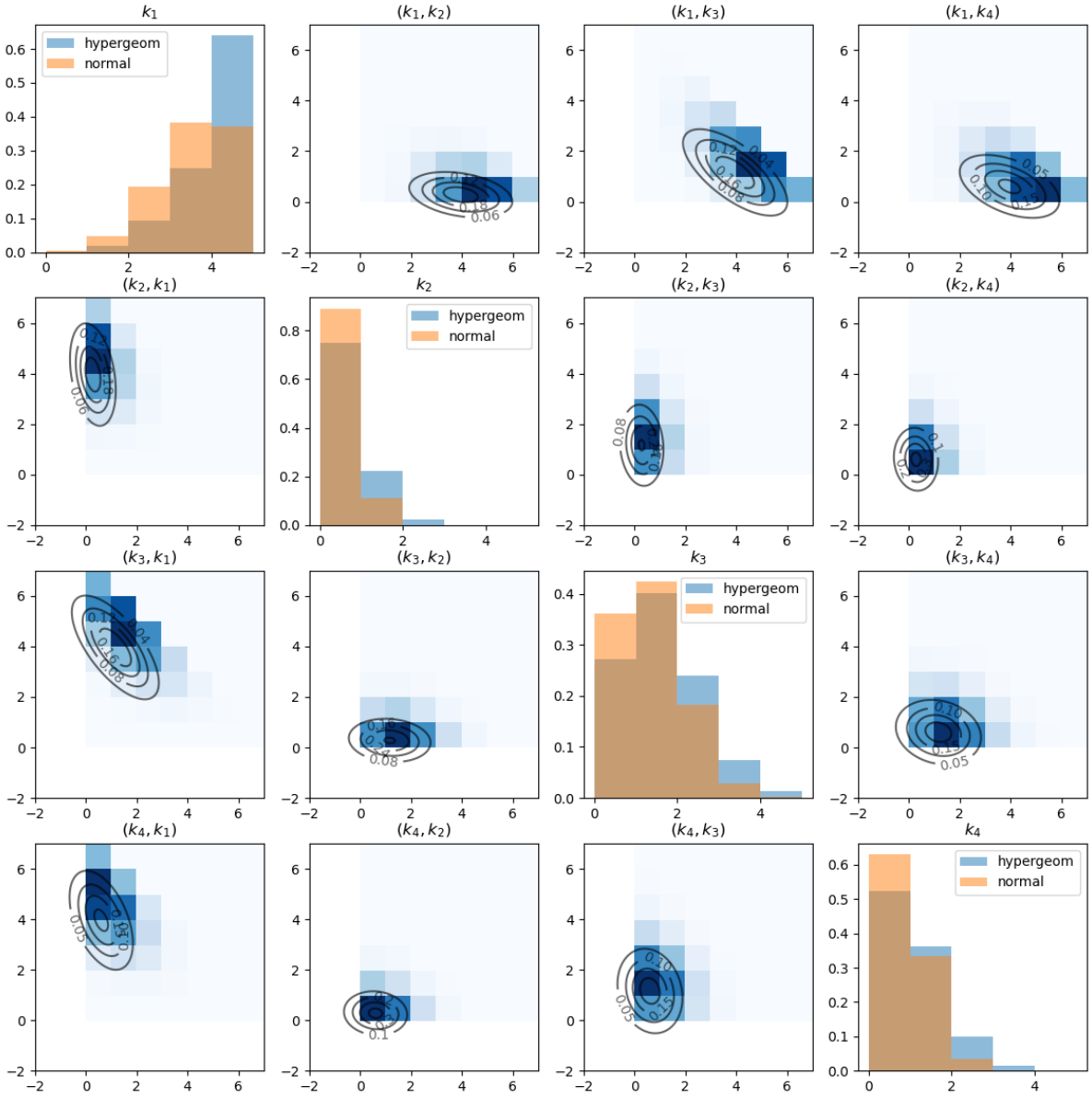
(continues on next page)

(continued from previous page)

```
    return np.exp(-z / (2 * (1 - ρ**2))) / denom
```

```
@njit
def count(vec1, vec2, n):
    size = sample.shape[0]

    count_mat = np.zeros((n+1, n+1))
    for i in prange(size):
        count_mat[vec1[i], vec2[i]] += 1

    return count_mat
```

```
c = urn.c
fig, axs = plt.subplots(c, c, figsize=(14, 14))

# grids for ploting the bivariate Gaussian
x_grid = np.linspace(-2, n+1, 100)
y_grid = np.linspace(-2, n+1, 100)
X, Y = np.meshgrid(x_grid, y_grid)

for i in range(c):
    axs[i, i].hist(sample[:, i], bins=np.arange(0, n, 1), alpha=0.5, density=True,␣
 ↪label='hypergeom')
    axs[i, i].hist(sample_normal[:, i], bins=np.arange(0, n, 1), alpha=0.5,␣
 ↪density=True, label='normal')
    axs[i, i].legend()
    axs[i, i].set_title('$k_{' +str(i+1) +'}$')
    for j in range(c):
        if i == j:
            continue

        # bivariate Gaussian density function
        Z = bivariate_normal(X, Y, μ, Σ, i, j)
        cs = axs[i, j].contour(X, Y, Z, 4, colors="black", alpha=0.6)
        axs[i, j].clabel(cs, inline=1, fontsize=10)

        # empirical multivariate hypergeometric distrbution
        count_mat = count(sample[:, i], sample[:, j], n)
        axs[i, j].pcolor(count_mat.T/size, cmap='Blues')
        axs[i, j].set_title('$(k_{' +str(i+1) +'}, k_{' + str(j+1) + '})$')

plt.show()
```



The diagonal graphs plot the marginal distributions of k_i for each i using histograms.

Note the substantial differences between hypergeometric distribution and the approximating normal distribution.

The off-diagonal graphs plot the empirical joint distribution of k_i and k_j for each pair (i, j).

The darker the blue, the more data points are contained in the corresponding cell. (Note that k_i is on the x-axis and k_j is on the y-axis).

The contour maps plot the bivariate Gaussian density function of $\left(k_i, k_j\right)$ with the population mean and covariance given by slices of μ and Σ that we computed above.

Let's also test the normality for each k_i using `scipy.stats.normaltest` that implements D'Agostino and Pearson's test that combines skew and kurtosis to form an omnibus test of normality.

The null hypothesis is that the sample follows normal distribution.

> `normaltest` returns an array of p-values associated with tests for each k_i sample.

```
test_multihyper = normaltest(sample)
test_multihyper.pvalue
```

```
array([0., 0., 0., 0.])
```

As we can see, all the p-values are almost 0 and the null hypothesis is soundly rejected.

By contrast, the sample from normal distribution does not reject the null hypothesis.

```
test_normal = normaltest(sample_normal)
test_normal.pvalue
```

```
array([0.55635946, 0.28900883, 0.95920753, 0.84097065])
```

The lesson to take away from this is that the normal approximation is imperfect.

CHAPTER

NINETEEN

FAULT TREE UNCERTAINTIES

19.1 Overview

This lecture puts elementary tools to work to approximate probability distributions of the annual failure rates of a system consisting of a number of critical parts.

We'll use log normal distributions to approximate probability distributions of critical component parts.

To approximate the probability distribution of the **sum** of n log normal probability distributions that describes the failure rate of the entire system, we'll compute the convolution of those n log normal probability distributions.

We'll use the following concepts and tools:

- log normal distributions
- the convolution theorem that describes the probability distribution of the sum independent random variables
- fault tree analysis for approximating a failure rate of a multi-component system
- a hierarchical probability model for describing uncertain probabilities
- Fourier transforms and inverse Fourier tranforms as efficient ways of computing convolutions of sequences

For more about Fourier transforms see this quantecon lecture Circulant Matrices as well as these lecture Covariance Stationary Processes and Estimation of Spectra.

El-Shanawany, Ardron, and Walker [ESAW18] and Greenfield and Sargent [GS93] used some of the methods described here to approximate probabilities of failures of safety systems in nuclear facilities.

These methods respond to some of the recommendations made by Apostolakis [Apo90] for constructing procedures for quantifying uncertainty about the reliability of a safety system.

We'll start by bringing in some Python machinery.

```
!pip install tabulate
```

```
Requirement already satisfied: tabulate in /opt/conda/envs/quantecon/lib/python3.
↪11/site-packages (0.8.10)
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and␣
↪conflicting behaviour with the system package manager. It is recommended to use␣
↪a virtual environment instead: https://pip.pypa.io/warnings/venv
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import fftconvolve
from tabulate import tabulate
import time
```

```
np.set_printoptions(precision=3, suppress=True)
```

19.2 Log normal distribution

If a random variable x follows a normal distribution with mean μ and variance σ^2, then the natural logarithm of x, say $y = \log(x)$, follows a **log normal distribution** with parameters μ, σ^2.

Notice that we said **parameters** and not **mean and variance** μ, σ^2.

- μ and σ^2 are the mean and variance of $x = \exp(y)$
- they are **not** the mean and variance of y
- instead, the mean of y is $e^{\mu+\frac{1}{2}\sigma^2}$ and the variance of y is $(e^{\sigma^2}-1)e^{2\mu+\sigma^2}$

A log normal random variable y is nonnegative.

The density for a log normal random variate y is

$$f(y) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(\frac{-(\log y - \mu)^2}{2\sigma^2}\right)$$

for $y \geq 0$.

Important features of a log normal random variable are

mean:	$e^{\mu+\frac{1}{2}\sigma^2}$
variance:	$(e^{\sigma^2}-1)e^{2\mu+\sigma^2}$
median:	e^{μ}
mode:	$e^{\mu-\sigma^2}$
.95 quantile:	$e^{\mu+1.645\sigma}$
.95-.05 quantile ratio:	$e^{1.645\sigma}$

Recall the following *stability* property of two independent normally distributed random variables:

If x_1 is normal with mean μ_1 and variance σ_1^2 and x_2 is independent of x_1 and normal with mean μ_2 and variance σ_2^2, then $x_1 + x_2$ is normally distributed with mean $\mu_1 + \mu_2$ and variance $\sigma_1^2 + \sigma_2^2$.

Independent log normal distributions have a different *stability* property.

The **product** of independent log normal random variables is also log normal.

In particular, if y_1 is log normal with parameters (μ_1, σ_1^2) and y_2 is log normal with parameters (μ_2, σ_2^2), then the product $y_1 y_2$ is log normal with parameters $(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.

Note: While the product of two log normal distributions is log normal, the **sum** of two log normal distributions is **not** log normal.

This observation sets the stage for challenge that confronts us in this lecture, namely, to approximate probability distributions of **sums** of independent log normal random variables.

To compute the probability distribution of the sum of two log normal distributions, we can use the following convolution property of a probability distribution that is a sum of independent random variables.

19.3 The Convolution Property

Let x be a random variable with probability density $f(x)$, where $x \in \mathbf{R}$.

Let y be a random variable with probability density $g(y)$, where $y \in \mathbf{R}$.

Let x and y be independent random variables and let $z = x + y \in \mathbf{R}$.

Then the probability distribution of z is

$$h(z) = (f * g)(z) \equiv \int_{-\infty}^{\infty} f(z)g(z - \tau)d\tau$$

where $(f * g)$ denotes the **convolution** of the two functions f and g.

If the random variables are both nonnegative, then the above formula specializes to

$$h(z) = (f * g)(z) \equiv \int_{0}^{\infty} f(z)g(z - \tau)d\tau$$

Below, we'll use a discretized version of the preceding formula.

In particular, we'll replace both f and g with discretized counterparts, normalized to sum to 1 so that they are probability distributions.

- by **discretized** we mean an equally spaced sampled version

Then we'll use the following version of the above formula

$$h_n = (f * g)_n = \sum_{m=0}^{\infty} f_m g_{n-m}, n \geq 0$$

to compute a discretized version of the probability distribution of the sum of two random variables, one with probability mass function f, the other with probability mass function g.

Before applying the convolution property to sums of log normal distributions, let's practice on some simple discrete distributions.

To take one example, let's consider the following two probability distributions

$$f_j = \text{Prob}(X = j), j = 0, 1$$

and

$$g_j = \text{Prob}(Y = j), j = 0, 1, 2, 3$$

and

$$h_j = \text{Prob}(Z \equiv X + Y = j), j = 0, 1, 2, 3, 4$$

The convolution property tells us that

$$h = f * g = g * f$$

Let's compute an example using the `numpy.convolve` and `scipy.signal.fftconvolve`.

```
f = [.75, .25]
g = [0., .6,  0., .4]
h = np.convolve(f,g)
hf = fftconvolve(f,g)

print("f = ", f,  ", np.sum(f) = ", np.sum(f))
print("g = ", g, ", np.sum(g) = ", np.sum(g))
print("h = ", h, ", np.sum(h) = ", np.sum(h))
print("hf = ", hf, ",np.sum(hf) = ", np.sum(hf))
```

```
f =  [0.75, 0.25] , np.sum(f) =  1.0
g =  [0.0, 0.6, 0.0, 0.4] , np.sum(g) =  1.0
h =  [0.   0.45 0.15 0.3  0.1 ] , np.sum(h) =  1.0
hf =  [0.   0.45 0.15 0.3  0.1 ] ,np.sum(hf) =  1.0000000000000002
```

A little later we'll explain some advantages that come from using `scipy.signal.ftconvolve` rather than `numpy.convolve`.numpy program convolve.

They provide the same answers but `scipy.signal.ftconvolve` is much faster.

That's why we rely on it later in this lecture.

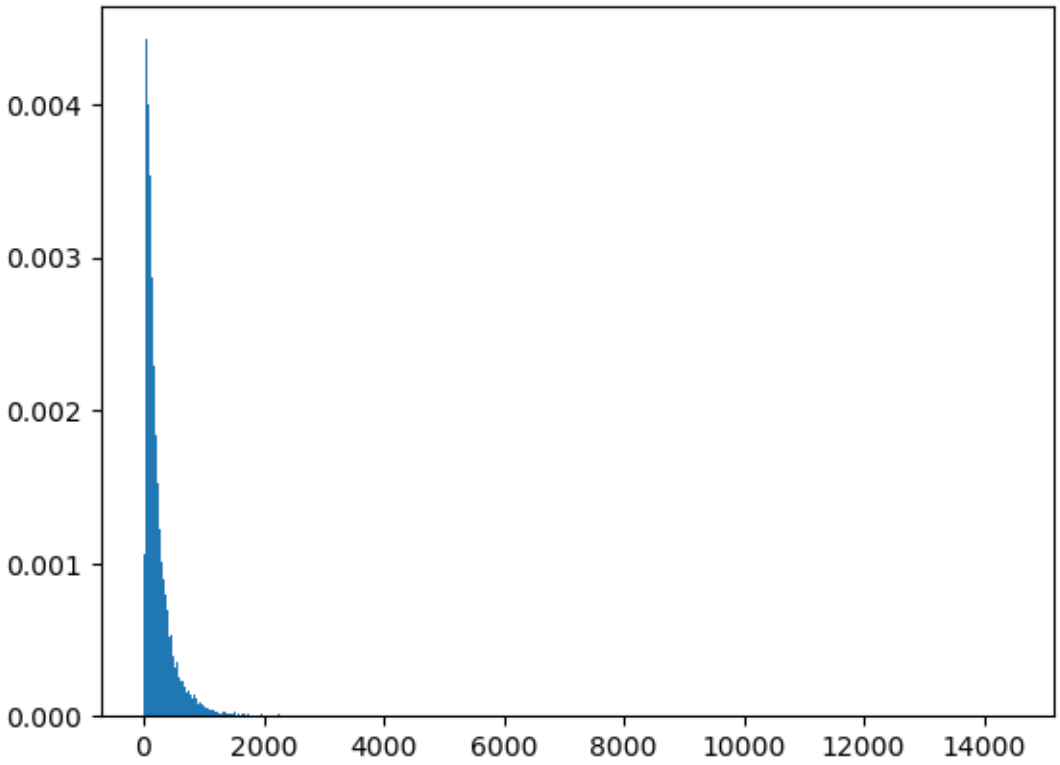
19.4 Approximating Distributions

We'll construct an example to verify that discretized distributions can do a good job of approximating samples drawn from underlying continuous distributions.

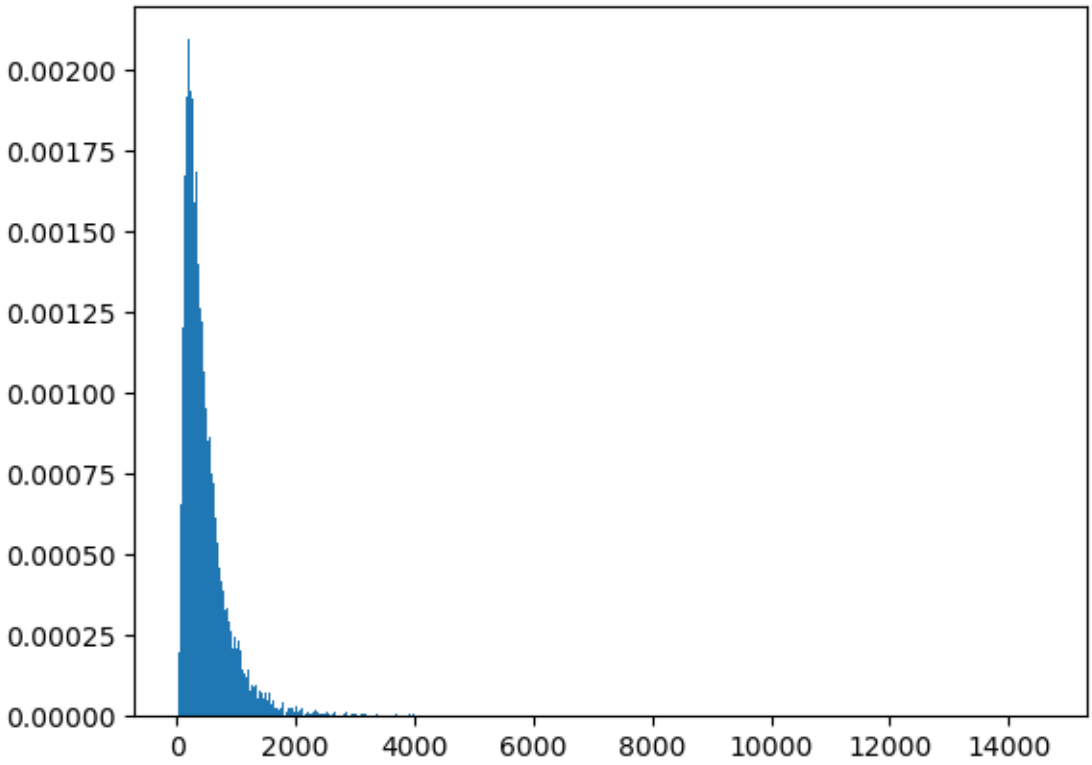
We'll start by generating samples of size 25000 of three independent log normal random variates as well as pairwise and triple-wise sums.

Then we'll plot histograms and compare them with convolutions of appropriate discretized log normal distributions.

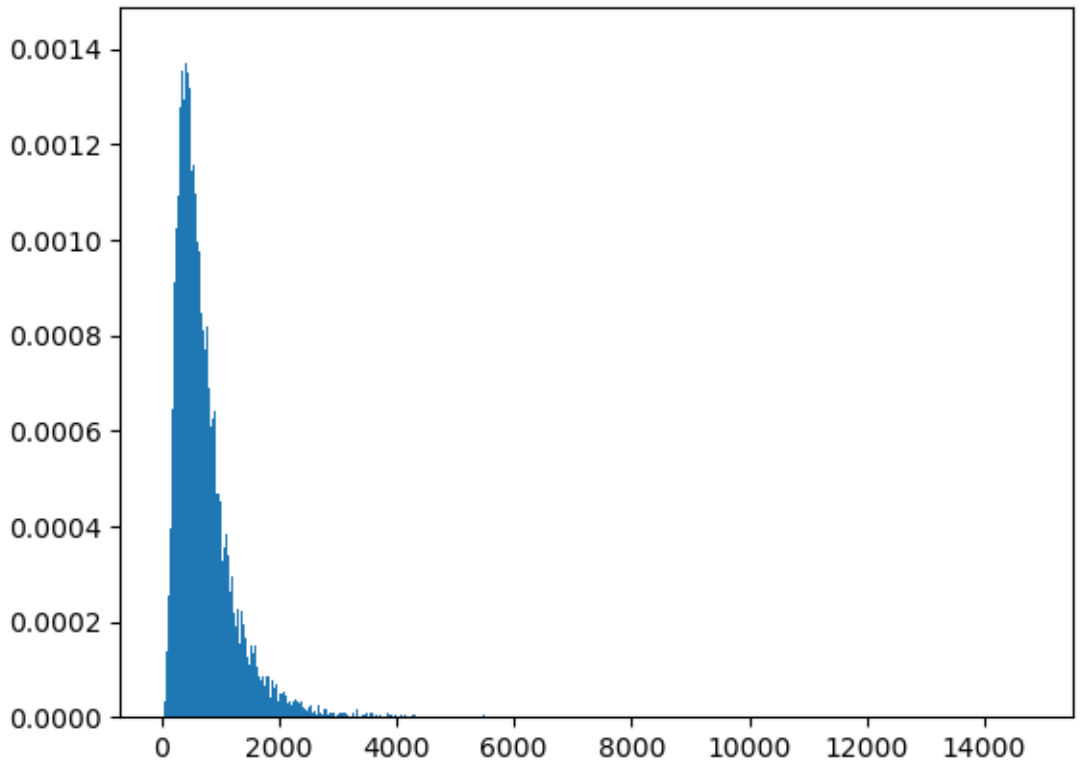
```
## create sums of two and three log normal random variates ssum2 = s1 + s2 and ssum3
 ↪= s1 + s2 + s3


mu1, sigma1 = 5., 1. # mean and standard deviation
s1 = np.random.lognormal(mu1, sigma1, 25000)

mu2, sigma2 = 5., 1. # mean and standard deviation
s2 = np.random.lognormal(mu2, sigma2, 25000)

mu3, sigma3 = 5., 1. # mean and standard deviation
s3 = np.random.lognormal(mu3, sigma3, 25000)

ssum2 = s1 + s2

ssum3 = s1 + s2 + s3

count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')
```




```
count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
```



```
count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
```




```
samp_mean2 = np.mean(s2)
pop_mean2 = np.exp(mu2+ (sigma2**2)/2)

pop_mean2, samp_mean2, mu2, sigma2
```

```
(244.69193226422038, 243.21202968536068, 5.0, 1.0)
```

Here are helper functions that create a discretized version of a log normal probability density function.

```
def p_log_normal(x,μ,σ):
    p = 1 / (σ*x*np.sqrt(2*np.pi)) * np.exp(-1/2*((np.log(x) - μ)/σ)**2)
    return p

def pdf_seq(μ,σ,I,m):
    x = np.arange(1e-7,I,m)
    p_array = p_log_normal(x,μ,σ)
    p_array_norm = p_array/np.sum(p_array)
    return p_array,p_array_norm,x
```

Now we shall set a grid length I and a grid increment size $m = 1$ for our discretizations.

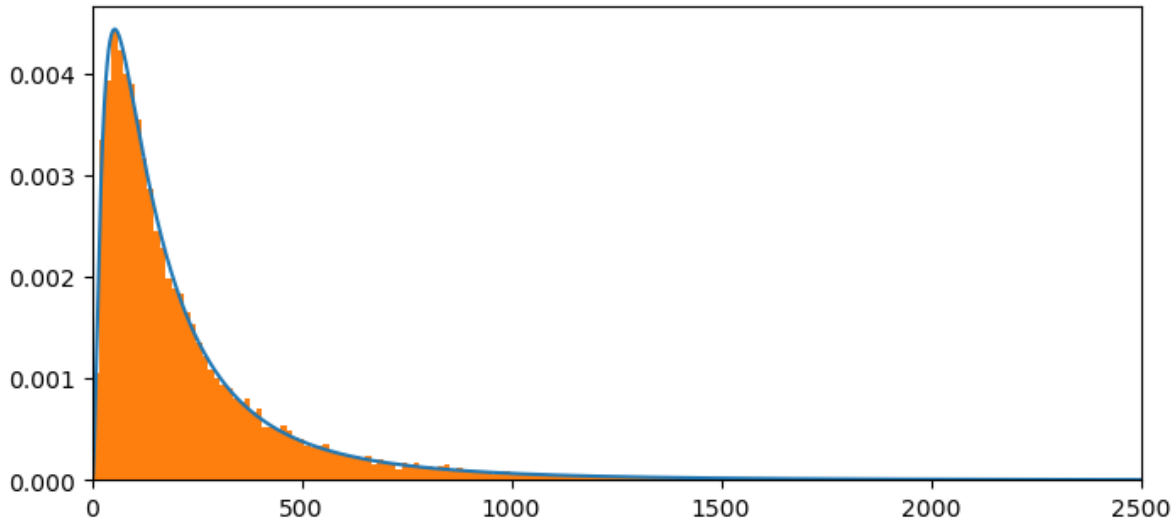
Note: We set I equal to a power of two because we want to be free to use a Fast Fourier Transform to compute a convolution of two sequences (discrete distributions).

We recommend experimenting with different values of the power p of 2.

Setting it to 15 rather than 12, for example, improves how well the discretized probability mass function approximates the original continuous probability density function being studied.

```
p=15
I = 2**p # Truncation value
m = .1 # increment size
```

```
## Cell to check -- note what happens when don't normalize!
## things match up without adjustment. Compare with above

p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
## compute number of points to evaluate the probability mass function
NT = x.size

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],p1[:int(NT)],label = '')
plt.xlim(0,2500)
count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')

plt.show()
```




```
# Compute mean from discretized pdf and compare with the theoretical value

mean= np.sum(np.multiply(x[:NT],p1_norm[:NT]))
meantheory = np.exp(mu1+.5*sigma1**2)
mean, meantheory
```

```
(244.69059898302908, 244.69193226422038)
```

19.5 Convolving Probability Mass Functions

Now let's use the convolution theorem to compute the probability distribution of a sum of the two log normal random variables we have parameterized above.

We'll also compute the probability of a sum of three log normal distributions constructed above.

Before we do these things, we shall explain our choice of Python algorithm to compute a convolution of two sequences.

Because the sequences that we convolve are long, we use the `scipy.signal.fftconvolve` function rather than the numpy.convove function.

These two functions give virtually equivalent answers but for long sequences `scipy.signal.fftconvolve` is much faster.

The program `scipy.signal.fftconvolve` uses fast Fourier transforms and their inverses to calculate convolutions.

Let's define the Fourier transform and the inverse Fourier transform.

The **Fourier transform** of a sequence $\{x_t\}_{t=0}^{T-1}$ is a sequence of complex numbers $\{x(\omega_j)\}_{j=0}^{T-1}$ given by

$$x(\omega_j) = \sum_{t=0}^{T-1} x_t \exp(-i\omega_j t) \tag{19.1}$$

where $\omega_j = \frac{2\pi j}{T}$ for $j = 0, 1, \ldots, T-1$.

The **inverse Fourier transform** of the sequence $\{x(\omega_j)\}_{j=0}^{T-1}$ is

$$x_t = T^{-1} \sum_{j=0}^{T-1} x(\omega_j) \exp(i\omega_j t) \tag{19.2}$$

The sequences $\{x_t\}_{t=0}^{T-1}$ and $\{x(\omega_j)\}_{j=0}^{T-1}$ contain the same information.

The pair of equations (19.1) and (19.2) tell how to recover one series from its Fourier partner.

The program `scipy.signal.fftconvolve` deploys the theorem that a convolution of two sequences $\{f_k\}, \{g_k\}$ can be computed in the following way:

- Compute Fourier transforms $F(\omega), G(\omega)$ of the $\{f_k\}$ and $\{g_k\}$ sequences, respectively
- Form the product $H(\omega) = F(\omega)G(\omega)$
- The convolution of $f * g$ is the inverse Fourier transform of $H(\omega)$

The **fast Fourier transform** and the associated **inverse fast Fourier transform** execute these calculations very quickly.

This is the algorithm that `scipy.signal.fftconvolve` uses.

Let's do a warmup calculation that compares the times taken by `numpy.convove` and `scipy.signal.fftconvolve`.

```
p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)

tic = time.perf_counter()

c1 = np.convolve(p1_norm,p2_norm)
c2 = np.convolve(c1,p3_norm)
```

(continues on next page)

(continued from previous page)

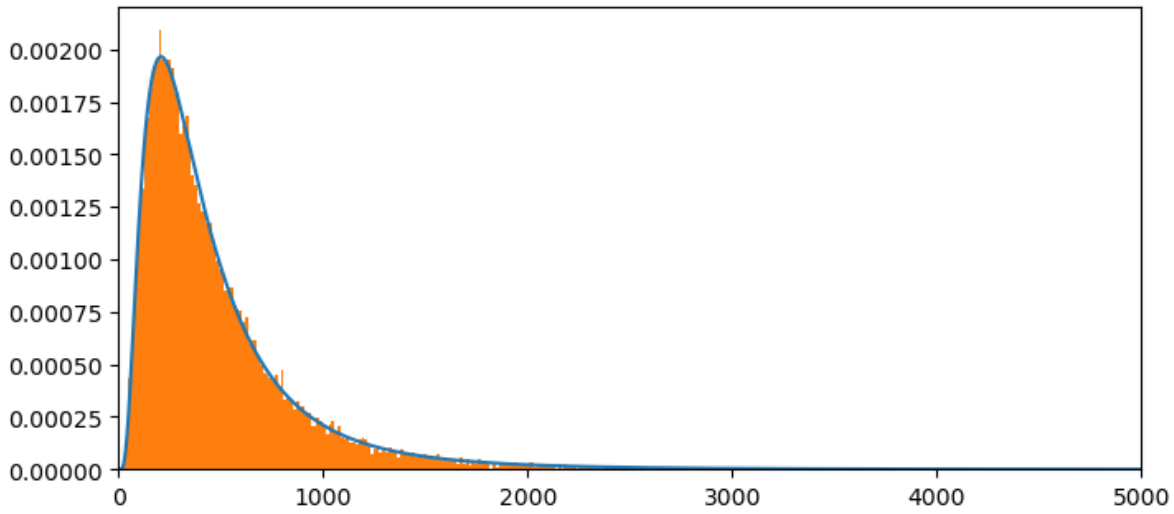
```
toc = time.perf_counter()

tdiff1 = toc - tic

tic = time.perf_counter()

c1f = fftconvolve(p1_norm,p2_norm)
c2f = fftconvolve(c1f,p3_norm)
toc = time.perf_counter()

toc = time.perf_counter()

tdiff2 = toc - tic

print("time with np.convolve = ", tdiff1,  "; time with fftconvolve = ",  tdiff2)
```

```
time with np.convolve =  44.894253018999734 ; time with fftconvolve =  0.
↪1635756340001535
```

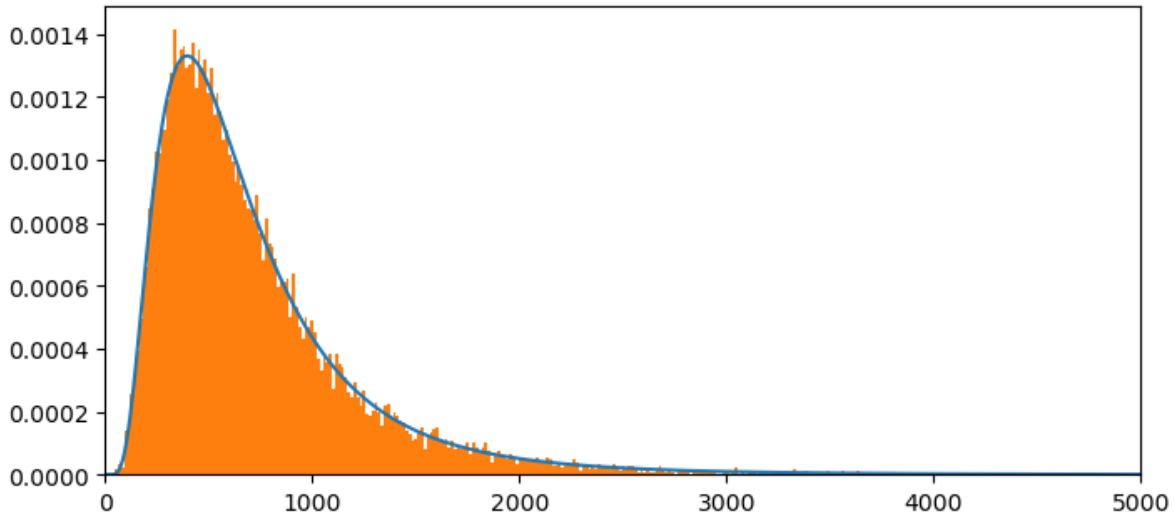
The fast Fourier transform is two orders of magnitude faster than `numpy.convolve`

Now let's plot our computed probability mass function approximation for the sum of two log normal random variables against the histogram of the sample that we formed above.

```
NT= np.size(x)

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c1f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```



```
NT= np.size(x)
plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c2f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```




```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c1f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 2*meantheory
```

```
(489.3810974093853, 489.38386452844077)
```

```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c2f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 3*meantheory
```

```
(734.0714863312277, 734.0757967926611)
```

19.6 Failure Tree Analysis

We shall soon apply the convolution theorem to compute the probability of a **top event** in a failure tree analysis.

Before applying the convolution theorem, we first describe the model that connects constituent events to the **top** end whose failure rate we seek to quantify.

The model is an example of the widely used **failure tree analysis** described by El-Shanawany, Ardron, and Walker [ESAW18].

To construct the statistical model, we repeatedly use what is called the **rare event approximation**.

We want to compute the probabilty of an event $A \cup B$.

- the union $A \cup B$ is the event that A OR B occurs

A law of probability tells us that A OR B occurs with probability

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

where the intersection $A \cap B$ is the event that A **AND** B both occur and the union $A \cup B$ is the event that A **OR** B occurs.

If A and B are independent, then

$$P(A \cap B) = P(A)P(B)$$

If $P(A)$ and $P(B)$ are both small, then $P(A)P(B)$ is even smaller.

The **rare event approximation** is

$$P(A \cup B) \approx P(A) + P(B)$$

This approximation is widely used in evaluating system failures.

19.7 Application

A system has been designed with the feature a system failure occurs when **any** of n critical components fails.

The failure probability $P(A_i)$ of each event A_i is small.

We assume that failures of the components are statistically independent random variables.

We repeatedly apply a **rare event approximation** to obtain the following formula for the problem of a system failure:

$$P(F) \approx P(A_1) + P(A_2) + \cdots + P(A_n)$$

or

$$P(F) \approx \sum_{i=1}^{n} P(A_i) \tag{19.3}$$

Probabilities for each event are recorded as failure rates per year.

19.8 Failure Rates Unknown

Now we come to the problem that really interests us, following [ESAW18] and Greenfield and Sargent [GS93] in the spirit of Apostolakis [Apo90].

The constituent probabilities or failure rates $P(A_i)$ are not known a priori and have to be estimated.

We address this problem by specifying **probabilities of probabilities** that capture one notion of not knowing the constituent probabilities that are inputs into a failure tree analysis.

Thus, we assume that a system analyst is uncertain about the failure rates $P(A_i), i = 1, \ldots, n$ for components of a system.

The analyst copes with this situation by regarding the systems failure probability $P(F)$ and each of the component probabilities $P(A_i)$ as random variables.

- dispersions of the probability distribution of $P(A_i)$ characterizes the analyst's uncertainty about the failure probability $P(A_i)$
- the dispersion of the implied probability distribution of $P(F)$ characterizes his uncertainty about the probability of a system's failure.

This leads to what is sometimes called a **hierarchical** model in which the analyst has probabilities about the probabilities $P(A_i)$.

The analyst formalizes his uncertainty by assuming that

- the failure probability $P(A_i)$ is itself a log normal random variable with parameters (μ_i, σ_i).
- failure rates $P(A_i)$ and $P(A_j)$ are statistically independent for all pairs with $i \neq j$.

The analyst calibrates the parameters (μ_i, σ_i) for the failure events $i = 1, \ldots, n$ by reading reliability studies in engineering papers that have studied historical failure rates of components that are as similar as possible to the components being used in the system under study.

The analyst assumes that such information about the observed dispersion of annual failure rates, or times to failure, can inform him of what to expect about parts' performances in his system.

The analyst assumes that the random variables $P(A_i)$ are statistically mutually independent.

The analyst wants to approximate a probability mass function and cumulative distribution function of the systems failure probability $P(F)$.

- We say probability mass function because of how we discretize each random variable, as described earlier.

The analyst calculates the probability mass function for the **top event** F, i.e., a **system failure**, by repeatedly applying the convolution theorem to compute the probability distribution of a sum of independent log normal random variables, as described in equation (19.3).

19.9 Waste Hoist Failure Rate

We'll take close to a real world example by assuming that $n = 14$.

The example estimates the annual failure rate of a critical hoist at a nuclear waste facility.

A regulatory agency wants the sytem to be designed in a way that makes the failure rate of the top event small with high probability.

This example is Design Option B-2 (Case I) described in Table 10 on page 27 of [GS93].

The table describes parameters μ_i, σ_i for fourteen log normal random variables that consist of **seven pairs** of random variables that are identically and independently distributed.

- Within a pair, parameters μ_i, σ_i are the same
- As described in table 10 of [GS93] p. 27, parameters of log normal distributions for the seven unique probabilities $P(A_i)$ have been calibrated to be the values in the following Python code:

```
mu1, sigma1 = 4.28, 1.1947
mu2, sigma2 = 3.39, 1.1947
mu3, sigma3 = 2.795, 1.1947
mu4, sigma4 = 2.717, 1.1947
mu5, sigma5 = 2.717, 1.1947
mu6, sigma6 = 1.444, 1.4632
mu7, sigma7 = -.040, 1.4632
```

Note: Because the failure rates are all very small, log normal distributions with the above parameter values actually describe $P(A_i)$ times 10^{-09}.

So the probabilities that we'll put on the x axis of the probability mass function and associated cumulative distribution function should be multiplied by 10^{-09}

To extract a table that summarizes computed quantiles, we'll use a helper function

```
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx
```

We compute the required thirteen convolutions in the following code.

(Please feel free to try different values of the power parameter p that we use to set the number of points in our grid for constructing the probability mass functions that discretize the continuous log normal distributions.)

We'll plot a counterpart to the cumulative distribution function (CDF) in figure 5 on page 29 of [GS93] and we'll also present a counterpart to their Table 11 on page 28.

```
p=15
I = 2**p # Truncation value
m =  .05 # increment size



p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)
p4,p4_norm,x = pdf_seq(mu4,sigma4,I,m)
p5,p5_norm,x = pdf_seq(mu5,sigma5,I,m)
p6,p6_norm,x = pdf_seq(mu6,sigma6,I,m)
p7,p7_norm,x = pdf_seq(mu7,sigma7,I,m)
p8,p8_norm,x = pdf_seq(mu7,sigma7,I,m)
p9,p9_norm,x = pdf_seq(mu7,sigma7,I,m)
p10,p10_norm,x = pdf_seq(mu7,sigma7,I,m)
p11,p11_norm,x = pdf_seq(mu7,sigma7,I,m)
p12,p12_norm,x = pdf_seq(mu7,sigma7,I,m)
p13,p13_norm,x = pdf_seq(mu7,sigma7,I,m)
p14,p14_norm,x = pdf_seq(mu7,sigma7,I,m)
```

(continues on next page)

(continued from previous page)

```
tic = time.perf_counter()

c1 = fftconvolve(p1_norm,p2_norm)
c2 = fftconvolve(c1,p3_norm)
c3 = fftconvolve(c2,p4_norm)
c4 = fftconvolve(c3,p5_norm)
c5 = fftconvolve(c4,p6_norm)
c6 = fftconvolve(c5,p7_norm)
c7 = fftconvolve(c6,p8_norm)
c8 = fftconvolve(c7,p9_norm)
c9 = fftconvolve(c8,p10_norm)
c10 = fftconvolve(c9,p11_norm)
c11 = fftconvolve(c10,p12_norm)
c12 = fftconvolve(c11,p13_norm)
c13 = fftconvolve(c12,p14_norm)

toc = time.perf_counter()

tdiff13 = toc - tic

print("time for 13 convolutions = ", tdiff13)
```

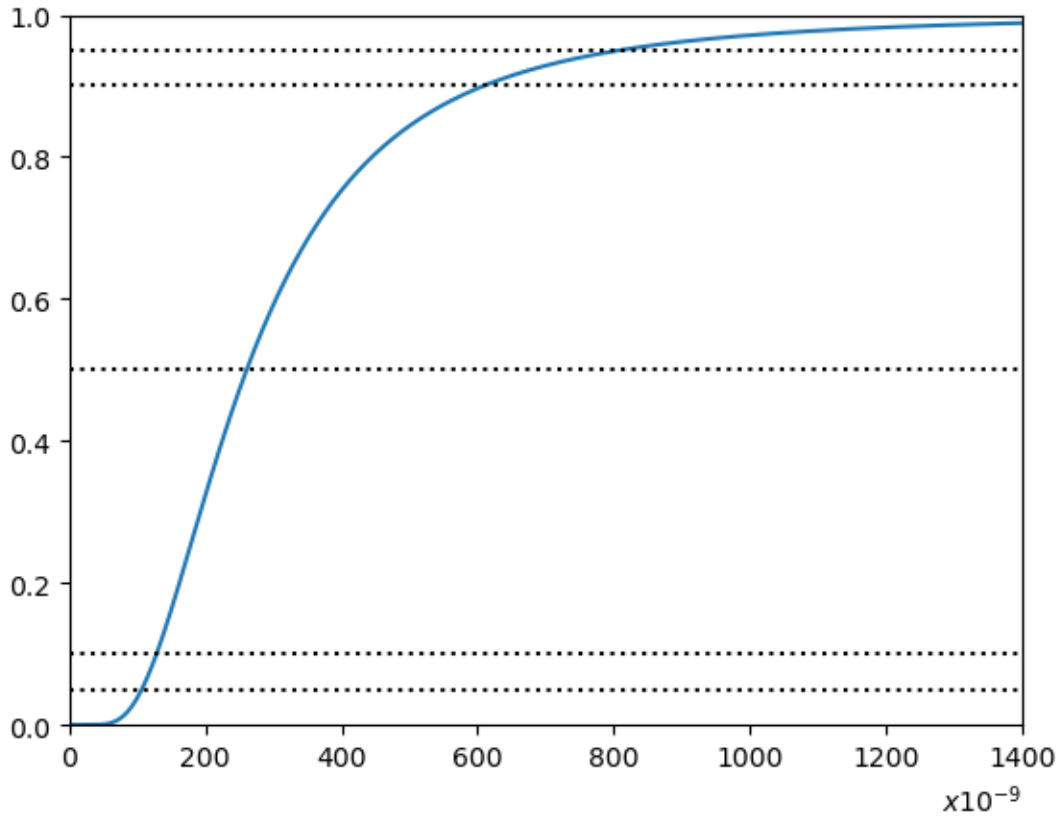
```
time for 13 convolutions =  10.217735029999858
```

```
d13 = np.cumsum(c13)
Nx=int(1400)
plt.figure()
plt.plot(x[0:int(Nx/m)],d13[0:int(Nx/m)])  # show Yad this -- I multiplied by m --␣
↪step size
plt.hlines(0.5,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.9,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.95,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.1,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.05,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.ylim(0,1)
plt.xlim(0,Nx)
plt.xlabel("$x10^{-9}$",loc = "right")
plt.show()

x_1 = x[find_nearest(d13,0.01)]
x_5 = x[find_nearest(d13,0.05)]
x_10 = x[find_nearest(d13,0.1)]
x_50 = x[find_nearest(d13,0.50)]
x_66 = x[find_nearest(d13,0.665)]
x_85 = x[find_nearest(d13,0.85)]
x_90 = x[find_nearest(d13,0.90)]
x_95 = x[find_nearest(d13,0.95)]
x_99 = x[find_nearest(d13,0.99)]
x_9978 = x[find_nearest(d13,0.9978)]

print(tabulate([
    ['1%',f"{x_1}"],
    ['5%',f"{x_5}"],
    ['10%',f"{x_10}"],
    ['50%',f"{x_50}"],
```

(continues on next page)

(continued from previous page)

```
    ['66.5%',f"{x_66}"],
    ['85%',f"{x_85}"],
    ['90%',f"{x_90}"],
    ['95%',f"{x_95}"],
    ['99%',f"{x_99}"],
    ['99.78%',f"{x_9978}"]],
    headers = ['Percentile', 'x * 1e-9']))
```



Percentile	x * 1e-9
1%	76.15
5%	106.5
10%	128.2
50%	260.55
66.5%	338.55
85%	509.4
90%	608.8
95%	807.6
99%	1470.2
99.78%	2474.85

The above table agrees closely with column 2 of Table 11 on p. 28 of of [GS93].

Discrepancies are probably due to slight differences in the number of digits retained in inputting $\mu_i, \sigma_i, i = 1, \ldots, 14$ and in the number of points deployed in the discretizations.

CHAPTER

TWENTY

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

```
!pip install --upgrade jax jaxlib
!conda install -y -c plotly plotly plotly-orca retrying
```

Note: If you are running this on Google Colab the above cell will present an error. This is because Google Colab doesn't use Anaconda to manage the Python packages. However this lecture will still execute as Google Colab has `plotly` installed.

20.1 Overview

Substantial parts of **machine learning** and **artificial intelligence** are about

- approximating an unknown function with a known function
- estimating the known function from a set of data on the left- and right-hand variables

This lecture describes the structure of a plain vanilla **artificial neural network** (ANN) of a type that is widely used to approximate a function f that maps x in a space X into y in a space Y.

To introduce elementary concepts, we study an example in which x and y are scalars.

We'll describe the following concepts that are brick and mortar for neural networks:

- a neuron
- an activation function
- a network of neurons
- A neural network as a composition of functions
- back-propagation and its relationship to the chain rule of differential calculus

20.2 A Deep (but not Wide) Artificial Neural Network

We describe a "deep" neural network of "width" one.

Deep means that the network composes a large number of functions organized into nodes of a graph.

Width refers to the number of right hand side variables on the right hand side of the function being approximated.

Setting "width" to one means that the network composes just univariate functions.

Let $x \in \mathbb{R}$ be a scalar and $y \in \mathbb{R}$ be another scalar.

We assume that y is a nonlinear function of x:

$$y = f(x)$$

We want to approximate $f(x)$ with another function that we define recursively.

For a network of depth $N \geq 1$, each **layer** $i = 1, \ldots N$ consists of

- an input x_i
- an **affine function** $w_i x_i + bI$, where w_i is a scalar **weight** placed on the input x_i and b_i is a scalar **bias**
- an **activation function** h_i that takes $(w_i x_i + b_i)$ as an argument and produces an output x_{i+1}

An example of an activation function h is the **sigmoid** function

$$h(z) = \frac{1}{1 + e^{-z}}$$

Another popular activation function is the **rectified linear unit** (ReLU) function

$$h(z) = \max(0, z)$$

Yet another activation function is the identity function

$$h(z) = z$$

As activation functions below, we'll use the sigmoid function for layers 1 to $N - 1$ and the identity function for layer N.

To approximate a function $f(x)$ we construct $\hat{f}(x)$ by proceeding as follows.

Let

$$l_i(x) = w_i x + b_i.$$

We construct $\hat{f}$ by iterating on compositions of functions $h_i \circ l_i$:

$$f(x) \approx \hat{f}(x) = h_N \circ l_N \circ h_{N-1} \circ l_1 \circ \cdots \circ h_1 \circ l_1(x)$$

If $N > 1$, we call the right side a "deep" neural net.

The larger is the integer N, the "deeper" is the neural net.

Evidently, if we know the parameters $\{w_i, b_i\}_{i=1}^N$, then we can compute $\hat{f}(x)$ for a given $x = \tilde{x}$ by iterating on the recursion

$$x_{i+1} = h_i \circ l_i(x_i), \quad , i = 1, \ldots N \tag{20.1}$$

starting from $x_1 = \tilde{x}$.

The value of x_{N+1} that emerges from this iterative scheme equals $\hat{f}(\tilde{x})$.

20.3 Calibrating Parameters

We now consider a neural network like the one describe above with width 1, depth N, and activation functions h_i for $1 \leqslant i \leqslant N$ that map $\mathbb{R}$ into itself.

Let $\{(w_i, b_i)\}_{i=1}^N$ denote a sequence of weights and biases.

As mentioned above, for a given input x_1, our approximating function $\hat{f}$ evaluated at x_1 equals the "output" x_{N+1} from our network that can be computed by iterating on $x_{i+1} = h_i(w_i x_i + b_i)$.

For a given **prediction** $\hat{y}(x)$ and **target** $y = f(x)$, consider the loss function

$$\mathcal{L}(\hat{y}, y)(x) = \frac{1}{2}(\hat{y} - y)^2(x).$$

This criterion is a function of the parameters $\{(w_i, b_i)\}_{i=1}^N$ and the point x.

We're interested in solving the following problem:

$$\min_{\{(w_i,b_i)\}_{i=1}^N} \int \mathcal{L}(x_{N+1}, y)(x) d\mu(x)$$

where $\mu(x)$ is some measure of points $x \in \mathbb{R}$ over which we want a good approximation $\hat{f}(x)$ to $f(x)$.

Stack weights and biases into a vector of parameters p:

$$p = \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \\ \vdots \\ w_N \\ b_N \end{bmatrix}$$

Applying a "poor man's version" of a **stochastic gradient descent** algorithm for finding a zero of a function leads to the following update rule for parameters:

$$p_{k+1} = p_k - \alpha \frac{d\mathcal{L}}{dx_{N+1}} \frac{dx_{N+1}}{dp_k} \tag{20.2}$$

where $\frac{d\mathcal{L}}{dx_{N+1}} = -(x_{N+1} - y)$ and $\alpha > 0$ is a step size.

(See this and this to gather insights about how stochastic gradient descent relates to Newton's method.)

To implement one step of this parameter update rule, we want the vector of derivatives $\frac{dx_{N+1}}{dp_k}$.

In the neural network literature, this step is accomplished by what is known as **back propagation**.

20.4 Back Propagation and the Chain Rule

Thanks to properties of

- the chain and product rules for differentiation from differential calculus, and
- lower triangular matrices

back propagation can actually be accomplished in one step by

- inverting a lower triangular matrix, and
- matrix multiplication

(This idea is from the last 7 minutes of this great youtube video by MIT's Alan Edelman)

https://youtu.be/rZS2LGiurKY

Here goes.

Define the derivative of $h(z)$ with respect to z evaluated at $z = z_i$ as δ_i:

$$\delta_i = \frac{d}{dz} h(z)|_{z=z_i}$$

or

$$\delta_i = h'\left(w_i x_i + b_i\right).$$

Repeated application of the chain rule and product rule to our recursion (20.1) allows us to obtain:

$$dx_{i+1} = \delta_i \left(dw_i x_i + w_i dx_i + b_i\right)$$

After imposing $dx_1 = 0$, we get the following system of equations:

$$\begin{pmatrix} dx_2 \\ \vdots \\ dx_{N+1} \end{pmatrix} = \underbrace{\begin{pmatrix} \delta_1 w_1 & \delta_1 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \delta_N w_N & \delta_N \end{pmatrix}}_{D} \begin{pmatrix} dw_1 \\ db_1 \\ \vdots \\ dw_N \\ db_N \end{pmatrix} + \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ w_2 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & w_N & 0 \end{pmatrix}}_{L} \begin{pmatrix} dx_2 \\ \vdots \\ dx_{N+1} \end{pmatrix}$$

or

$$dx = Ddp + Ldx$$

which implies that

$$dx = (I - L)^{-1} Ddp$$

which in turn implies

$$\begin{pmatrix} dx_{N+1}/dw_1 \\ dx_{N+1}/db_1 \\ \vdots \\ dx_{N+1}/dw_N \\ dx_{N+1}/db_N \end{pmatrix} = e_N \left(I - L\right)^{-1} D.$$

We can then solve the above problem by applying our update for p multiple times for a collection of input-output pairs $\left\{(x_1^i, y^i)\right\}_{i=1}^M$ that we'll call our "training set".

20.5 Training Set

Choosing a training set amounts to a choice of measure μ in the above formulation of our function approximation problem as a minimization problem.

In this spirit, we shall use a uniform grid of, say, 50 or 200 points.

There are many possible approaches to the minimization problem posed above:

- batch gradient descent in which you use an average gradient over the training set
- stochastic gradient descent in which you sample points randomly and use individual gradients
- something in-between (so-called "mini-batch gradient descent")

The update rule (20.2) described above amounts to a stochastic gradient descent algorithm.

```
from IPython.display import Image
import jax.numpy as jnp
from jax import grad, jit, jacfwd, vmap
from jax import random
import jax
import plotly.graph_objects as go
```

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1.):
    w_key, b_key = random.split(key)
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))

# Initialize all layers for a fully-connected neural network with sizes "sizes"
def init_network_params(sizes, key):
    keys = random.split(key, len(sizes))
    return [random_layer_params(m, n, k) for m, n, k in zip(sizes[:-1], sizes[1:],␣
 ↪keys)]
```

```
def compute_xδw_seq(params, x):
    # Initialize arrays
    δ = jnp.zeros(len(params))
    xs = jnp.zeros(len(params) + 1)
    ws = jnp.zeros(len(params))
    bs = jnp.zeros(len(params))

    h = jax.nn.sigmoid

    xs = xs.at[0].set(x)
    for i, (w, b) in enumerate(params[:-1]):
        output = w * xs[i] + b
        activation = h(output[0, 0])

        # Store elements
        δ = δ.at[i].set(grad(h)(output[0, 0]))
        ws = ws.at[i].set(w[0, 0])
        bs = bs.at[i].set(b[0])
        xs = xs.at[i+1].set(activation)

    final_w, final_b = params[-1]
    preds = final_w * xs[-2] + final_b

    # Store elements
    δ = δ.at[-1].set(1.)
    ws = ws.at[-1].set(final_w[0, 0])
    bs = bs.at[-1].set(final_b[0])
    xs = xs.at[-1].set(preds[0, 0])

    return xs, δ, ws, bs
```

(continues on next page)

(continued from previous page)

```
def loss(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    preds = xs[-1]

    return 1 / 2 * (y - preds) ** 2
```

```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
param_scale = 0.1
step_size = 0.01
params = init_network_params(layer_sizes, random.PRNGKey(1))
```

```
x = 5
y = 3
xs, δ, ws, bs = compute_xδw_seq(params, x)
```

```
dxs_ad = jacfwd(lambda params, x: compute_xδw_seq(params, x)[0], argnums=0)(params, x)
dxs_ad_mat = jnp.block([dx.reshape((-1, 1)) for dx_tuple in dxs_ad for dx in dx_tuple␣
↪])[1:]
```

```
jnp.block([[δ * xs[:-1]], [δ]])
```

```
Array([[8.5726520e-03, 4.0850643e-04, 6.1021703e-01],
       [1.7145304e-03, 2.3785220e-01, 1.0000000e+00]], dtype=float32)
```

```
L = jnp.diag(δ * ws, k=-1)
L = L[1:, 1:]

D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ * xs[:-
↪1]], [δ]]).T])

dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)
```

```
# Check that the `dx` generated by the linear algebra method
# are the same as the ones generated using automatic differentiation
jnp.max(jnp.abs(dxs_ad_mat - dxs_la))
```

```
Array(0., dtype=float32)
```

```
grad_loss_ad = jnp.block([dx.reshape((-1, 1)) for dx_tuple in grad(loss)(params, x,␣
↪y) for dx in dx_tuple ])
```

```
# Check that the gradient of the loss is the same for both approaches
jnp.max(jnp.abs(-(y - xs[-1]) * dxs_la[-1] - grad_loss_ad))
```

```
Array(1.4901161e-08, dtype=float32)
```

```
@jit
def update_ad(params, x, y):
    grads = grad(loss)(params, x, y)
    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads)]

@jit
def update_la(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    N = len(params)
    L = jnp.diag(δ * ws, k=-1)
    L = L[1:, 1:]

    D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ *␣
 ↪xs[:-1]], [δ]]).T])

    dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)

    grads = -(y - xs[-1]) * dxs_la[-1]

    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads.reshape((-1, 2)))]
```

```
# Check that both updates are the same
update_la(params, x, y)
```

```
[(Array([[-1.3489482]], dtype=float32), Array([0.37956238], dtype=float32)),
 (Array([[-0.00782906]], dtype=float32), Array([0.44972023], dtype=float32)),
 (Array([[0.22937916]], dtype=float32), Array([-0.04793657], dtype=float32))]
```

```
update_ad(params, x, y)
```

```
[(Array([[-1.3489482]], dtype=float32), Array([0.37956238], dtype=float32)),
 (Array([[-0.00782906]], dtype=float32), Array([0.44972023], dtype=float32)),
 (Array([[0.22937916]], dtype=float32), Array([-0.04793657], dtype=float32))]
```

20.6 Example 1

Consider the function

$$f(x) = -3x + 2$$

on $[0.5, 3]$.

We use a uniform grid of 200 points and update the parameters for each point on the grid 300 times.

h_i is the sigmoid activation function for all layers except the final one for which we use the identity function and $N = 3$.

Weights are initialized randomly.

```
def f(x):
    return -3 * x + 2

M = 200
grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
indices = jnp.arange(M)
key = random.PRNGKey(0)

def train(params, grid, f_val, key, num_epochs=300):
    for epoch in range(num_epochs):
        key, _ = random.split(key)
        random_permutation = random.permutation(random.PRNGKey(1), indices)
        for x, y in zip(grid[random_permutation], f_val[random_permutation]):
            params = update_la(params, x, y)

    return params
```

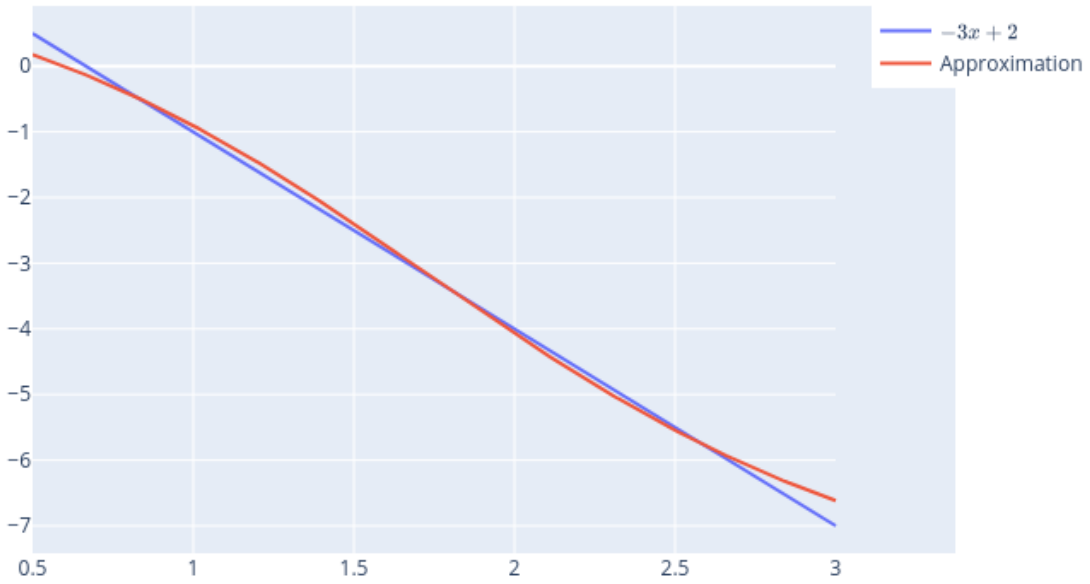
```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex1 = init_network_params(layer_sizes, key)
```

```
%%time
params_ex1 = train(params_ex1, grid, f_val, key, num_epochs=500)
```

```
CPU times: user 41.9 s, sys: 4.8 s, total: 46.7 s
Wall time: 29.8 s
```

```
predictions = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex1, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$-3x+2$'))
fig.add_trace(go.Scatter(x=grid, y=predictions, name='Approximation'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```



20.7 How Deep?

It is fun to think about how deepening the neural net for the above example affects the quality of approximation

- If the network is too deep, you'll run into the vanishing gradient problem
- Other parameters such as the step size and the number of epochs can be as important or more important than the number of layers in the situation considered in this lecture.
- Indeed, since f is a linear function of x, a one-layer network with the identity map as an activation would probably work best.

20.8 Example 2

We use the same setup as for the previous example with

$$f(x) = \log(x)$$

```
def f(x):
    return jnp.log(x)

grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
# Parameters
N = 1  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_1 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 2  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_2 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_3 = init_network_params(layer_sizes, key)
```

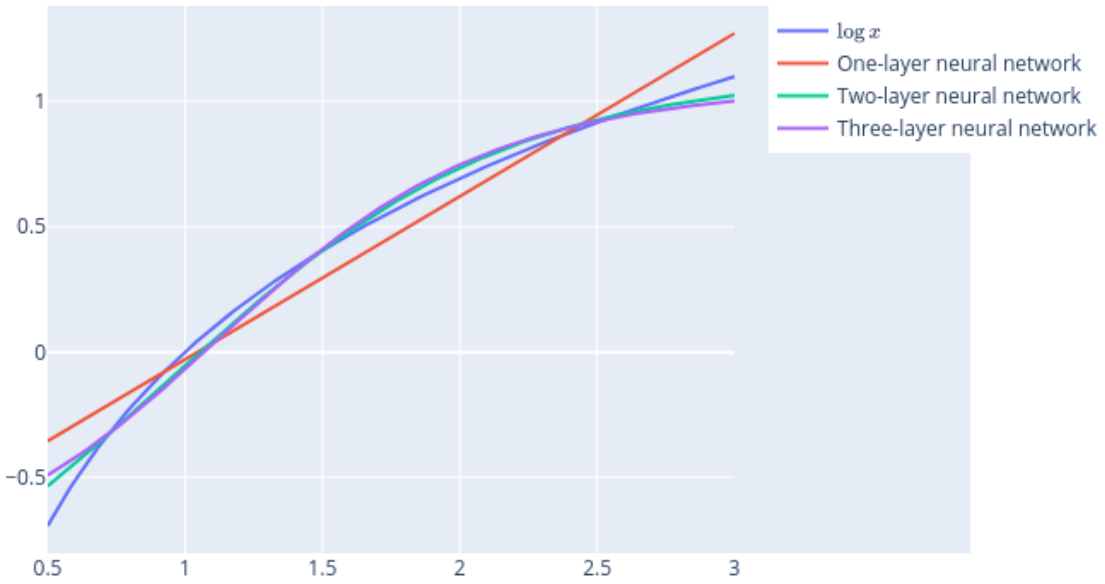
```
params_ex2_1 = train(params_ex2_1, grid, f_val, key, num_epochs=300)
```

```
params_ex2_2 = train(params_ex2_2, grid, f_val, key, num_epochs=300)
```

```
params_ex2_3 = train(params_ex2_3, grid, f_val, key, num_epochs=300)
```

```
predictions_1 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_1, grid)[0][:, -1]
predictions_2 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_2, grid)[0][:, -1]
predictions_3 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_3, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$\log{x}$'))
fig.add_trace(go.Scatter(x=grid, y=predictions_1, name='One-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_2, name='Two-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_3, name='Three-layer neural network'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```




```
## to check that gpu is activated in environment

from jax.lib import xla_bridge
print(xla_bridge.get_backend().platform)
```

```
gpu
```

Note: Cloud Environment: This lecture site is built in a server environment that doesn't have access to a `gpu` If you run this lecture locally this lets you know where your code is being executed, either via the `cpu` or the `gpu`

CHAPTER

TWENTYONE

RANDOMIZED RESPONSE SURVEYS

21.1 Overview

Social stigmas can inhibit people from confessing potentially embarrassing activities or opinions.

When people are reluctant to participate a sample survey about personally sensitive issues, they might decline to participate, and even if they do participate, they might choose to provide incorrect answers to sensitive questions.

These problems induce **selection** biases that present challenges to interpreting and designing surveys.

To illustrate how social scientists have thought about estimating the prevalence of such embarrassing activities and opinions, this lecture describes a classic approach of S. L. Warner [War65].

Warner used elementary probability to construct a way to protect the privacy of **individual** respondents to surveys while still estimating the fraction of a **collection** of individuals who have a socially stigmatized characteristic or who engage in a socially stigmatized activity.

Warner's idea was to add **noise** between the respondent's answer and the **signal** about that answer that the survey maker ultimately receives.

Knowing about the structure of the noise assures the respondent that the survey maker does not observe his answer.

Statistical properties of the noise injection procedure provide the respondent **plausible deniability**.

Related ideas underlie modern **differential privacy** systems.

(See https://en.wikipedia.org/wiki/Differential_privacy)

21.2 Warner's Strategy

As usual, let's bring in the Python modules we'll be using.

```
import numpy as np
import pandas as pd
```

Suppose that every person in population either belongs to Group A or Group B.

We want to estimate the proportion π who belong to Group A while protecting individual respondents' privacy.

Warner [War65] proposed and analyzed the following procedure.

- A random sample of n people is drawn with replacement from the population and each person is interviewed.
- Draw n random samples from the population with replacement and interview each person.

- Prepare a **random spinner** that with p probability points to the Letter A and with $(1-p)$ probability points to the Letter B.
- Each subject spins a random spinner and sees an outcome (A or B) that the interviewer does **not observe**.
- The subject states whether he belongs to the group to which the spinner points.
- If the spinner points to the group that the spinner belongs, the subject reports "yes"; otherwise he reports "no".
- The subject answers the question truthfully.

Warner constructed a maximum likelihood estimators of the proportion of the population in set A.

Let

- π : True probability of A in the population
- p : Probability that the spinner points to A
- $X_i = \begin{cases} 1, \text{ if the } i\text{th subject says yes} \\ 0, \text{ if the } i\text{th subject says no} \end{cases}$

Index the sample set so that the first n_1 report "yes", while the second $n - n_1$ report "no".

The likelihood function of a sample set is

$$L = [\pi p + (1-\pi)(1-p)]^{n_1} [(1-\pi)p + \pi(1-p)]^{n-n_1} \tag{21.1}$$

The log of the likelihood function is:

$$\log(L) = n_1 \log [\pi p + (1-\pi)(1-p)] + (n-n_1) \log [(1-\pi)p + \pi(1-p)] \tag{21.2}$$

The first-order necessary condition for maximizing the log likelihood function with respect to π is:

$$\frac{(n-n_1)(2p-1)}{(1-\pi)p + \pi(1-p)} = \frac{n_1(2p-1)}{\pi p + (1-\pi)(1-p)}$$

or

$$\pi p + (1-\pi)(1-p) = \frac{n_1}{n} \tag{21.3}$$

If $p \neq \frac{1}{2}$, then the maximum likelihood estimator (MLE) of π is:

$$\hat{\pi} = \frac{p-1}{2p-1} + \frac{n_1}{(2p-1)n} \tag{21.4}$$

We compute the mean and variance of the MLE estimator $\hat{\pi}$ to be:

$$\begin{aligned} \mathbb{E}(\hat{\pi}) &= \frac{1}{2p-1} \left[p - 1 + \frac{1}{n} \sum_{i=1}^{n} \mathbb{E} X_i \right] \\ &= \frac{1}{2p-1} [p - 1 + \pi p + (1-\pi)(1-p)] \\ &= \pi \end{aligned} \tag{21.5}$$

and

$$\begin{aligned} Var(\hat{\pi}) &= \frac{nVar(X_i)}{(2p-1)^2 n^2} \\ &= \frac{[\pi p + (1-\pi)(1-p)] [(1-\pi)p + \pi(1-p)]}{(2p-1)^2 n^2} \\ &= \frac{\frac{1}{4} + (2p^2 - 2p + \frac{1}{2})(-2\pi^2 + 2\pi - \frac{1}{2})}{(2p-1)^2 n^2} \\ &= \frac{1}{n} \left[\frac{1}{16(p-\frac{1}{2})^2} - (\pi - \frac{1}{2})^2 \right] \end{aligned} \tag{21.6}$$

Equation (21.5) indicates that $\hat{\pi}$ is an **unbiased estimator** of π while equation (21.6) tell us the variance of the estimator.

To compute a confidence interval, first rewrite (21.6) as:

$$Var(\hat{\pi}) = \frac{\frac{1}{4} - (\pi - \frac{1}{2})^2}{n} + \frac{\frac{1}{16(p-\frac{1}{2})^2} - \frac{1}{4}}{n} \tag{21.7}$$

This equation indicates that the variance of $\hat{\pi}$ can be represented as a sum of the variance due to sampling plus the variance due to the random device.

From the expressions above we can find that:

- When p is $\frac{1}{2}$, expression (21.1) degenerates to a constant.
- When p is 1 or 0, the randomized estimate degenerates to an estimator without randomized sampling.

We shall only discuss situations in which $p \in (\frac{1}{2}, 1)$

(a situation in which $p \in (0, \frac{1}{2})$ is symmetric).

From expressions (21.5) and (21.7) we can deduce that:

- The MSE of $\hat{\pi}$ decreases as p increases.

21.3 Comparing Two Survey Designs

Let's compare the preceding randomized-response method with a stylized non-randomized response method.

In our non-randomized response method, we suppose that:

- Members of Group A tells the truth with probability T_a while the members of Group B tells the truth with probability T_b
- Y_i is 1 or 0 according to whether the sample's ith member's report is in Group A or not.

Then we can estimate π as:

$$\hat{\pi} = \frac{\sum_{i=1}^{n} Y_i}{n} \tag{21.8}$$

We calculate the expectation, bias, and variance of the estimator to be:

$$\mathbb{E}(\hat{\pi}) = \pi T_a + [(1 - \pi)(1 - T_b)] \tag{21.9}$$

$$\begin{aligned} Bias(\hat{\pi}) &= \mathbb{E}(\hat{\pi} - \pi) \\ &= \pi[T_a + T_b - 2] + [1 - T_b] \end{aligned} \tag{21.10}$$

$$Var(\hat{\pi}) = \frac{[\pi T_a + (1 - \pi)(1 - T_b)]\,[1 - \pi T_a - (1 - \pi)(1 - T_b)]}{n} \tag{21.11}$$

It is useful to define a

$$\text{MSE Ratio} = \frac{\text{Mean Square Error Randomized}}{\text{Mean Square Error Regular}}$$

We can compute MSE Ratios for different survey designs associated with different parameter values.

The following Python code computes objects we want to stare at in order to make comparisons under different values of π_A and n:

```
class Comparison:
    def __init__(self, A, n):
        self.A = A
        self.n = n
        TaTb = np.array([[0.95,  1], [0.9,   1], [0.7,    1],
                         [0.5,   1], [1,  0.95], [1,    0.9],
                         [1,   0.7], [1,   0.5], [0.95, 0.95],
                         [0.9, 0.9], [0.7, 0.7], [0.5,  0.5]])
        self.p_arr = np.array([0.6, 0.7, 0.8, 0.9])
        self.p_map = dict(zip(self.p_arr, [f"MSE Ratio: p = {x}" for x in self.p_
↪arr]))
        self.template = pd.DataFrame(columns=self.p_arr)
        self.template[['T_a','T_b']] = TaTb
        self.template['Bias'] = None

    def theoretical(self):
        A = self.A
        n = self.n
        df = self.template.copy()
        df['Bias'] = A * (df['T_a'] + df['T_b'] - 2) + (1 - df['T_b'])
        for p in self.p_arr:
            df[p] = (1 / (16 * (p - 1/2)**2) - (A - 1/2)**2) / n / \
                    (df['Bias']**2 + ((A * df['T_a'] + (1 - A) * (1 - df['T_b'])) *␣
↪(1 - A * df['T_a'] - (1 - A) * (1 - df['T_b'])) / n))
            df[p] = df[p].round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df

    def MCsimulation(self, size=1000, seed=123456):
        A = self.A
        n = self.n
        df = self.template.copy()
        np.random.seed(seed)
        sample = np.random.rand(size, self.n) <= A
        random_device = np.random.rand(size, n)
        mse_rd = {}
        for p in self.p_arr:
            spinner = random_device <= p
            rd_answer = sample * spinner + (1 - sample) * (1 - spinner)
            n1 = rd_answer.sum(axis=1)
            pi_hat = (p - 1) / (2 * p - 1) + n1 / n / (2 * p - 1)
            mse_rd[p] = np.sum((pi_hat - A)**2)
        for inum, irow in df.iterrows():
            truth_a = np.random.rand(size, self.n) <= irow.T_a
            truth_b = np.random.rand(size, self.n) <= irow.T_b
            trad_answer = sample * truth_a + (1 - sample) * (1 - truth_b)
            pi_trad = trad_answer.sum(axis=1) / n
            df.loc[inum, 'Bias'] = pi_trad.mean() - A
            mse_trad = np.sum((pi_trad - A)**2)
            for p in self.p_arr:
                df.loc[inum, p] = (mse_rd[p] / mse_trad).round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df
```

Let's put the code to work for parameter values

- $\pi_A = 0.6$

- $n = 1000$

We can generate MSE Ratios theoretically using the above formulas.

We can also perform Monte Carlo simulations of a MSE Ratio.

```
cp1 = Comparison(0.6, 1000)
df1_theoretical = cp1.theoretical()
df1_theoretical
```

```
                 MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03                5.45                1.36                0.60
0.90 1.00 -0.06                1.62                0.40                0.18
0.70 1.00 -0.18                0.19                0.05                0.02
0.50 1.00 -0.30                0.07                0.02                0.01
1.00 0.95  0.02                9.82                2.44                1.08
     0.90  0.04                3.41                0.85                0.37
     0.70  0.12                0.43                0.11                0.05
     0.50  0.20                0.16                0.04                0.02
0.95 0.95 -0.01               18.25                4.54                2.00
0.90 0.90 -0.02                9.70                2.41                1.06
0.70 0.70 -0.06                1.62                0.40                0.18
0.50 0.50 -0.10                0.61                0.15                0.07

                 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03                0.33
0.90 1.00 -0.06                0.10
0.70 1.00 -0.18                0.01
0.50 1.00 -0.30                0.00
1.00 0.95  0.02                0.60
     0.90  0.04                0.21
     0.70  0.12                0.03
     0.50  0.20                0.01
0.95 0.95 -0.01                1.11
0.90 0.90 -0.02                0.59
0.70 0.70 -0.06                0.10
0.50 0.50 -0.10                0.04
```

```
df1_mc = cp1.MCsimulation()
df1_mc
```

```
                              MSE Ratio: p = 0.6 MSE Ratio: p = 0.7  \
T_a  T_b  Bias
0.95 1.00 -0.030060000000000087               5.76               1.36
0.90 1.00 -0.060045000000000015               1.73               0.41
0.70 1.00 -0.17952999999999997               0.21               0.05
0.50 1.00 -0.300077                           0.07               0.02
1.00 0.95 0.019769999999999954               10.59                2.5
     0.90 0.04005000000000014                 3.63               0.86
     0.70 0.12005199999999994                 0.46               0.11
     0.50 0.1997460000000001                  0.17               0.04
0.95 0.95 -0.010137000000000063              18.65               4.41
0.90 0.90 -0.020103000000000093              10.48               2.48
0.70 0.70 -0.060487999999999875               1.71                0.4
```

(continues on next page)

(continued from previous page)

```
0.50 0.50 -0.09934100000000001                0.66                0.16

                                  MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.030060000000000087               0.63                0.35
0.90 1.00 -0.060045000000000015               0.19                 0.1
0.70 1.00 -0.17952999999999997                0.02                0.01
0.50 1.00 -0.300077                           0.01                 0.0
1.00 0.95 0.019769999999999954                1.15                0.64
     0.90 0.04005000000000014                 0.39                0.22
     0.70 0.12005199999999994                 0.05                0.03
     0.50 0.1997460000000001                  0.02                0.01
0.95 0.95 -0.010137000000000063               2.02                1.12
0.90 0.90 -0.020103000000000093               1.14                0.63
0.70 0.70 -0.060487999999999875               0.19                 0.1
0.50 0.50 -0.09934100000000001                0.07                0.04
```

The theoretical calculations do a good job of predicting Monte Carlo results.

We see that in many situations, especially when the bias is not small, the MSE of the randomized-sampling methods is smaller than that of the non-randomized sampling method.

These differences become larger as p increases.

By adjusting parameters π_A and n, we can study outcomes in different situations.

For example, for another situation described in Warner [War65]:

- $\pi_A = 0.5$
- $n = 1000$

we can use the code

```
cp2 = Comparison(0.5, 1000)
df2_theoretical = cp2.theoretical()
df2_theoretical
```

```
                  MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.025                7.15                1.79                0.79
0.90 1.00 -0.050                2.27                0.57                0.25
0.70 1.00 -0.150                0.27                0.07                0.03
0.50 1.00 -0.250                0.10                0.02                0.01
1.00 0.95  0.025                7.15                1.79                0.79
     0.90  0.050                2.27                0.57                0.25
     0.70  0.150                0.27                0.07                0.03
     0.50  0.250                0.10                0.02                0.01
0.95 0.95  0.000               25.00                6.25                2.78
0.90 0.90  0.000               25.00                6.25                2.78
0.70 0.70  0.000               25.00                6.25                2.78
0.50 0.50  0.000               25.00                6.25                2.78

                  MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.025                0.45
0.90 1.00 -0.050                0.14
```

(continues on next page)

(continued from previous page)

```
0.70 1.00 -0.150                0.02
0.50 1.00 -0.250                0.01
1.00 0.95  0.025                0.45
     0.90  0.050                0.14
     0.70  0.150                0.02
     0.50  0.250                0.01
0.95 0.95  0.000                1.56
0.90 0.90  0.000                1.56
0.70 0.70  0.000                1.56
0.50 0.50  0.000                1.56
```

```
df2_mc = cp2.MCsimulation()
df2_mc
```

```
                                  MSE Ratio: p = 0.6 MSE Ratio: p = 0.7  \
T_a  T_b  Bias
0.95 1.00 -0.02523000000000003                  7.0                1.69
0.90 1.00 -0.05027900000000002                 2.23                0.54
0.70 1.00 -0.14986600000000005                 0.27                0.07
0.50 1.00 -0.250211                             0.1                0.02
1.00 0.95 0.024410000000000043                 7.38                1.78
     0.90 0.04983899999999997                  2.26                0.54
     0.70 0.14976900000000004                  0.27                0.07
     0.50 0.24985100000000005                   0.1                0.02
0.95 0.95 -0.00025999999999998247             24.29                5.86
0.90 0.90 -0.00010899999999997023             25.73                 6.2
0.70 0.70 -0.0004390000000000782              25.75                6.21
0.50 0.50 0.0007679999999999909               24.91                6.01

                                  MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.02523000000000003                 0.75                0.44
0.90 1.00 -0.05027900000000002                 0.24                0.14
0.70 1.00 -0.14986600000000005                 0.03                0.02
0.50 1.00 -0.250211                            0.01                0.01
1.00 0.95 0.024410000000000043                 0.79                0.46
     0.90 0.04983899999999997                  0.24                0.14
     0.70 0.14976900000000004                  0.03                0.02
     0.50 0.24985100000000005                  0.01                0.01
0.95 0.95 -0.00025999999999998247              2.59                1.52
0.90 0.90 -0.00010899999999997023              2.74                1.61
0.70 0.70 -0.0004390000000000782               2.74                1.61
0.50 0.50 0.0007679999999999909                2.65                1.56
```

We can also revisit a calculation in the concluding section of Warner [War65] in which

- $\pi_A = 0.6$
- $n = 2000$

We use the code

```
cp3 = Comparison(0.6, 2000)
df3_theoretical = cp3.theoretical()
df3_theoretical
```

```
                MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03                3.05                0.76                0.33
0.90 1.00 -0.06                0.84                0.21                0.09
0.70 1.00 -0.18                0.10                0.02                0.01
0.50 1.00 -0.30                0.03                0.01                0.00
1.00 0.95  0.02                6.03                1.50                0.66
     0.90  0.04                1.82                0.45                0.20
     0.70  0.12                0.22                0.05                0.02
     0.50  0.20                0.08                0.02                0.01
0.95 0.95 -0.01               14.12                3.51                1.55
0.90 0.90 -0.02                5.98                1.49                0.66
0.70 0.70 -0.06                0.84                0.21                0.09
0.50 0.50 -0.10                0.31                0.08                0.03

                MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03                0.19
0.90 1.00 -0.06                0.05
0.70 1.00 -0.18                0.01
0.50 1.00 -0.30                0.00
1.00 0.95  0.02                0.37
     0.90  0.04                0.11
     0.70  0.12                0.01
     0.50  0.20                0.00
0.95 0.95 -0.01                0.86
0.90 0.90 -0.02                0.36
0.70 0.70 -0.06                0.05
0.50 0.50 -0.10                0.02
```

```
df3_mc = cp3.MCsimulation()
df3_mc
```

```
                                 MSE Ratio: p = 0.6 MSE Ratio: p = 0.7  \
T_a  T_b  Bias
0.95 1.00 -0.03031649999999997               3.27                0.8
0.90 1.00 -0.0603515                         0.91               0.22
0.70 1.00 -0.18008749999999996               0.11               0.03
0.50 1.00 -0.299849                          0.04               0.01
1.00 0.95 0.01973400000000003                 6.7               1.64
     0.90 0.03976600000000008                2.01               0.49
     0.70 0.11978900000000003                0.24               0.06
     0.50 0.2001385                          0.09               0.02
0.95 0.95 -0.010474500000000053             14.78               3.61
0.90 0.90 -0.020373499999999933              6.32               1.54
0.70 0.70 -0.059945499999999985              0.92               0.23
0.50 0.50 -0.10010349999999996               0.34               0.08

                                 MSE Ratio: p = 0.8 MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03031649999999997               0.34               0.19
0.90 1.00 -0.0603515                         0.09               0.05
0.70 1.00 -0.18008749999999996               0.01               0.01
0.50 1.00 -0.299849                           0.0                0.0
1.00 0.95 0.01973400000000003                0.69               0.39
```

(continues on next page)

(continued from previous page)

```
     0.90 0.03976600000000008              0.21              0.12
     0.70 0.11978900000000003              0.02              0.01
     0.50 0.2001385                        0.01               0.0
0.95 0.95 -0.010474500000000053            1.53              0.85
0.90 0.90 -0.020373499999999933            0.65              0.36
0.70 0.70 -0.059945499999999985             0.1              0.05
0.50 0.50 -0.10010349999999996             0.03              0.02
```

Evidently, as n increases, the randomized response method does better performance in more situations.

21.4 Concluding Remarks

This QuantEcon lecture describes some alternative randomized response surveys.

That lecture presents a utilitarian analysis of those alternatives conducted by Lars Ljungqvist [Lju93].

```
import matplotlib.pyplot as plt
import numpy as np
```

CHAPTER

TWENTYTWO

EXPECTED UTILITIES OF RANDOM RESPONSES

22.1 Overview

This QuantEcon lecture describes randomized response surveys in the tradition of Warner [War65] that are designed to protect respondents' privacy.

Lars Ljungqvist [Lju93] analyzed how a respondent's decision about whether to answer truthfully depends on **expected utility**.

The lecture tells how Ljungqvist used his framework to shed light on alternative randomized response survey techniques proposed, for example, by [Lan75], [Lan76], [LW76], [And76], [FPS77], [GKAH77], [GAESH69].

22.2 Privacy Measures

We consider randomized response models with only two possible answers, "yes" and "no."

The design determines probabilities

$$\Pr(\text{yes}|A) = 1 - \Pr(\text{no}|A)$$
$$\Pr(\text{yes}|A^{'}) = 1 - \Pr(\text{no}|A^{'})$$

These design probabilities in turn can be used to compute the conditional probability of belonging to the sensitive group A for a given response, say r:

$$\Pr(A|r) = \frac{\pi_A \Pr(r|A)}{\pi_A \Pr(r|A) + (1 - \pi_A)\Pr(r|A')} \tag{22.1}$$

22.3 Zoo of Concepts

At this point we describe some concepts proposed by various researchers

22.3.1 Leysieffer and Warner(1976)

The response r is regarded as jeopardizing with respect to A or $A^{'}$ if

$$\begin{gathered} \Pr(A|r) > \pi_A \\ \text{or} \\ \Pr(A^{'}|r) > 1 - \pi_A \end{gathered} \tag{22.2}$$

From Bayes's rule:

$$\frac{\Pr(A|r)}{\Pr(A^{'}|r)} \times \frac{(1-\pi_A)}{\pi_A} = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \tag{22.3}$$

If this expression is greater (less) than unity, it follows that r is jeopardizing with respect to $A(A^{'})$. Then, the natural measure of jeopardy will be:

$$\begin{gathered} g(r|A) = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \\ \text{and} \\ g(r|A^{'}) = \frac{\Pr(r|A^{'})}{\Pr(r|A)} \end{gathered} \tag{22.4}$$

Suppose, without loss of generality, that $\Pr(\text{yes}|A) > \Pr(\text{yes}|A^{'})$, then a yes (no) answer is jeopardizing with respect $A(A^{'})$, that is,

$$\begin{gathered} g(\text{yes}|A) > 1 \\ \text{and} \\ g(\text{no}|A^{'}) > 1 \end{gathered}$$

Leysieffer and Warner proved that the variance of the estimate can only be decreased through an increase in one or both of these two measures of jeopardy.

An efficient randomized response model is, therefore, any model that attains the maximum acceptable levels of jeopardy that are consistent with cooperation of the respondents.

As a special example, Leysieffer and Warner considered "a problem in which there is no jeopardy in a no answer"; that is, $g(\text{no}|A^{'})$ can be of unlimited magnitude.

Evidently, an optimal design must have

$$\Pr(\text{yes}|A) = 1$$

which implies that

$$\Pr(A|\text{no}) = 0$$

22.3.2 Lanke(1976)

Lanke (1975) [Lan75] argued that "it is membership in Group A that people may want to hide, not membership in the complementary Group A'."

For that reason, Lanke (1976) [Lan76] argued that an appropriate measure of protection is to minimize

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{22.5}$$

Holding this measure constant, he explained under what conditions the smallest variance of the estimate was achieved with the unrelated question model or Warner's (1965) original model.

22.3.3 2.3 Fligner, Policello, and Singh

Fligner, Policello, and Singh reached similar conclusion as Lanke (1976). [FPS77]

They measured "private protection" as

$$\frac{1 - \max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\}}{1 - \pi_A} \tag{22.6}$$

22.3.4 2.4 Greenberg, Kuebler, Abernathy, and Horvitz (1977)

[GKAH77]

Greenberg, Kuebler, Abernathy, and Horvitz (1977) stressed the importance of examining the risk to respondents who do not belong to A as well as the risk to those who do belong to the sensitive group.

They defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{22.7}$$

Similarly, the hazard for an individual who does not belong to A would be

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{22.8}$$

Greenberg et al. (1977) also considered an alternative related measure of hazard that "is likely to be closer to the actual concern felt by a respondent."

The "limited hazard" for an individual in A and $A^{'}$ is

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{22.9}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{22.10}$$

This measure is just the first term in (22.7), i.e., the probability that an individual answers "yes" and is perceived to belong to A.

22.4 Respondent's Expected Utility

22.4.1 Truth Border

Key assumptions that underlie a randomized response technique for estimating the fraction of a population that belongs to A are:

- **Assumption 1**: Respondents feel discomfort from being thought of as belonging to A.
- **Assumption 2**: Respondents prefer to answer questions truthfully than to lie, so long as the cost of doing so is not too high. The cost is taken to be the discomfort in 1.

Let r_i denote individual i's response to the randomized question.

r_i can only take values "yes" or "no".

For a given design of a randomized response interview and a given belief about the fraction of the population that belongs to A, the respondent's answer is associated with a conditional probability $\Pr(A|r_i)$ that the individual belongs to A.

Given r_i and complete privacy, the individual's utility is higher if r_i represents a truthful answer rather than a lie.

In terms of a respondent's expected utility as a function of $\Pr(A|r_i)$ and r_i

- The higher is $\Pr(A|r_i)$, the lower isindividual i's expected utility.
- expected utility is higher if r_i represents a truthful answer rather than a lie

Define:

- $\phi_i \in \{\text{truth}, \text{lie}\}$, a dichotomous variable that indicates whether or not r_i is a truthful statement.
- $U_i\left(\Pr(A|r_i), \phi_i\right)$, a utility function that is differentiable in its first argument, summarizes individual i's expected utility.

Then there is an r_i such that

$$\frac{\partial U_i\left(\Pr(A|r_i), \phi_i\right)}{\partial \Pr(A|r_i)} < 0, \text{ for } \phi_i \in \{\text{truth}, \text{lie}\} \tag{22.11}$$

and

$$U_i\left(\Pr(A|r_i), \text{truth}\right) > U_i\left(\Pr(A|r_i), \text{lie}\right), \text{ for } \Pr(A|r_i) \in [0, 1] \tag{22.12}$$

Suppose now that correct answer for individual i is "yes".

Individual i would choose to answer truthfully if

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{22.13}$$

If the correct answer is "no", individual i would volunteer the correct answer only if

$$U_i\left(\Pr(A|\text{no}), \text{truth}\right) \geq U_i\left(\Pr(A|\text{yes}), \text{lie}\right) \tag{22.14}$$

Assume that

$$\Pr(A|\text{yes}) > \pi_A > \Pr(A|\text{no})$$

so that a "yes" answer increases the odds that an individual belongs to A.

Constraint (22.14) holds for sure.

Consequently, constraint (22.13) becomes the single necessary condition for individual i always to answer truthfully.

At equality, constraint $(10.\text{a})$ determines conditional probabilities that make the individual indifferent between telling the truth and lying when the correct answer is "yes":

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) = U_i\left(\Pr(A|\text{no}), \text{lie}\right) \tag{22.15}$$

Equation (22.15) defines a "truth border".

Differentiating (22.15) with respect to the conditional probabilities shows that the truth border has a positive slope in the space of conditional probabilities:

$$\frac{\partial \Pr(A|\text{no})}{\partial \Pr(A|\text{yes})} = \frac{\frac{\partial U_i(\Pr(A|\text{yes}), \text{truth})}{\partial \Pr(A|\text{yes})}}{\frac{\partial U_i(\Pr(A|\text{no}), \text{lie})}{\partial \Pr(A|\text{no})}} > 0 \tag{22.16}$$

The source of the positive relationship is:

- The individual is willing to volunteer a truthful "yes" answer so long as the utility from doing so (i.e., the left side of (22.15)) is at least as high as the utility of lying on the right side of (22.15).
- Suppose now that $\Pr(A|\text{yes})$ increases. That reduces the utility of telling the truth. To preserve indifference between a truthful answer and a lie, $\Pr(A|\text{no})$ must increase to reduce the utility of lying.

22.4.2 Drawing a Truth Border

We can deduce two things about the truth border:

- The truth border divides the space of conditional probabilities into two subsets: "truth telling" and "lying". Thus, sufficient privacy elicits a truthful answer, whereas insufficient privacy results in a lie. The truth border depends on a respondent's utility function.
- Assumptions in (22.11) and (22.11) are sufficient only to guarantee a positive slope of the truth border. The truth border can have either a concave or a convex shape.

We can draw some truth borders with the following Python code:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
x2 = np.arange(0.4**2, 1, 0.001)
y2 = (pow(x2, 0.5) - 0.4)**2
x3 = np.arange(0.4**0.5, 1, 0.001)
y3 = pow(x3**2 - 0.4, 0.5)
plt.figure(figsize=(12, 10))
plt.plot(x1, y1, 'r-', label='Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\
↪phi_i)$')
plt.fill_between(x1, 0, y1, facecolor='red', alpha=0.05)
plt.plot(x2, y2, 'b-', label='Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)^{2}
↪+f(\phi_i)$')
plt.fill_between(x2, 0, y2, facecolor='blue', alpha=0.05)
plt.plot(x3, y3, 'y-', label='Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-\sqrt{Pr(A|r_
↪i)}+f(\phi_i)$')
plt.fill_between(x3, 0, y3, facecolor='green', alpha=0.05)
plt.plot(x1, x1, ':', linewidth=2)
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.42, 0.3, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.1')
plt.show()
```

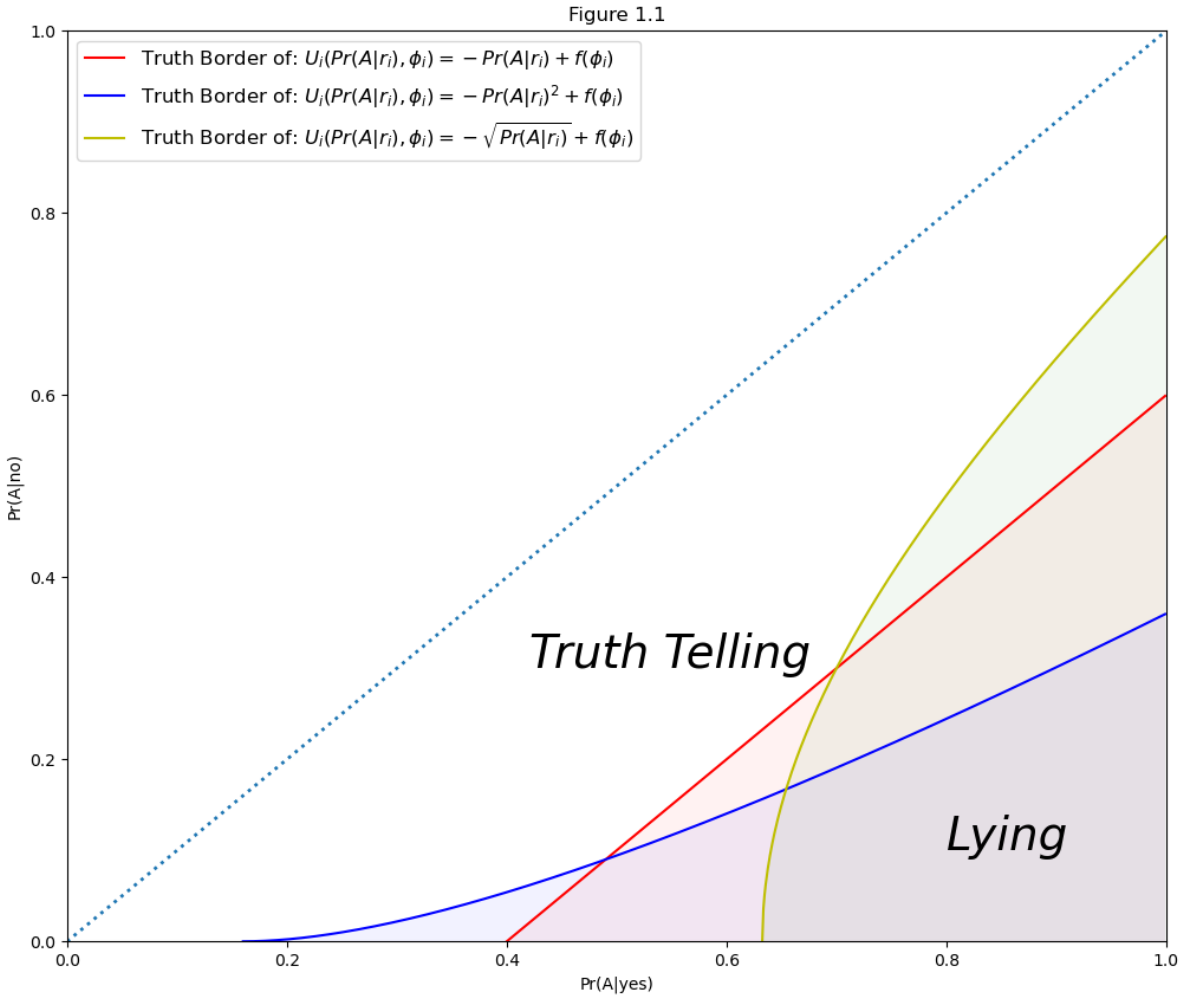


Figure 1.1 three types of truth border.

Without loss of generality, we consider the truth border:

$$U_i(\Pr(A|r_i), \phi_i) = -\Pr(A|r_i) + f(\phi_i)$$

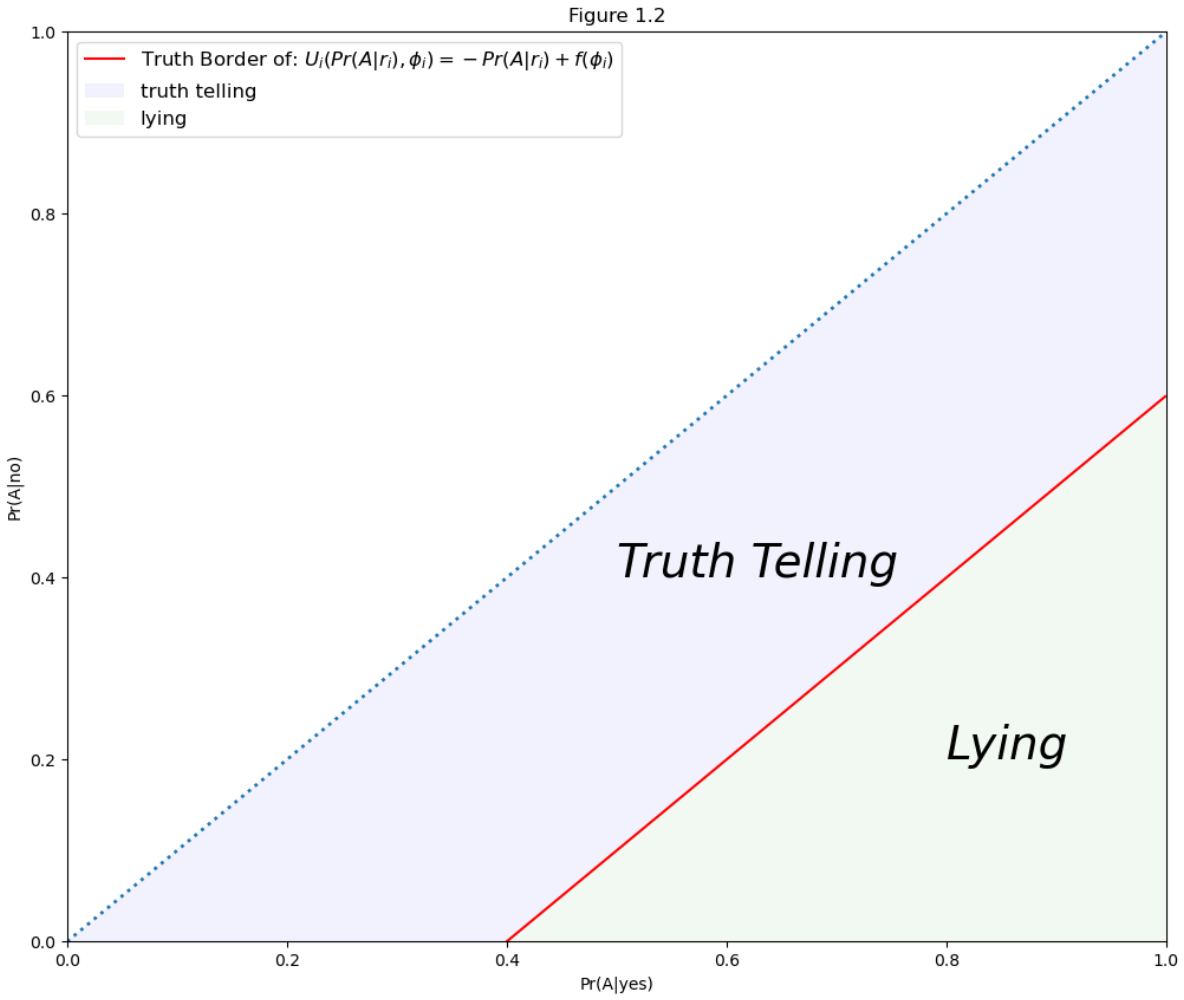
and plot the "truth telling" and "lying area" of individual i in Figure 1.2:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
z1 = x1
z2 = 0
plt.figure(figsize=(12, 10))
plt.plot(x1, y1,'r-',label='Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\phi_
↪i)$')
plt.plot(x1, x1, ':', linewidth=2)
plt.fill_between(x1, y1, z1, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x1, z2, y1, facecolor='green', alpha=0.05, label='lying')
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
```

(continues on next page)

(continued from previous page)

```
plt.text(0.5, 0.4, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.2, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.2')
plt.show()
```



22.5 Utilitarian View of Survey Design

22.5.1 Iso-variance Curves

A statistician's objective is

- to find a randomized response survey design that minimizes the bias and the variance of the estimator.

Given a design that ensures truthful answers by all respondents, Anderson(1976, Theorem 1) [And76] showed that the minimum variance estimate in the two-response model has variance

$$V(\Pr(A|\text{yes}), \Pr(A|\text{no})) = \frac{{\pi_A}^2(1-\pi_A)^2}{n} \times \frac{1}{\Pr(A|\text{yes}) - \pi_A} \times \frac{1}{\pi_A - \Pr(A|\text{no})} \tag{22.17}$$

where the random sample with replacement consists of n individuals.

We can use Expression (22.17) to draw iso-variance curves.

The following inequalities restrict the shapes of iso-variance curves:

$$\left.\frac{d\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})}\right|_{\text{constant variance}} = \frac{\pi_A - \Pr(A|\text{no})}{\Pr(A|\text{yes}) - \pi_A} > 0 \tag{22.18}$$

$$\left.\frac{d^2\,\Pr(A|\text{no})}{d\,\Pr(A|\text{yes})^2}\right|_{\text{constant variance}} = -\frac{2\left[\pi_A - \Pr(A|\text{no})\right]}{\left[\Pr(A|\text{yes}) - \pi_A\right]^2} < 0 \tag{22.19}$$

From expression (22.17), (22.18) and (22.19) we can see that:

- Variance can be reduced only by increasing the distance of $\Pr(A|\text{yes})$ and/or $\Pr(A|\text{no})$ from r_A.
- Iso-variance curves are always upward-sloping and concave.

22.5.2 Drawing Iso-variance Curves

We use Python code to draw iso-variance curves.

The pairs of conditional probabilities can be attained using Warner's (1965) model.

Note that:

- Any point on the iso-variance curves can be attained with the unrelated question model as long as the statistician can completely control the model design.
- Warner's (1965) original randomized response model is less flexible than the unrelated question model.

```
class Iso_Variance:
    def __init__(self, pi, n):
        self.pi = pi
        self.n = n

    def plotting_iso_variance_curve(self):
        pi = self.pi
        n = self.n

        nv = np.array([0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7])
        x = np.arange(0, 1, 0.001)
        x0 = np.arange(pi, 1, 0.001)
        x2 = np.arange(0, pi, 0.001)
        y1 = [pi for i in x0]
        y2 = [pi for i in x2]
        y0 = 1 / (1 + (x0 * (1 - pi)**2) / ((1 - x0) * pi**2))

        plt.figure(figsize=(12, 10))
        plt.plot(x0, y0, 'm-', label='Warner')
        plt.plot(x, x, 'c:', linewidth=2)
        plt.plot(x0, y1,'c:', linewidth=2)
        plt.plot(y2, x2, 'c:', linewidth=2)
        for i in range(len(nv)):
            y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
            plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
        plt.xlim([0, 1])
        plt.ylim([0, 0.5])
        plt.xlabel('Pr(A|yes)')
```

(continues on next page)

(continued from previous page)

```
        plt.ylabel('Pr(A|no)')
        plt.legend(loc=0, fontsize='large')
        plt.text(0.32, 0.28, "High Var", fontdict={'size':15, 'style':'italic'})
        plt.text(0.91, 0.01, "Low Var", fontdict={'size':15, 'style':'italic'})
        plt.title('Figure 2')
        plt.show()
```

Properties of iso-variance curves are:

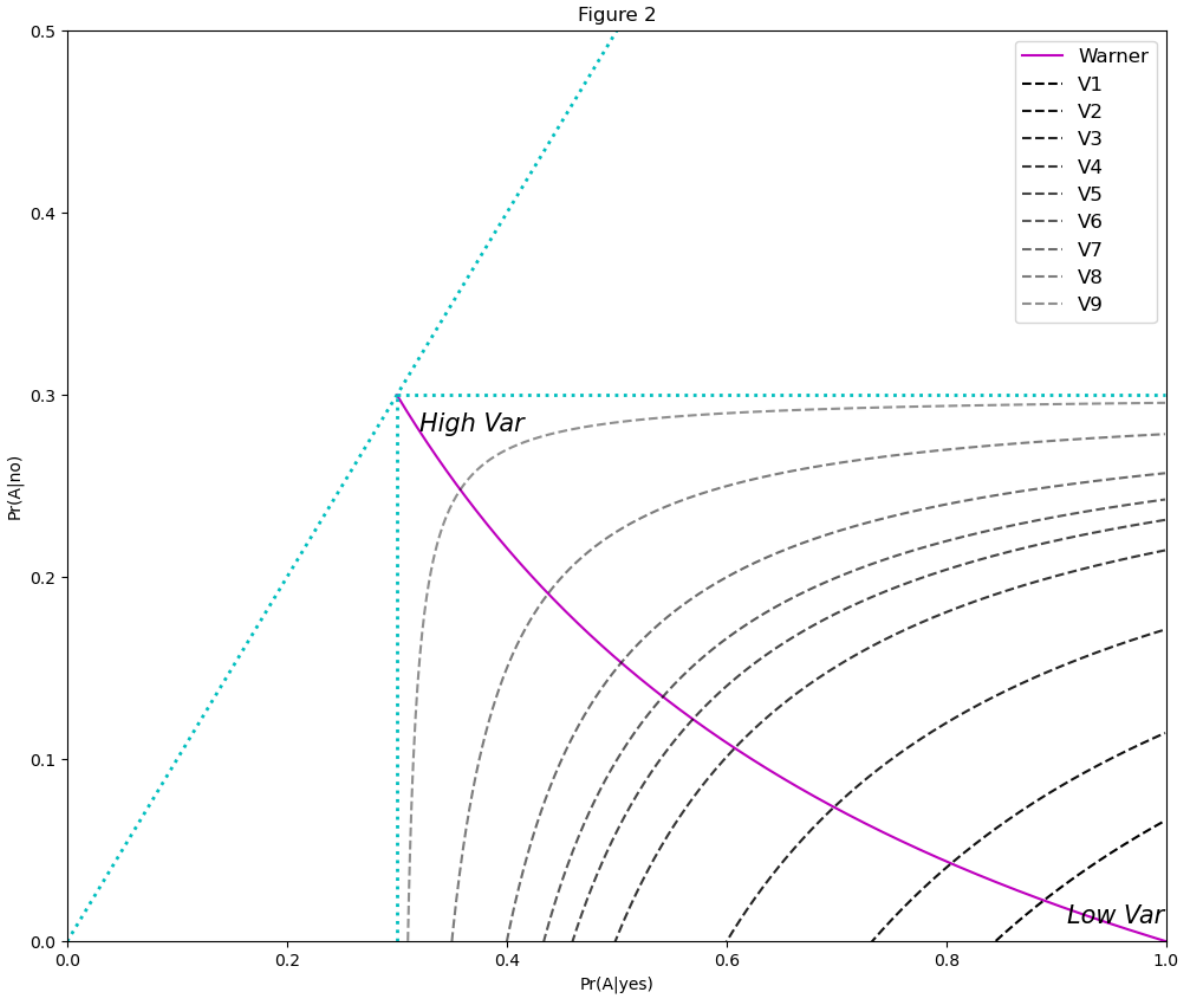
- All points on one iso-variance curve share the same variance
- From V_1 to V_9, the variance of the iso-variance curve increase monotonically, as colors brighten monotonically

Suppose the parameters of the iso-variance model follow those in Ljungqvist [Lju93], which are:

- $\pi = 0.3$
- $n = 100$

Then we can plot the iso-variance curve in Figure 2:

```
var = Iso_Variance(pi=0.3, n=100)
var.plotting_iso_variance_curve()
```



22.5.3 Optimal Survey

A point on an iso-variance curves can be attained with the unrelated question design.

We now focus on finding an "optimal survey design" that

- Minimizes the variance of the estimator subject to privacy restrictions.

To obtain an optimal design, we first superimpose all individuals' truth borders on the iso-variance mapping.

To construct an optimal design

- The statistician should find the intersection of areas above all truth borders; that is, the set of conditional probabilities ensuring truthful answers from all respondents.
- The point where this set touches the lowest possible iso-variance curve determines an optimal survey design.

Consquently, a minimum variance unbiased estimator is pinned down by an individual who is the least willing to volunteer a truthful answer.

Here are some comments about the model design:

- An individual's decision of whether or not to answer truthfully depends on his or her belief about other respondents' behavior, because this determines the individual's calculation of $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$.
- An equilibrium of the optimal design model is a Nash equilibrium of a noncooperative game.
- Assumption (22.12) is sufficient to guarantee existence of an optimal model design. By choosing $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$ sufficiently close to each other, all respondents will find it optimal to answer truthfully. The closer are these probabilities, the higher the variance of the estimator becomes.
- If respondents experience a large enough increase in expected utility from telling the truth, then there is no need to use a randomized response model. The smallest possible variance of the estimate is then obtained at $\Pr(A|\text{yes}) = 1$ and $\Pr(A|\text{no}) = 0$; that is, when respondents answer truthfully to direct questioning.
- A more general design problem would be to minimize some weighted sum of the estimator's variance and bias. It would be optimal to accept some lies from the most "reluctant" respondents.

22.6 Criticisms of Proposed Privacy Measures

We can use a utilitarian approach to analyze some privacy measures.

We'll enlist Python Code to help us.

22.6.1 Analysis of Method of Lanke's (1976)

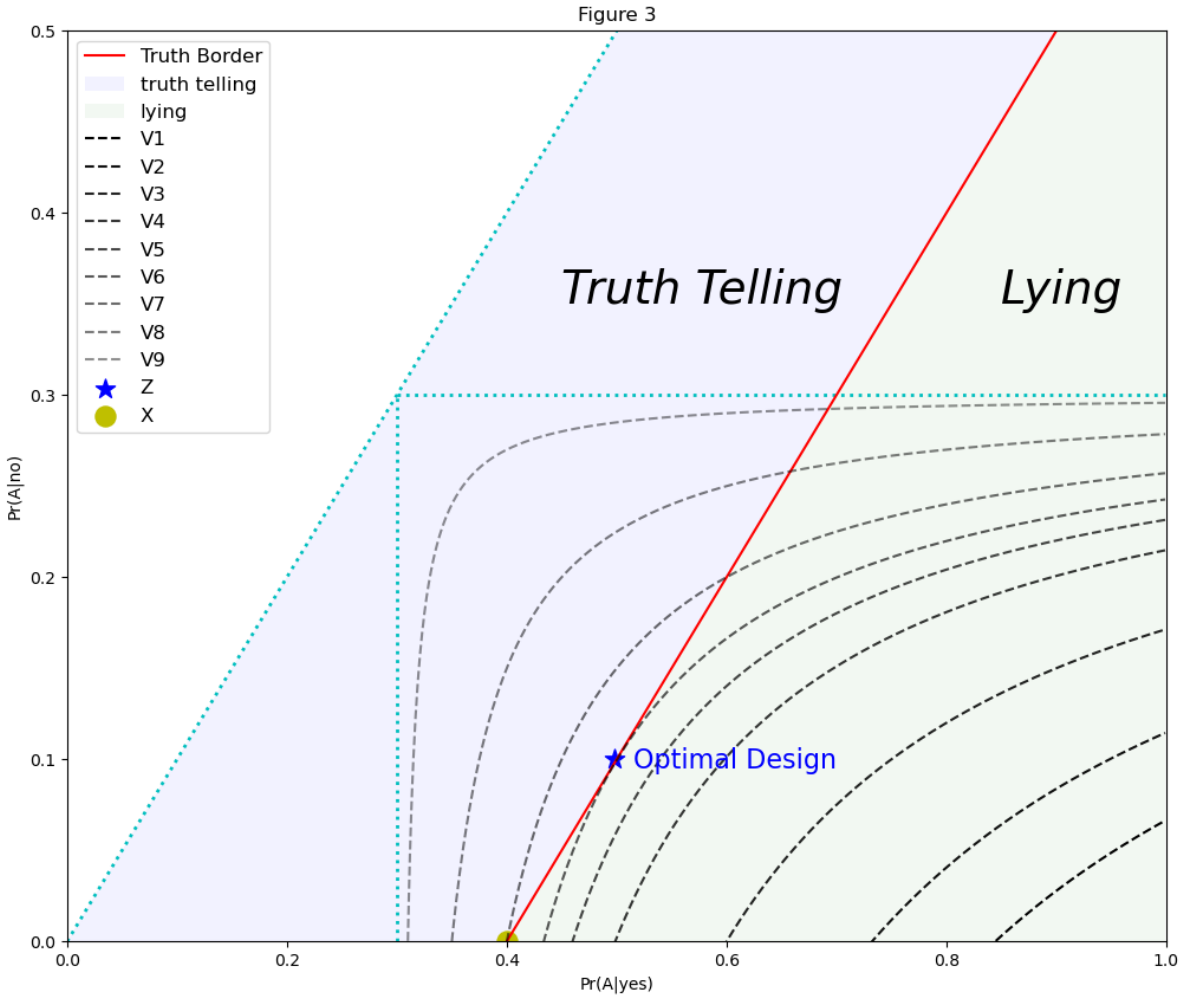
Lanke (1976) recommends a privacy protection criterion that minimizes:

$$\max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\} \tag{22.20}$$

Following Lanke's suggestion, the statistician should find the highest possible $\Pr(A|\text{yes})$ consistent with truth telling while $\Pr(A|\text{no})$ is fixed at 0. The variance is then minimized at point X in Figure 3.

However, we can see that in Figure 3, point Z offers a smaller variance that still allows cooperation of the respondents, and it is achievable following our discussion of the truth border in Part III:

```
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = x - 0.4
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1, 'c:', linewidth=2)
plt.plot(y2, x2, 'c:', linewidth=2)
plt.plot(x, y, 'r-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x, 0, y, facecolor='green', alpha=0.05, label='lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')


plt.scatter(0.498, 0.1, c='b', marker='*', label='Z', s=150)
plt.scatter(0.4, 0, c='y', label='X', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.85, 0.35, "Lying",fontdict = {'size':28, 'style':'italic'})
plt.text(0.515, 0.095, "Optimal Design", fontdict={'size':16,'color':'b'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 3')
plt.show()
```



22.6.2 Method of Leysieffer and Warner (1976)

Leysieffer and Warner (1976) recommend a two-dimensional measure of jeopardy that reduces to a single dimension when there is no jeopardy in a 'no' answer", which means that

$$\Pr(\text{yes}|A) = 1$$

and

$$\Pr(A|\text{no}) = 0$$

This is not an optimal choice under a utilitarian approach.

22.6.3 Analysis on the Method of Chaudhuri and Mukerjee's (1988)

[CM88]

Chaudhuri and Mukerjee (1988) argued that the individual may find that since "yes" may sometimes relate to the sensitive group A, a clever respondent may falsely but safely always be inclined to respond "no". In this situation, the truth border is such that individuals choose to lie whenever the truthful answer is "yes" and

$$\Pr(A|\text{no}) = 0$$

Here the gain from lying is too high for someone to volunteer a "yes" answer.

This means that

$$U_i\left(\Pr(A|\text{yes}), \text{truth}\right) < U_i\left(\Pr(A|\text{no}), \text{lie}\right)$$

in any situation always.

As a result, there is no attainable model design.

However, under a utilitarian approach there should exist other survey designs that are consistent with truthful answers.

In particular, respondents will choose to answer truthfully if the relative advantage from lying is eliminated.

We can use Python to show that the optimal model design corresponds to point Q in Figure 4:

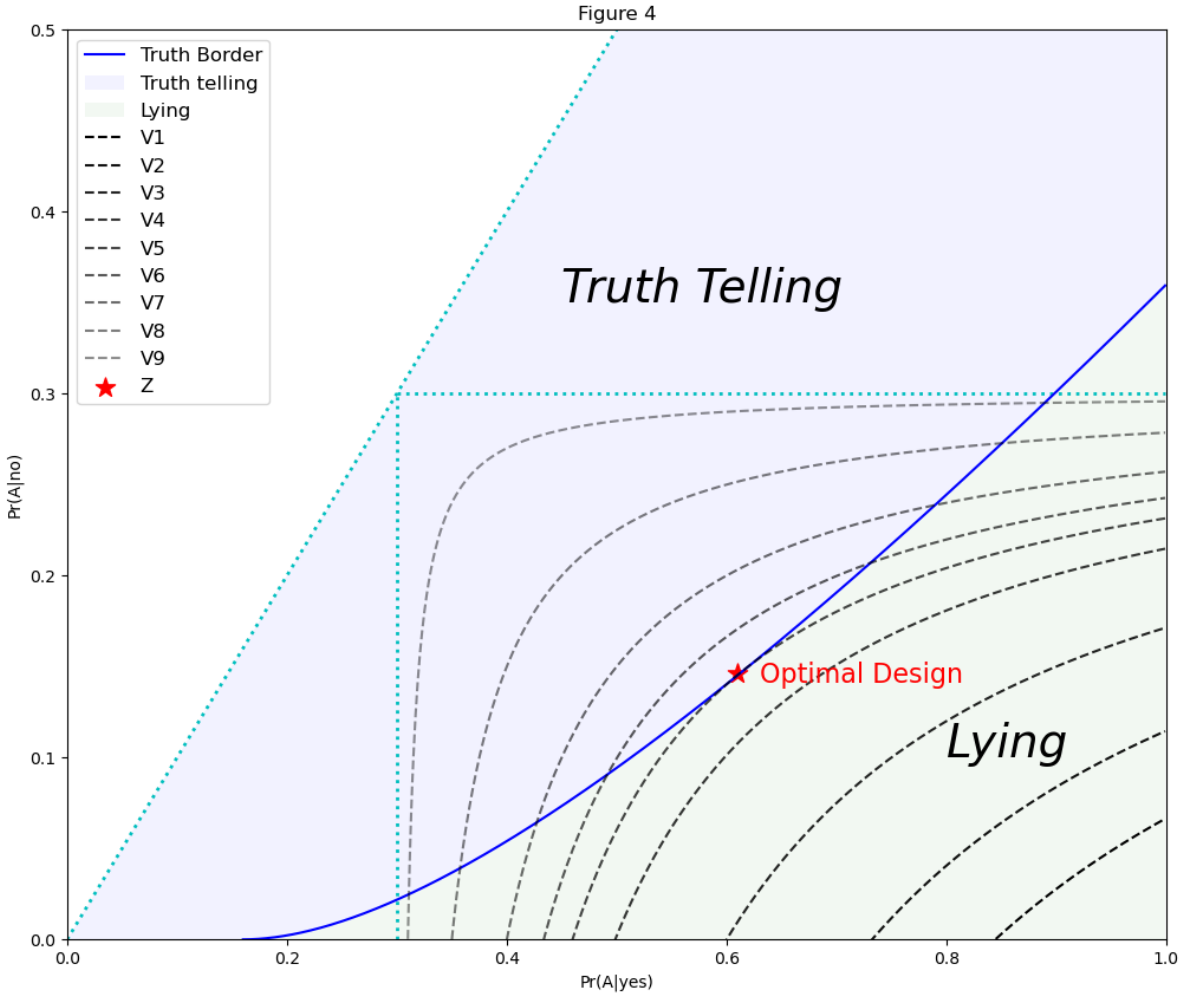
```
def f(x):
    if x < 0.16:
        return 0
    else:
        return (pow(x, 0.5) - 0.4)**2
```

```
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = [f(i) for i in x]
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]
x3 = np.arange(0.16, 1, 0.001)
y3 = (pow(x3, 0.5) - 0.4)**2

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1,'c:', linewidth=2)
plt.plot(y2, x2,'c:', linewidth=2)
plt.plot(x3, y3,'b-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='Truth telling')
plt.fill_between(x3, 0, y3,facecolor='green', alpha=0.05, label='Lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
plt.scatter(0.61, 0.146, c='r', marker='*', label='Z', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})
plt.text(0.63, 0.141, "Optimal Design", fontdict={'size':16,'color':'r'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 4')
plt.show()
```



22.6.4 Method of Greenberg et al. (1977)

[GKAH77]

Greenberg et al. (1977) defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{22.21}$$

The hazard for an individual who does not belong to A is

$$\Pr(\text{yes}|A') \times \Pr(A|\text{yes}) + \Pr(\text{no}|A') \times \Pr(A|\text{no}) \tag{22.22}$$

They also considered an alternative related measure of hazard that they said "is likely to be closer to the actual concern felt by a respondent."

Their "limited hazard" for an individual in A and $A^{'}$ is

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{22.23}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{22.24}$$

According to Greenberg et al. (1977), a respondent commits himself or herself to answer truthfully on the basis of a probability in (22.21) or (22.23) **before** randomly selecting the question to be answered.

Suppose that the appropriate privacy measure is captured by the notion of "limited hazard" in (22.23) and (22.24).

Consider an unrelated question model where the unrelated question is replaced by the instruction "Say the word 'no'", which implies that

$$\Pr(A|\text{yes}) = 1$$

and it follows that:

- Hazard for an individual in $A^{'}$ is 0.
- Hazard for an individual in A can also be made arbitrarily small by choosing a sufficiently small $\Pr(\text{yes}|A)$.

Even though this hazard can be set arbitrarily close to 0, an individual in A will completely reveal his or her identity whenever truthfully answering the sensitive question.

However, under utilitarian framework, it is obviously contradictory.

If the individuals are willing to volunteer this information, it seems that the randomized response design was not necessary in the first place.

It ignores the fact that respondents retain the option of lying until they have seen the question to be answered.

22.7 Concluding Remarks

The justifications for a randomized response procedure are that

- Respondents are thought to feel discomfort from being perceived as belonging to the sensitive group.
- Respondents prefer to answer questions truthfully than to lie, unless it is too revealing.

If a privacy measure is not completely consistent with the rational behavior of the respondents, all efforts to derive an optimal model design are futile.

A utilitarian approach provides a systematic way to model respondents' behavior under the assumption that they maximize their expected utilities.

In a utilitarian analysis:

- A truth border divides the space of conditional probabilities of being perceived as belonging to the sensitive group, $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$, into the truth-telling region and the lying region.
- The optimal model design is obtained at the point where the truth border touches the lowest possible iso-variance curve.

A practical implication of the analysis of [Lju93] is that uncertainty about respondents' demands for privacy can be acknowledged by **choosing $\Pr(A|\text{yes})$ and $\Pr(A|\text{no})$ sufficiently close to each other**.

Part IV

Other

CHAPTER

TWENTYTHREE

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

23.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

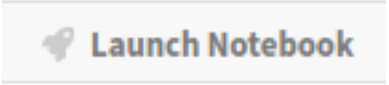
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use conda install -y quantecon on the command line, or
- execute !conda install -y quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

23.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

TWENTYFOUR

REFERENCES

CHAPTER

TWENTYFIVE

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
ar1_bayes	2024-01-30 05:23	cache	581.28	✅
ar1_turningpts	2024-01-30 05:23	cache	59.4	✅
back_prop	2024-01-30 05:26	cache	133.66	✅
bayes_nonconj	2024-01-30 06:36	cache	4229.27	✅
exchangeable	2024-01-30 06:36	cache	10.22	✅
hoist_failure	2024-01-30 06:38	cache	74.47	✅
imp_sample	2024-01-30 06:42	cache	277.69	✅
intro	2024-01-30 06:42	cache	1.32	✅
likelihood_bayes	2024-01-30 06:43	cache	48.94	✅
likelihood_ratio_process	2024-01-30 06:43	cache	10.69	✅
lln_clt	2024-01-30 06:44	cache	14.86	✅
mix_model	2024-01-30 06:44	cache	39.79	✅
mle	2024-01-30 06:44	cache	6.16	✅
multi_hyper	2024-01-30 06:45	cache	25.96	✅
multivariate_normal	2024-01-30 06:45	cache	5.75	✅
navy_captain	2024-01-30 06:46	cache	38.82	✅
ols	2024-01-30 06:46	cache	17.5	✅
pandas_panel	2024-01-30 06:46	cache	6.18	✅
prob_matrix	2024-01-30 06:46	cache	18.22	✅
prob_meaning	2024-01-30 06:47	cache	75.81	✅
rand_resp	2024-01-30 06:48	cache	3.24	✅
status	2024-01-30 06:42	cache	1.32	✅
troubleshooting	2024-01-30 06:42	cache	1.32	✅
util_rand_resp	2024-01-30 06:48	cache	3.52	✅
wald_friedman	2024-01-30 06:48	cache	20.58	✅
zreferences	2024-01-30 06:42	cache	1.32	✅

These lectures are built on `linux` instances through `github actions` and `amazon web services (aws)` to enable access to a `gpu`. These lectures are built on a p3.2xlarge that has access to `8 vcpu's`, a `V100 NVIDIA Tesla GPU`, and `61 Gb` of memory.

BIBLIOGRAPHY

[AJR01] Daron Acemoglu, Simon Johnson, and James A Robinson. The colonial origins of comparative development: an empirical investigation. *The American Economic Review*, 91(5):1369–1401, 2001.

[And76] Harald Anderson. Estimation of a proportion through randomized response. *International Statistical Review/Revue Internationale de Statistique*, pages 213–217, 1976.

[Apo90] George Apostolakis. The concept of probability in safety assessments of technological systems. *Science*, 250(4986):1359–1364, 1990.

[Ber75] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.

[Bur23] Jennifer Burns. *Milton Friedman: The Last Conservative by Jennifer Burns*. Farrar, Straus, and Giroux, New York, 2023.

[CM88] A Chadhuri and R Mukerjee. *Randomized Response: Theory and Technique*. Marcel Dekker, New York, 1988.

[dF37] Bruno de Finetti. La prevision: ses lois logiques, ses sources subjectives. *Annales de l'Institute Henri Poincare'*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), \it Studies in Subjective Probability, Wiley, New York, 1964.

[Dud02] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.

[ESAW18] Ashraf Ben El-Shanawany, Keith H Ardron, and Simon P Walker. Lognormal approximations of fault tree uncertainty distributions. *Risk Analysis*, 38(8):1576–1584, 2018.

[FPS77] Michael A Fligner, George E Policello, and Jagbir Singh. A comparison of two randomized response survey methods with consideration for the level of respondent protection. *Communications in Statistics-Theory and Methods*, 6(15):1511–1524, 1977.

[FF98] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[GAESH69] Bernard G Greenberg, Abdel-Latif A Abul-Ela, Walt R Simmons, and Daniel G Horvitz. The unrelated question randomized response model: theoretical framework. *Journal of the American Statistical Association*, 64(326):520–539, 1969.

[GKAH77] Bernard G Greenberg, Roy R Kuebler, James R Abernathy, and Daniel G Horvitz. Respondent hazards in the unrelated question randomized response model. *Journal of Statistical Planning and Inference*, 1(1):53–60, 1977.

[GS93] Moses A Greenfield and Thomas J Sargent. A probabilistic analysis of a catastrophic transuranic waste hoise accident at the wipp. Environmental Evaluation Group, Albuquerque, New Mexico, June 1993. URL: http://www.tomsargent.com/research/EEG-53.pdf.

[Hur50] Leonid Hurwicz. Least squares bias in time series. *Statistical inference in dynamic economic models*, 10:365–383, 1950.

[Kre88] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[Lan75] Jan Lanke. On the choice of the unrelated question in simmons' version of randomized response. *Journal of the American Statistical Association*, 70(349):80–83, 1975.

[Lan76] Jan Lanke. On the degree of protection in randomized interviews. *International Statistical Review/Revue Internationale de Statistique*, pages 197–203, 1976.

[LW76] Frederick W Leysieffer and Stanley L Warner. Respondent jeopardy and optimal designs in randomized response models. *Journal of the American Statistical Association*, 71(355):649–656, 1976.

[Lju93] Lars Ljungqvist. A unified approach to measures of privacy in randomized response models: a utilitarian perspective. *Journal of the American Statistical Association*, 88(421):97–103, 1993.

[McC70] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[NP33] J. Neyman and E. S Pearson. On the problem of the most efficient tests of statistical hypotheses. *Phil. Trans. R. Soc. Lond. A. 231 (694–706)*, pages 289–337, 1933.

[OW69] Guy H. Orcutt and Herbert S. Winokur. First order autoregression: inference, estimation, and prediction. *Econometrica*, 37(1):1–14, 1969.

[Tre16] Daniel Treisman. Russia's billionaires. *The American Economic Review*, 106(5):236–241, 2016.

[Wal47] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.

[Wal80] W Allen Wallis. The statistical research group, 1942–1945. *Journal of the American Statistical Association*, 75(370):320–330, 1980.

[War65] Stanley L Warner. Randomized response: a survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[Wec79] William E Wecker. Predicting the turning points of a time series. *Journal of business*, pages 35–50, 1979.

[Woo15] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

INDEX