Data Analytics for Sense-Making

Vik Gopal, Lim Tiong Wee

2025-09-23

Contents

Preface

IND5003 is a foundation course for students in a Master's program for working adults, at the National University of Singapore. The title of the course is

> Data Analytics for Sense-Making

The content of the course aims to bring students up to speed in a variety of analytic techniques that will be useful for them in later courses of the program. You can read some reviews of the course on NUSmods.

In designing this course, we tried to cover the techniques from a couple of different angles:

- From the viewpoint of the types of data encountered, e.g. time series, unstructured text data, image data, and so on.
- From the viewpoint of the types of questions that could be asked of the data, e.g. are we attempting to plan for unseen scenarios (simulation), are we interrogating the data for hidden structure (unsupervised learning), and so on.

The entire course uses the Python programming language. By the end of the course, the aim was for students to become familiar with Python for data analysis.

If you are a fellow instructor and you find something useful in this textbook, please do let me know at vik.gopal@nus.edu.sg. If you need more details about anything, do feel free to write as well.

So long, and thanks for reading!

Vik

https://blog.nus.edu.sg/stavg

1 Introduction to Python

1.1 Introduction

Python is a general-purpose programming language. It is a higher-level language than C, C++ and Java in the sense that a Python program does not have to be compiled before execution.

It was originally conceived back in the 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. The language is named after a BBC TV show (Guido's favorite program) "Monty Python's Flying Circus".

Python reached version 1.0 in January 1994. Python 2.0 was released on October 16, 2000. Python 3.0, which is backwards-*incompatible* with earlier versions, was released on 3 December 2008.

Python is a very flexible language; it is simple to learn yet is fast enough to be used in production. Over the past ten years, more and more comprehensive data science toolkits (e.g. scikit-learn, NTLK, tensorflow, keras) have been written in Python and are now the standard frameworks for those models.

Python is an open-source software. It is free to use and extend.

1.2 Installing Python and Jupyter Lab

To install Python, navigate to the official Python download page to obtain the appropriate installer for your operating system.

> **!** Important
>
> For our class, please ensure that you are using Python 3.10.12.

The next step is to create a virtual environment for this course. Virtual environments are specific to Python. They allow you to retain multiple versions of Python, and of packages, on the same computer. Go through the videos on Canvas relevant to your operating system to create a virtual environment and install Jupyter Lab on your machine.

Jupyter notebooks are great for interactive work with Python, but more advanced users may prefer a full-fledged IDE. If you are an advanced user, and are comfortable with an IDE of your own choice (e.g. Spyder or VSCode), feel free to continue using that to run the codes for this course.

> **!** Important
>
> Even if you are using Anaconda/Spyder/VSCode, you still need to create a virtual environment.

Jupyter notebooks consist of cells, which can be of three main types:

- code cells,
- output cells, and
- markdown cells.

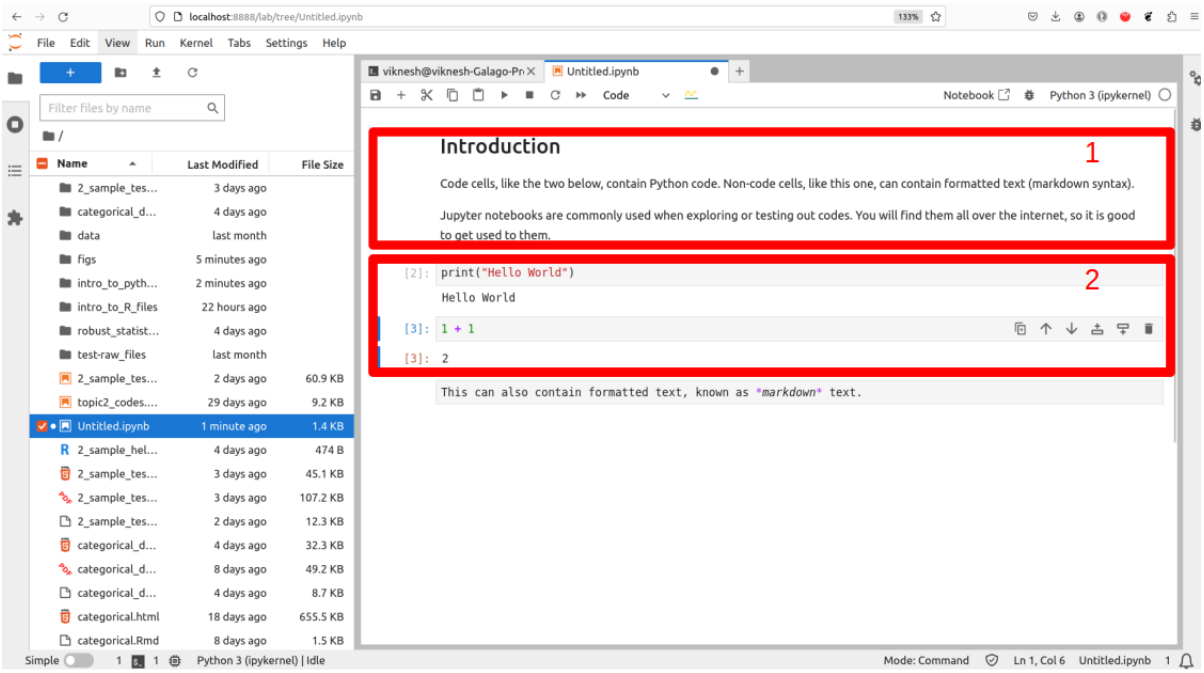


Figure 1.1: Jupyter Lab

In Figure 1.1, the red box labelled 1 is a markdown cell. It can be used to contain descriptions or summary of the code. The cells in the box labelled 2 are code cells. To run the codes from our notes, you can copy and paste the codes into a new cell, and then execute them with Ctrl-Enter.

Try out this Easter egg that comes with any Python installation:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

More information on using Jupyter notebooks can be obtained from this link.

1.3 Basic Data Structures in Python

The main objects in native[1] Python that contain data are

- **Lists**, which are defined with [ ]. Lists are mutable.
- **Tuples**, which are defined with ( ). Tuples are immutable.
- **Dictionaries**, which are defined with { }. Dictionaries have keys and items. They are also mutable.

Very soon, we shall see that for data analysis, the more common objects we shall deal with are dataframes (from pandas) and arrays (from numpy). However, the latter two require add-on packages; the three object classes listed above are baked into Python.

By the way, this is what mean by (im)mutable:

```
x = [1, 3, 5, 7, 8, 9, 10]

# The following is OK, because "x" is a list, and hence mutable
x[3] = 17
print(x)
```

```
[1, 3, 5, 17, 8, 9, 10]
```

```
# The following will return an error, because x_tuple is a tuple, and hence
# immutable.
x_tuple = (1, 3, 5, 6, 8, 9, 10)
x_tuple[3] = 17
```

Here is how we create lists, tuples and dictionaries.

```
x_list = [1, 2, 3]
x_tuple = (1, 2, 3)
x_dict = {'a': 1, 'b': 2, 'c': 3} # access with x_dict['a']
```

[1] i.e., Python without any packages imported.

1.4 Slice Operator in Python

One important point to take note is that, Python begins indexing of objects starting with 0. Second, indexing is aided by the slicing operator ':'. It is used in Python to extract regular sequences from a list, tuple or string easily.

In general, the syntax is `<list-like object>[a:b]`, where `a` and `b` are integers. Such a call would return the elements at indices `a`, `a+1` until `b-1`. Take note that the end point index is not included.

```
char_list = ['P', 'y', 't', 'h', 'o', 'n']
char_list[0]           # returns first element
char_list[-1]          # returns last element
len(char_list)         # returns number of elements in list-like object.
char_list[::2]         # from first to last, every 2 apart.
```

```
['P', 't', 'o']
```

This indexing syntax is used in the additional packages we use as well, so it is good to know about it. Figure 1.2 displays a pictorial representation of how positive and negative indexes work together.

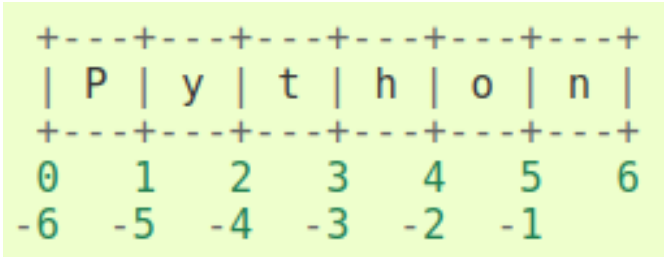


Figure 1.2: Positive and negative indices

1.5 Loops in Python

It is extremely efficient to execute "for" loops in Python. Many objects in Python are *iterators*, which means they can be iterated over. Lists, tuples and dictionaries can all be iterated over very easily.

Before getting down to examples, take note that Python does not use curly braces to denote code blocks. Instead, these are defined by the number of indentations in a line.

```
for i in x[:2]:
  print(f"The current element is {i}.")
```

```
The current element is 1.
The current element is 3.
```

Notice how we do not need to set up any running index; the object is just iterated over directly. The argument to the `print()` function is an f-string. It is the recommended way to create string literals that can vary according to arguments.

Here is another example of iteration, this time using dictionaries which have key-value pairs. In this case, we iterate over the keys.

```
dict1 = {'holmes': 'male', 'watson': 'male', 'mycroft': 'male',
         'hudson': 'female', 'moriarty': 'male', 'adler': 'female'}
# dict1['hudson']

for x in dict1.keys():
    print(f"The gender of {x} is {dict1[x]}")
```

```
The gender of holmes is male
The gender of watson is male
The gender of mycroft is male
The gender of hudson is female
The gender of moriarty is male
The gender of adler is female
```

1.6 Strings

Text data in Python is handled with `str` objects. A string object is an **immutable** sequence of characters. Here are some useful string methods and properties:

- catenating (joining) strings.
- finding sub-strings.
- iterating over strings.
- converting to lower/upper-case.

A few cells earlier, we saw how we can format a string before sending it to the `print()` function. Here's another such example. This is useful when we are debugging loops.

```
for x in range(1, 11):
    print(f"Sq:{x*x:3d} Cu:{x*x*x:4d}.")
```

```
Sq:  1 Cu:   1.
Sq:  4 Cu:   8.
Sq:  9 Cu:  27.
Sq: 16 Cu:  64.
Sq: 25 Cu: 125.
Sq: 36 Cu: 216.
Sq: 49 Cu: 343.
Sq: 64 Cu: 512.
Sq: 81 Cu: 729.
Sq:100 Cu:1000.
```

Here are more examples of working with strings.

```
test_str = "Where in the World is Carmen san Diego?"
```

Remember that a string is immutable, just like a tuple, so this assignment will *not* work:

```
test_str[5] = 'z'
```

But like a tuple, we can also iterate over a string.

```
count = 0
for x in test_str:
    if x.isupper():
        count += 1
print(f"There were {count} upper-case characters in the sentence.")
```

```
There were 4 upper-case characters in the sentence.
```

```
test_str.lower()
```

```
'where in the world is carmen san diego?'
```

To join strings, we can use the '+' operator, the `str.join()` method, or, if they are part of the same expression, we can just place them next to each other separated by whitespace.

```
x = "Where shall "
y = "we "
print("Where shall " "we " "go today?")

# also works - the '+' operator is overloaded to work with strings:
# "Where " + "shall" +" we go" + " today?"

# join using another character:
# ':'.join(["Where", "shall", "we", "go", "today?"])
```

```
Where shall we go today?
```

To find simple patterns, we may turn to the `find`, `replace`, `startswith` and `endswith` methods.

```
test_str.find('Carmen')
```

```
22
```

```
test_str.replace('Carmen', 'John')
```

```
'Where in the World is John san Diego?'
```

For more complicated search operations over strings, we use a special mini-language known as regular expressions. These are used in several other languages such as R and Perl, so it is worth knowing about them if you have time. See Section 1.11 for a good introduction.

1.7 Functions, Modules and Packages

Functions provide a way to package code that you can re-use several times. To define a function in Python, use the `def` keyword.

```
def test_function(x):
    print('You typed', x)

test_function('test')
```

```
You typed test
```

A Python module is a file containing Python definitions (of functions and constants) and statements. Instead of re-typing functions every time, we can simply load the module. We would then have access to the updated functions. We access objects within the module using the "dot" notation. There are several modules that ship with the default Python installation. Note that Python packages are collections of modules.

Here are a couple of ways of importing (and then using) constants from the math module.

```
import math

# compute e^2
math.exp(2)

# print pi
math.pi
```

```
3.141592653589793
```

Alternatively, we could import the constant π so that we do not need to use the dot notation.

```
from math import pi
pi
```

```
3.141592653589793
```

1.8 Object-Oriented Programming

Python has been developed as both a functional and object-oriented programming language. Much of the code we will soon use involves creation of an instance, and then accessing the attributes (data or methods) of that instance.

Consider the following class Circle.

```
class Circle:
    """ A simple class definition

    c0 = Circle()
    c0.radius

    """
    def __init__(self, radius = 1.0):
        self.radius = radius

    def area(self):
        """ Compute area of circle"""
        return pi*(self.radius**2)
```

Having defined the class, we can *instantiate* it and use the methods it contains.

```
c1 = Circle(3.2)
c2 = Circle(4.0)
c2.area()
```

```
50.26548245743669
```

In this simple class, we can also set the value of the attribute of an instance, although this is not the recommended way.

```
c1.radius = 6
c1.area()
```

```
113.09733552923255
```

1.9 Numpy

```
import numpy as np
import pandas as pd
from itables import show
```

The basic object in this package is the `ndarray` object, which can represent n-dimensional arrays of homogeneous data types. This is the key difference between NumPy and Pandas objects, which we shall encounter later on in this chapter. While Pandas objects are also tabular in nature, they allow you to deal with inhomogenous objects. Specifically, Pandas' `DataFrame`s allow columns to be of different types.

An `ndarray` object is an n-dimensional array (i.e., a tensor) of elements, indexed by a tuple of non-negative integers.

The dimensions of the array are referred to as **axes** in NumPy: a three-dimensional array will have three axes.

Each array has several attributes. These include:

- `ndim`: the number of axes/dimensions.
- `shape`: a tuple describing the length of each dimension.
- `size`: the total number of elements in the array. This is a product of the integers in the shape attribute.

```
arr = np.array([(1.5, 2, 3), (4, 5, 6)])

# number of axes
arr.ndim
```

```
2
```

```
# the length of each axis
arr.shape
```

```
(2, 3)
```

```
# number of elements in the array
arr.size
```

```
6
```

1.9.1 Array Creation

One way to create an array containing regular sequences is to use the `np.arange()` function. This creates a sequence of integers, with a specified separation.

```
seq = np.arange(0, 10, 3)
seq
```

```
array([0, 3, 6, 9])
```

The shape of an `ndarray` is given by a tuple. Note that an array of shape (4,) is different from one with shape (4, 1). The former has only **1 dimension**, while the latter has **2 dimensions**.

```
# seq.shape
col_vect = seq.reshape(4,1)
col_vect
```

```
array([[0],
       [3],
       [6],
       [9]])
```

To create an array of regularly spaced **real numbers**, use `np.linspace()`.

```
arr_real = np.linspace(start = 0.2, stop = 3.3, num = 24).reshape(2, 3, 4)
arr_real
```

```
array([[[0.2       , 0.33478261, 0.46956522, 0.60434783],
        [0.73913043, 0.87391304, 1.00869565, 1.14347826],
        [1.27826087, 1.41304348, 1.54782609, 1.6826087 ]],

       [[1.8173913 , 1.95217391, 2.08695652, 2.22173913],
        [2.35652174, 2.49130435, 2.62608696, 2.76086957],
        [2.89565217, 3.03043478, 3.16521739, 3.3       ]]])
```

Sometimes we need to create a placeholder array with the appropriate dimensions, and then fill it in later. This is preferrable to growing an array by appending to it.

```
np.zeros((3, 5)) # there is also an np.ones() function
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Instead of specifying the dimensions of an array ourselves, we can create arrays of zeros or ones in the shape of other existing arrays.

```
# Creates an array of zeros, of the same shape as "arr_real".
np.ones_like(arr_real)
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

1.9.2 Slice Operator in Multiple Dimensions

Multidimensional NumPy arrays can be accessed with comma separated slice notation. When fewer indices are provided than the number of axes, the missing indices are considered complete slices for the remaining dimensions.

By the way, when printing, the *last* axis will be printed left-to-right, and the second last axis will be printed from top-to-bottom. The remaining axes will be printed with a line in between:

```
arr_real
```

```
array([[[0.2       , 0.33478261, 0.46956522, 0.60434783],
        [0.73913043, 0.87391304, 1.00869565, 1.14347826],
        [1.27826087, 1.41304348, 1.54782609, 1.6826087 ]],

       [[1.8173913 , 1.95217391, 2.08695652, 2.22173913],
        [2.35652174, 2.49130435, 2.62608696, 2.76086957],
        [2.89565217, 3.03043478, 3.16521739, 3.3       ]]])
```

Here are examples based on this array. Try to guess what each will return before you run it:

```
arr_real[1,2,3]
arr_real[0, 2, ::-1]
arr_real[1, 0:3:2]
arr_real[:,2,:]
```

Here are examples using Boolean indexing, which means that we use an array of `True` and `False` entries to determine which elements to return.

```
arr_real > 3
```

```
array([[[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]],

       [[False, False, False, False],
        [False, False, False, False],
        [False,  True,  True,  True]]])
```

```
arr_real[arr_real > 3]
```

```
array([3.03043478, 3.16521739, 3.3       ])
```

1.9.3 Basic Operations

```
# Setting a seed allows for reproducibility of random number generation
# across sessions.
rng = np.random.default_rng(1361)
```

```
a = rng.uniform(size=(3,5))
b = rng.uniform(size=(3,5))
b
```

```
array([[0.47031767, 0.43520574, 0.74929584, 0.19794778, 0.91397998],
       [0.95979996, 0.10301973, 0.82972687, 0.61118187, 0.04727632],
       [0.51642555, 0.32188431, 0.57800994, 0.73144202, 0.74020866]])
```

```
# Element-wise addition.
a + b
```

```
array([[0.92582851, 0.90610288, 1.3360719 , 0.43528561, 1.30890054],
       [1.78346478, 0.31376731, 1.51889626, 1.40816571, 0.21158293],
       [0.52725269, 0.57843605, 1.07039529, 1.54221413, 0.82312206]])
```

```
# Element-wise multiplication: NOT matrix multiplication.
a * b
```

```
array([[0.2142348 , 0.20493714, 0.43966886, 0.0469805 , 0.36094949],
       [0.79055346, 0.02171116, 0.57182236, 0.48710208, 0.00776781],
       [0.00559141, 0.08257998, 0.28460362, 0.59303279, 0.06137322]])
```

```
# Matrix multiplication (need to transpose "b" to match get the right dimensions).
# We can also do "a @ b.T".
a.dot(b.T)
```

```
array([[1.26677078, 1.13630182, 1.19189672],
       [1.30342857, 1.87895687, 1.5961133 ],
       [0.721959  , 0.94481619, 1.02718103]])
```

1.9.4 Axis-wise Operations

```
arr_real.shape
```

```
(2, 3, 4)
```

```
arr_real.mean(axis = 0) # mean across the 0th ("first") axis
```

```
array([[1.00869565, 1.14347826, 1.27826087, 1.41304348],
       [1.54782609, 1.6826087 , 1.8173913 , 1.95217391],
       [2.08695652, 2.22173913, 2.35652174, 2.49130435]])
```

The top-left element comes from the average of `arr_real[0,0,0]` and `arr_real[1,0,0]`. Similarly, the element to the right of it comes from the average of `arr_real[0,0,1]` and `arr_real[1,0,1]`:

```
(arr_real[0,0,1] + arr_real[1,0,1]) / 2
```

```
1.143478260869565
```

```
arr_real.mean(axis = 1)
```

```
array([[0.73913043, 0.87391304, 1.00869565, 1.14347826],
       [2.35652174, 2.49130435, 2.62608696, 2.76086957]])
```

Note that `arr_real[0]` is a 2D array, with shape (3, 4). Suppose we wish to compute the row means. This means we have to apply the operation by the column axis (axis = 1).

```
# the mean across the second axis of arr_real[0], not of arr_real itself.
arr_real[0].mean(axis = 1)
```

```
array([0.40217391, 0.94130435, 1.48043478])
```

If we wanted to identify the row with the largest mean, we use `argmax()` on the resulting array.

```
arr_real[0].mean(axis=1).argmax()
```

```
2
```

Here is a table with some common operations that you can apply on a numpy array.

Method	Description
`shape`	Returns dimensions, e.g. `matrix1.shape`
`T`	Transposes the array, e.g. `matrix1.T`
`mean`	Computes col- or row-wise means, e.g. `matrix1.mean(axis=0)` or `matrix1.mean(axis=1)`
`sum`	Computes col- or row-wise means, e.g. `matrix1.sum(axis=0)` or `matrix1.sum(axis=1)`
`argmax`	Return the index corresponding to the max within the specified dimension, e.g. `matrix1.argmax(axis=0)` for the position with the max within each column.
`reshape`	To change the dimensions, e.g. `array1.reshape((5,1))` converts the array into a 5x1 matrix

1.10 Pandas

1.10.1 Series

A *Series* is a one-dimensional labeled array. The axis labels are referred to as the **index**. The simplest way to create a Series is to pass a sequence and an index to `pd.Series()`.

Example 1.1 (Creating Pandas Series)**.** Consider the following data, from the football league in Spain.

```
year = pd.Series(list(range(2010, 2013) ) * 3)

team = ["Barcelona", "RealMadrid", "Valencia"] * 3
team.sort()
team = pd.Series(team)

wins = pd.Series([30, 28, 32, 29, 32, 26, 21, 17, 19])
draws = pd.Series([6, 7, 4, 5, 4, 7, 8, 10, 8])
losses = pd.Series([2, 3, 2, 4, 2, 5, 9, 11, 11])

#wins.index
#wins.values
```

To access particular values, we can use the slice operator.

```
wins[0:6:2]
```

```
0    30
2    32
4    32
dtype: int64
```

To convert a Series object to an `ndarray`, we use the following method:

```
wins.to_numpy()
```

```
array([30, 28, 32, 29, 32, 26, 21, 17, 19])
```

If we specify an index, we can use it to access values in the Series.

```
s = pd.Series(rng.uniform(size=5),
              index=['a', 'b', 'c', 'd', 'e'])
# s
# s.index
# s.values
```

```
s[['a', 'c']]
```

```
a    0.974173
c    0.948907
dtype: float64
```

Be careful when you combine the slice operator with label-based indexing. Unlike vanilla Python, Pandas includes **both** end-points!

```
s['a':'d']
```

```
a    0.974173
b    0.270230
c    0.948907
d    0.675365
dtype: float64
```

1.10.2 DataFrames

A *DataFrame* is a 2-dimensional labeled data structure with possibly **different** data types. It is the most commonly used Pandas object. The *index* of a DataFrame refers to the row labels (axis 0). The *columns* refer to the column labels (axis 1).

DataFrames can be constructed from Series, dictionaries, lists and 2-d arrays. For our course, we will typically create a DataFrame directly from a file.

Example 1.2 (Creating Pandas DataFrame from Series)**.** We can create a DataFrame from the earlier series.

```
laliga = pd.DataFrame({'Year': year,
               'Team': team,
               'Wins': wins,
               'Draws': draws,
               'Losses': losses
})
```

To inspect a DataFrame, we can use `info()`, `head()` and `tail()` methods.

```
laliga.head()
```

	Year	Team	Wins	Draws	Losses
0	2010	Barcelona	30	6	2
1	2011	Barcelona	28	7	3
2	2012	Barcelona	32	4	2
3	2010	RealMadrid	29	5	4
4	2011	RealMadrid	32	4	2

1.10.3 Reading in Data

Pandas can read in data stored in multiple formats, including CSV, tab-separated files, Excel files and HDF5 files.

Example 1.3 (Happiness Dataset)**.** The CSV file read in here contains the happiness scores of 164 countries from 2015 to 2017. Click here for a full report on the dataset. The final score was based on many other factors (such as GDP per capita, family, freedom etc) which is included in the file as well. We will simplify things by just reading in the country, final score computed and year.

In each year, not all of the 164 countries had their scores surveyed and taken. This results in some countries having missing values (`NaN`) in certain years.

```
happ = pd.read_csv('data/happiness_report.csv', header=0,
                   na_values='NA')
```

```
happ.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 492 entries, 0 to 491
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Country            492 non-null    object
 1   Happiness.Rank     471 non-null    float64
 2   Happiness.Score    471 non-null    float64
 3   GDP                471 non-null    float64
 4   Family             471 non-null    float64
 5   Life.Expectancy    471 non-null    float64
 6   Freedom            471 non-null    float64
 7   Govt.Corruption    471 non-null    float64
 8   Generosity         471 non-null    float64
 9   Dystopia.Residual  471 non-null    float64
 10  Year               492 non-null    int64
dtypes: float64(9), int64(1), object(1)
memory usage: 42.4+ KB
```

1.10.4 Basic Selection

In dataframes, row selection can be done with integers in with the slice operator. In practice, this is not used, because we typically wish to select a set of rows based on a condition.

```
print(happ[10:12])
```

```
       Country  Happiness.Rank  Happiness.Score      GDP   Family  \
10      Israel            11.0            7.278  1.22857  1.22393
11  Costa Rica            12.0            7.226  0.95578  1.23788

    Life.Expectancy  Freedom  Govt.Corruption  Generosity  Dystopia.Residual  \
10          0.91387  0.41319          0.07785     0.33172            3.08854
11          0.86027  0.63376          0.10583     0.25497            3.17728

    Year
```

```
10  2015
11  2015
```

To select columns, you may use a list of column names.

```
happ[['GDP', 'Freedom']] # note the difference with happ['GDP']
# happ.GDP.head()
```

	GDP	Freedom
0	1.39651	0.66557
1	1.30232	0.62877
2	1.32548	0.64938
3	1.45900	0.66973
4	1.32629	0.63297
...	...	...
487	NaN	NaN
488	NaN	NaN
489	NaN	NaN
490	NaN	NaN
491	NaN	NaN

Remember that we are not working with numpy arrays, so this will *not* work:

```
happ[0:10, 2:4]
```

1.10.5 Indexing and Selecting Data

The two main methods of advanced data selection use the `.loc` and `.iloc` functions. Although we call them functions, they are summoned using the `[ ]` notation. The `.loc` is primarily label-based. The common allowed inputs to `.loc` are

- a single label,
- a list of labels,
- a slice object,
- a boolean array.

The `.iloc` is primarily an integer-based input. The common allowed inputs to `.iloc` are

- a single integer,
- a list of integers,
- a slice object,
- a boolean array.

When selecting from a DataFrame with `.loc` or `.iloc`, we can provide a comma-separated index, just as with NumPy. It is good to keep this reference page bookmarked.

Take note that this next command will only work if the index is made up of integers!

```
print(happ.loc[2:5])

# happ.loc[[2,3,4,5]]
```

```
   Country  Happiness.Rank  Happiness.Score      GDP   Family  \
2  Denmark             3.0            7.527  1.32548  1.36058
3   Norway             4.0            7.522  1.45900  1.33095
4   Canada             5.0            7.427  1.32629  1.32261
5  Finland             6.0            7.406  1.29025  1.31826

   Life.Expectancy  Freedom  Govt.Corruption  Generosity  Dystopia.Residual  \
2          0.87464  0.64938          0.48357     0.34139            2.49204
3          0.88521  0.66973          0.36503     0.34699            2.46531
4          0.90563  0.63297          0.32957     0.45811            2.45176
5          0.88911  0.64169          0.41372     0.23351            2.61955

   Year
2  2015
3  2015
4  2015
5  2015
```

Notice below how the slice operator is inclusive when we use `.loc`, but not inclusive when we use `.iloc`.

```
happ.loc[2:10:4, "GDP":"Generosity":2]
```

	GDP	Life.Expectancy	Govt.Corruption
2	1.32548	0.87464	0.48357
6	1.32944	0.89284	0.31814
10	1.22857	0.91387	0.07785

```
happ.iloc[2:11:4, 3:8:2] # Same as above, but with .iloc
```

	GDP	Life.Expectancy	Govt.Corruption
2	1.32548	0.87464	0.48357
6	1.32944	0.89284	0.31814
10	1.22857	0.91387	0.07785

1.10.6 Filtering Data

Suppose we are interested in the very happy countries. Here is how we can filter the data with a boolean array.

Example 1.4 (Happiest Countries)**.** Suppose we retrieve the happiest countries; those with a score more than 6.95. Can you surmise why we use this value?

```
happiest = happ[happ['Happiness.Score'] > 6.95]

happiest.Country.unique()
```

```
array(['Switzerland', 'Iceland', 'Denmark', 'Norway', 'Canada', 'Finland',
       'Netherlands', 'Sweden', 'New Zealand', 'Australia', 'Israel',
       'Costa Rica', 'Austria', 'Mexico', 'United States', 'Brazil',
       'Ireland', 'Germany'], dtype=object)
```

Notice that there isn't a single Asian or African country in the happiest 10% of countries!

When filtering, we can also combine Boolean indices.

```
# Top 3 happiest countries in 2015
print(happ[(happ.Year == 2015) & (happ['Happiness.Rank'] <= 3)])
```

```
       Country  Happiness.Rank  Happiness.Score      GDP   Family  \
0  Switzerland             1.0            7.587  1.39651  1.34951
1      Iceland             2.0            7.561  1.30232  1.40223
2      Denmark             3.0            7.527  1.32548  1.36058

   Life.Expectancy  Freedom  Govt.Corruption  Generosity  Dystopia.Residual  \
0          0.94143  0.66557          0.41978     0.29678            2.51738
1          0.94784  0.62877          0.14145     0.43630            2.70201
2          0.87464  0.64938          0.48357     0.34139            2.49204

   Year
0  2015
1  2015
2  2015
```

1.10.7 Missing Values

The `.info()` method will yield information on missing values, column by column. We can see there are 21 rows with missing values.

```
happ.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 492 entries, 0 to 491
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Country            492 non-null    object
 1   Happiness.Rank     471 non-null    float64
```

```
 2   Happiness.Score    471 non-null    float64
 3   GDP                471 non-null    float64
 4   Family             471 non-null    float64
 5   Life.Expectancy    471 non-null    float64
 6   Freedom            471 non-null    float64
 7   Govt.Corruption    471 non-null    float64
 8   Generosity         471 non-null    float64
 9   Dystopia.Residual  471 non-null    float64
 10  Year               492 non-null    int64
dtypes: float64(9), int64(1), object(1)
memory usage: 42.4+ KB
```

Sometimes, it is appropriate to drop rows with missing values. This can be done with the `.dropna` method. Remember that it returns a new dataframe. The original one remains unchanged, unless you include the `inplace=True` argument.

```
new_df = happ.dropna()

# pd.isna(happ)
```

1.11 References

In this chapter, we have introduced the following data science tools:

- Python programming language
- Jupyter notebooks for working with Python
- Computer set-up for data science with Python

Python is very widely used for data science, and especially for the machine learning aspect of it. For those of you with intentions to take up the GC in Deep Learning or Data Mining, it will be critical to be familiar with the language. It will be used again in at least DSA5102.

1. Regular expression HOWTO A tutorial with examples on regular expressions (for manipulating and searching through strings).
2. Formatting string literals: Or just f-strings
3. Python documentation: This is the official Python documentation page. It contains a tutorial, detailed description of the usual libraries and HOWTOs for many specific tasks. Most sections contain working examples that you can learn from, or modify to suit your task. It is good to bookmark it.
4. Obtaining a github copilot license For proof of student status, access EduRec -> Academics -> Academic Records -> View Student Status Letter, then take a photo of the pdf. Your application status should be shown in the same GitHub page. It will take a few days for the copilot access to be granted after your application is approved.

2 Statistical Inference

2.1 Introduction

In statistics, we often wish to make inference about a *population*, using a *sample*. The sample typically has uncertainty associated with it, because the precise values will differ each time we draw a sample from the population. The process of utilising the observations from the sample to make conclusions regarding population characteristics, is known as **statistical inference**. This topic introduces two main techniques for statistical inference in common application contexts. The two techniques are (1) hypothesis tests and (2) confidence intervals.

2.1.1 Hypothesis Tests

You might have been introduced to hypothesis tests in an introductory course before, but just to get us all on the same page, here is the general approach for conducting a hypothesis test:

Step 1: Assumptions

In this step, we make a note of the assumptions required for the test to be valid. In some tests, this step is carried out last, but, it is nonetheless essential to perform as it could invalidate the test. Some tests are very sensitive to the assumptions - this is in fact the main reason that the class of *robust statistics* was invented.

Step 2: State the hypotheses and significance level

The purpose of hypothesis testing is to make an inferential statement about the population from which the data arose. This inferential statement is what we refer to as the hypothesis regarding the population.

The hypotheses will be stated as a pair: The first hypothesis is the null hypothesis H_0 and the second is the alternative hypothesis H_1. Both statements will involve the **population parameter** (not the data summary) of interest. For example, if we have a sample of observations from two groups A and B, and we wish to assess if the mean of the populations is different, the hypotheses would be

$$H_0: \quad \mu_A = \mu_B \tag{2.1}$$
$$H_1: \quad \mu_A \neq \mu_B \tag{2.2}$$

H_0 is usually a statement that indicates "no difference", and H_1 is the complement of H_0, or a subset of it.

At this stage, it is also crucial to state the significance level of the test. The significance level corresponds to the Type I error of the test - the probability of rejecting H_0 when in fact it was true. This level is usually denoted as α, and is usually taken to be 5%, but there is no reason to adopt this blindly. Where possible, the significance level should be chosen to be appropriate for the problem at hand.

> Think of the choice of 5% as corresponding to accepting an error at a rate of 1 in 20 - that's how it was originally decided upon by R A Fisher.

Step 3: Compute the test statistic

The test statistic is usually a measure of how far the observed data deviates from the scenario defined by H_0. Usually, the larger it is, the more evidence we have against H_0. The construction of a hypothesis test (by theoretical statisticians) involves the derivation of the exact or approximate distribution of the test statistic under H_0. Deviations from the assumption could render this distribution incorrect.

Step 4: Compute the p-value

The p-value quantifies the chance of occurrence of a dataset as extreme as the one observed, under the assumptions of H_0. The distribution of the test statistic under H_0 is used to compute this value between 0 and 1. A value closer to 0 indicates stronger evidence against H_0.

Step 5: State your conclusion

This is the binary decision stage - either we reject H_0, or we do not reject H_0. It is conventional to use this terminology (instead of "accepting H_1") since our p-value is a measure of evidence *against* H_0 (not *for* it).

2.1.2 Confidence Intervals

Confidence intervals are an alternative method of inference for population parameters. Instead of yielding a binary reject/do-not-reject result, they return an
interval that contains the plausible values for the population parameter. Many confidence intervals are derived by inverting hypothesis tests, and almost all confidence intervals are of the form

> Sample estimate $\pm$ margin of error

For instance, if we observe $x_1, \ldots, x_n$ from a Normal distribution, and wish to estimate the mean of the distribution, the 95% confidence interval based on the the t distribution is

$$\bar{x} \pm t_{0.025,n-1} \times \frac{s}{\sqrt{n}}$$

where

- $\bar{x}$ is the sample mean,
- s is the sample standard deviation, and

- $t_{0.025,n-1}$ is the 0.025-quantile from the t distribution with $n-1$ degrees of freedom.

The formulas for many confidence intervals rely on asymptotic Normality of the estimator. However, this is an assumption that can be overcome with the technique of bootstrapping. If time permits, we shall touch on this in a later topic of our course. Bootstrapping can also be used to sidestep the distributional assumptions in hypothesis tests, but I still much prefer confidence intervals to tests because they yield an interval; they provide much more information than a binary outcome.

2.2 Comparing Means

2.2.1 2-sample Tests

In an independent samples t-test, observations in one group yield *no information* about the observations in the other group. Independent samples can arise in a few ways:

- In an experimental study, study units could be assigned randomly to different treatments, thus forming the two groups.
- In an observational study, we could draw a random sample from the population, and then record an explanatory categorical variable on each unit, such as the gender or senior-citizen status.
- In an observational study, we could draw a random sample from a group (say smokers), and then a random sample from another group (say non-smokers). This would result in a situation where the independent 2-sample t-test is appropriate.

Formal Set-up

Formally speaking, this is how the independent 2-sample t-test works:

Suppose that $X_1, X_2, \ldots, X_{n_1}$ are independent observations from group 1, and $Y_1, \ldots Y_{n_2}$ are independent observations from group 2. It is assumed that

$$X_i \sim N(\mu_1, \sigma^2),\ i = 1, \ldots, n_1 \tag{2.3}$$
$$Y_j \sim N(\mu_2, \sigma^2),\ j = 1, \ldots, n_2 \tag{2.4}$$

The null and alternative hypotheses would be

$$H_0 : \ \mu_1 = \mu_2 \tag{2.5}$$
$$H_1 : \ \mu_1 \neq \mu_2 \tag{2.6}$$

The test statistic for this test is:

$$T_1 = \frac{(\bar{X} - \bar{Y}) - 0}{s_p\sqrt{1/n_1 + 1/n_2}}$$

where

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$$

Notice that the numerator of T_1 will be large (in absolute value) when the difference in sample group means is also large. This is what we mean when we say that the test-statistic measures deviation from the null hypothesis.

Under H_0, the test statistic T_1 follows a t-distribution with $n_1 + n_2 - 2$ degrees of freedom. When we use a software to apply the test above, it will typically also return a confidence interval, computed as

$$(\bar{X} - \bar{Y}) \pm t_{n_1+n_2-2,1-\alpha/2} \times s_p \sqrt{1/n_1 + 1/n_2}$$

For more details on the test, refer to the links in the references section Section 2.7.

```
import pandas as pd
import numpy as np

from scipy import stats

import statsmodels.api as sm
from statsmodels.formula.api import ols
import statsmodels.stats.multicomp as mc

import seaborn as sns
import matplotlib.pyplot as plt
from itables import show
```

Example 2.1 (Example: Abalone Measurements)**.** The dataset on abalone measurements from the UCI machine learning repository contains measurements of physical characteristics, along with the gender status. We derive a sample of 50 measurements of male and female abalone records for use here. Our goal is to study if there is a significant difference between the viscera weight between males and females. The derived dataset can be found on Canvas.

```
abl = pd.read_csv("data/abalone_sub.csv")
show(abl)
```

```
x = abl.viscera[abl.gender == "F"]
y = abl.viscera[abl.gender == "M"]

t_out = stats.ttest_ind(y, x)
ci_95 = t_out.confidence_interval()
```

```
print(f"The $p$-value for the test is {t_out.pvalue:.3f}.")#
print(f"The actual value of the test statistic is {t_out.statistic:.3f}.")
print(f"The upper and lower limits of the CI are ({ci_95[0]:.3f}, {ci_95[1]:.3f}).")
```

```
The $p$-value for the test is 0.365.
The actual value of the test statistic is 0.910.
The upper and lower limits of the CI are (-0.023, 0.063).
```

To assess the normality assumption, we make histograms and qq-plots. Histograms of data from a Normal distribution should appear symmetric and bell-shaped. The tails on both sides should come down at a moderate pace. If we observe asymmetry in our histogram, we might suspect deviation from Normality. As we can see from the figure below, a histogram with a long tail on the right (left) is referred to as right-skewed (corr. left-skewed).

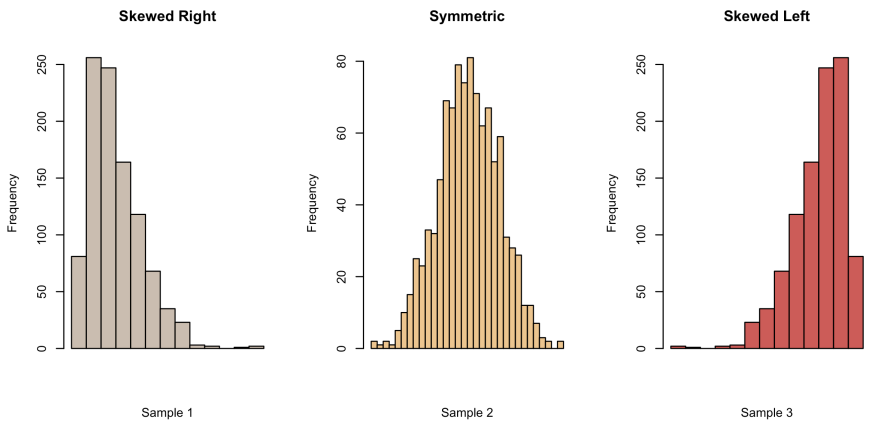
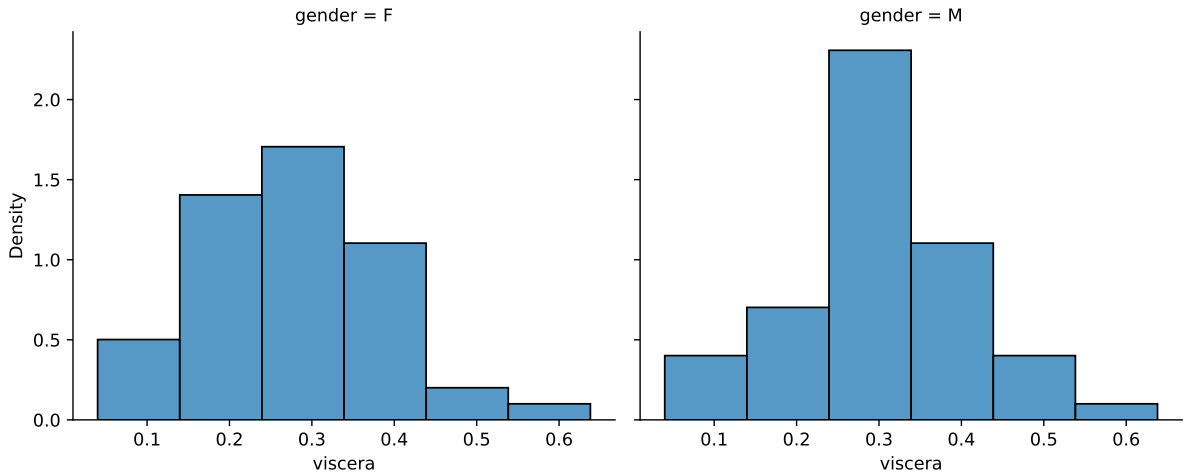


Figure 2.1: Sample Histograms

Example 2.2 (Example: Abalone Measurements)**.** Now we turn back to the histograms for the abalone data. Indeed, they do indicate some deviation from Normality - the female group is a little skewed to the right, while the male group appears to have a sharper peak than a Normal.

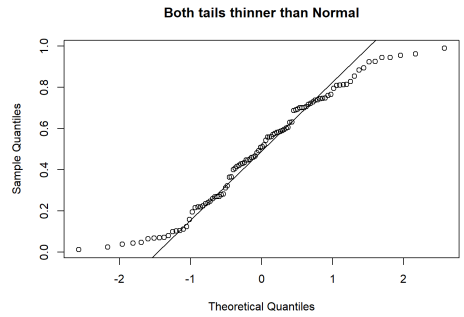
```
sns.displot(abl, x='viscera', col='gender', kind='hist', stat='density',
            binwidth=0.1, height=4, aspect=1.2);
```



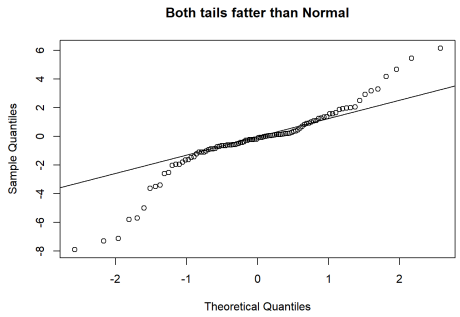
A Quantile-Quantile plot is a graphical diagnostic tool for assessing if a dataset follows a particular distribution. Most of the time we would be interested in comparing against a Normal distribution.

A QQ-plot plots the standardized sample quantiles against the theoretical quantiles of a N(0; 1) distribution. If they fall on a straight line, then we would say that there is evidence that

the data came from a normal distribution. Especially for unimodal datasets, the points in the middle will fall close to the line. The value of a QQ-plot is in judging if the tails of the data are fatter or thinner than the tails of the Normal.



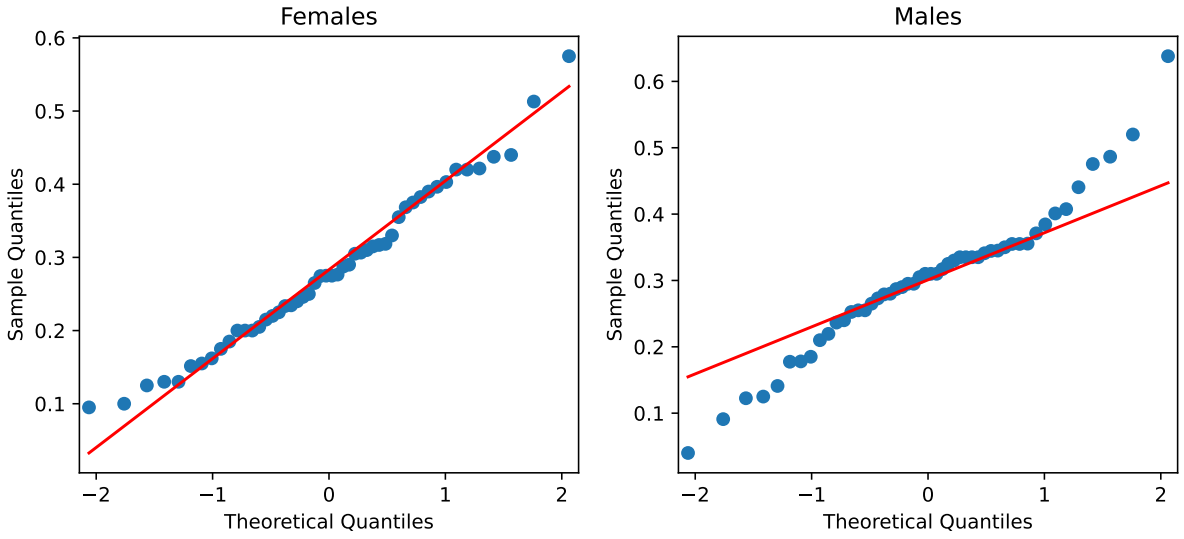
(a) Thinner



(a) Fatter

Example 2.3 (Example: Abalone Measurements)**.** If we compare the qq-plots from the data (below) with the reference (above), we can infer that for females, the left tail is thinner than a Normal - it abruptly cuts off. For males, both the left and the right tail are fatter than a Normal's.

```
f, axs = plt.subplots(1, 2, figsize=(10,4))
tmp = plt.subplot(121)
sm.qqplot(x, line="q", ax=tmp)
tmp.set_title('Females')
tmp = plt.subplot(122)
sm.qqplot(y, line="q", ax=tmp)
tmp.set_title('Males');
```



If you are keen on learning about particular hypothesis tests for Normality, take a look at the references Section 2.7.

We also need to assess if the variances are equal. While there are many hypothesis tests specifically for assessing if variances are equal (e.g. Levene, Bartlett), in our class, I advocate a

simple rule of thumb. If the larger s.d is more than twice the smaller one, than we should not use the equal variance form of the test. This rule of thumb is widely used in practice (see the references Section 2.7).

```
abl.groupby('gender').describe()
```

gender	viscera count	mean	std	min	25%	50%	75%	max
F	50.0	0.28241	0.108707	0.095	0.201250	0.275	0.365125	0.575
M	50.0	0.30220	0.108746	0.040	0.253125	0.310	0.348750	0.638

We would conclude that there is no significant difference between the mean viscera weight of males and females.

2.3 Paired Sample Tests

The data in a paired sample test also arises from two groups, but the two groups are not independent. A very common scenario that gives rise to this test is when the same subject receives both treatments. His/her measurement under each treatment gives rise to a measurement in each group. However, the measurements are no longer independent.

Example 2.4 (Example: Reaction time of drivers)**.** Consider a study on 32 drivers sampled from a driving school. Each driver is put in a simulation of a driving situation, where a target flashes red and green at random periods. Whenever the driver sees red, he/she has to press a brake button.

For each driver, the study is carried out twice - at one of the repetitions, the individual carries on a phone conversation while at the other, the driver listens to the radio. Each measurement falls under one of two groups - "phone" or "radio", but the measurements for driver i are clearly related.

Some people might just have a slower/faster baseline reaction time!

This is a situation where a paired sample test is appropriate, not an independent sample test.

2.3.1 Formal Set-up

Suppose that we observe $X_1, \ldots, X_n$ independent observations from group 1 and $Y_1, \ldots, Y_n$ independent observations from group 2. However the pair (X_i, Y_i) are correlated. Similar to the previous section, it is assumed that

$$X_i \sim N(\mu_1, \sigma_1^2),\ i = 1, \ldots, n \tag{2.7}$$
$$Y_j \sim N(\mu_2, \sigma_2^2),\ j = 1, \ldots, n \tag{2.8}$$

We let $D_i = X_i - Y_i$ for $i = 1, \ldots, n$. It follows that

$$D_i \sim N(\mu_1 - \mu_2,\ \sigma_1^2 + \sigma_2^2 - 2cov(X_i, Y_i))$$

The null and alternative hypotheses are stated in terms of the distribution of D_i:

$$\begin{aligned} H_0 &: \ \mu_D = 0 \\ H_1 &: \ \mu_D \neq 0 \end{aligned}$$

The test statistic for this test is:

$$T_2 = \frac{\bar{D} - 0}{s/\sqrt{n}}$$

where

$$s^2 = \frac{\sum_{i=1}^{n}(D_i - \bar{D})^2}{(n-1)}$$

Under H_0, the test statistic $T_2 \sim t_{n-1}$. When we use a software to apply the test above, it will typically also return a confidence interval, computed as

$$\bar{D} \pm t_{n-1,1-\alpha/2} \times s/\sqrt{n}$$

Example 2.5 (Example: Heart Rate Before/After Treadmill)**.** The following dataset comes from a textbook. In a self-recorded experiment, an individual recorded his heart rate before using a treadmill (baseline) and 5 minutes after use, for 12 days in 2006.

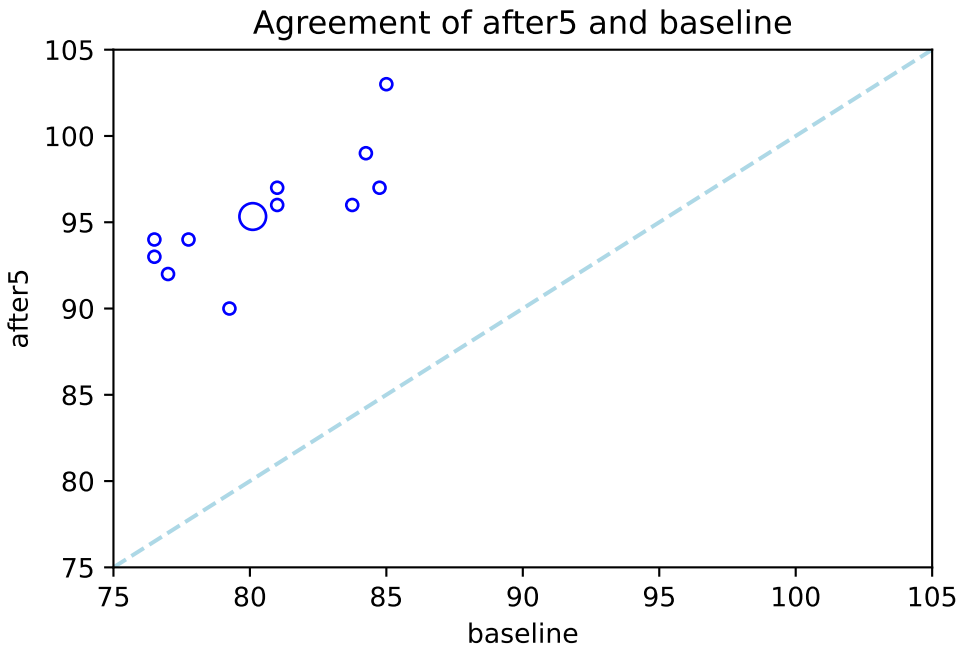
```
hr_df = pd.read_csv("data/health_promo_hr.csv")
#hr_df.head()
p_test_out = stats.ttest_rel(hr_df.baseline, hr_df.after5)

print(f"The $p$-value for the test is {p_test_out.pvalue:.2g}.")
print(f"""
The difference in means is {hr_df.baseline.mean() - hr_df.after5.mean():.3f}.
      """)
```

```
The $p$-value for the test is 2.2e-10.

The difference in means is -15.229.
```

While we do not include them here, it is imperative to also make the checks for Normality. If you were to make them, you would realise that the sample size is rather small. Even so, it is difficult to make the case for Normality here.

Here is a plot that is particularly useful in paired sample studies:

```
ax1 = hr_df.plot(x='baseline', y='after5',kind='scatter', marker='o',
                 edgecolor='blue', color='none')
group_means = hr_df.loc[:, ['baseline', 'after5']].mean(axis=0)
ax1.set_xlim(75, 105)
ax1.set_ylim(75,105)
ax1.plot([75,105], [75,105], color="lightblue", linestyle="dashed");
ax1.scatter(group_means.iloc[0], group_means.iloc[1],  marker='o',
            edgecolor='blue', color='none', s=100);
ax1.set_title('Agreement of after5 and baseline');
```



2.4 ANoVA

In this section, we introduce the *one-way analysis of variance* (ANOVA), which generalises the t-test methodology to more than 2 groups. Hypothesis tests in the ANOVA framework require the assumption of Normality.

While the F-test in ANOVA provides a determination of whether or not the group means are different, in practice, we would always want to follow up with specific comparisons between groups as well.

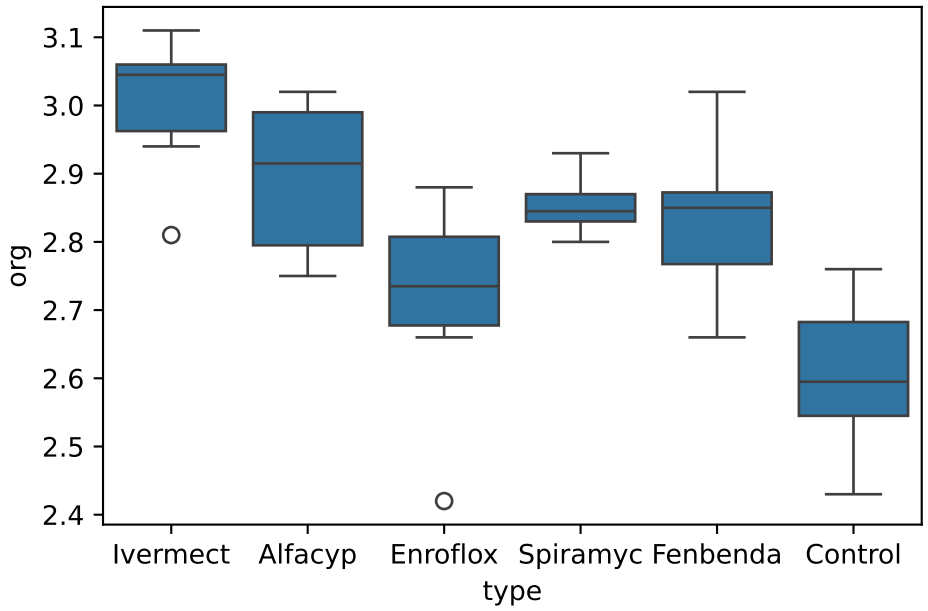
Example 2.6 (Example: Heifers)**.** The following example was taken from Introduction to Statistical Data Analysis for Life Sciences.

An experiment with dung from heifers was carried out in order to explore the influence of antibiotics on the decomposition of dung organic material. As part of the experiment, 36 heifers were randomly assigned into six groups. Note that a heifer is a young, female cow that has not had her first calf yet.

Antibiotics of different types were added to the feed for heifers in five of the groups. The remaining group served as a control group. For each heifer, a bag of dung was dug into the soil, and after 8 weeks the amount of organic material was measured for each bag.

Here is a boxplot of the data from each group, along with summary statistics in a table below.

```
heifers = pd.read_csv('data/antibio.csv')
sns.boxplot(heifers, x='type', y='org',);
```



Compared to the control group, it does appear that the median organic weight of the dung from the other heifer groups is higher. The following table displays the mean, standard deviation, and count from each group:

```
heifers.groupby('type').describe()
```

	org							
	count	mean	std	min	25%	50%	75%	max
type								
Alfacyp	6.0	2.895000	0.116748	2.75	2.7950	2.915	2.9900	3.02
Control	6.0	2.603333	0.118771	2.43	2.5450	2.595	2.6825	2.76
Enroflox	6.0	2.710000	0.161988	2.42	2.6775	2.735	2.8075	2.88
Fenbenda	6.0	2.833333	0.123558	2.66	2.7675	2.850	2.8725	3.02
Ivermect	6.0	3.001667	0.109438	2.81	2.9625	3.045	3.0600	3.11
Spiramyc	4.0	2.855000	0.054467	2.80	2.8300	2.845	2.8700	2.93

Observe that the Spiramycin group only yielded 4 readings instead of 6. Our goal in this topic is to understand a technique for assessing if group means are statistically different from one another. Here are the specific analyses that we shall carry out:

Heifers: Questions of Interest

1. Is there any significant difference, at 5% level, between the mean decomposition level of the groups?
2. At 5% level, is the mean level for Enrofloxacin different from the control group?
3. Pharmacologically speaking, Ivermectin and Fenbendazole are similar to each other. Let us call this sub-group (A). They work differently than Enrofloxacin. At 5% level, is there a significant difference between the mean from sub-group A and Enrofloxacin?

2.4.1 Formal Set-up

Suppose there are k groups with n_i observations in the i-th group. The j-th observation in the i-th group will be denoted by Y_{ij}. In the One-Way ANOVA, we assume the following model:

$$Y_{ij} = \mu + \alpha_i + e_{ij},\ i = 1, \dots, k,\ j = 1, \dots, n_i \tag{2.9}$$

- μ is a constant, representing the underlying mean of all groups taken together.
- α_i is a constant specific to the i-th group. It represents the difference between the mean of the i-th group and the overall mean.
- e_{ij} represents random error about the mean $\mu + \alpha_i$ for an individual observation from the i-th group.

In terms of distributions, we assume that the e_{ij} are i.i.d from a Normal distribution with mean 0 and variance σ^2. This leads to the model for each observation:

$$Y_{ij} \sim N(\mu + \alpha_i,\ \sigma^2)$$

It is not possible to estimate both μ and all the k different α_i's, since we only have k observed mean values for the k groups. For identifiability purposes, we need to constrain the parameters. There are two common constraints used, and note that different software have different defaults:

1. Setting $\sum_{i=1}^{k} \alpha_i = 0$, or
2. Setting $\alpha_1 = 0$.

Continuing on from the equation for Y_{ij}, let us denote the mean for the i-th group as $\overline{Y_i}$, and the overall mean of all observations as $\overline{\overline{Y}}$. We can then write the deviation of an individual observation from the overall mean as:

$$Y_{ij} - \overline{\overline{Y}} = \underbrace{(Y_{ij} - \overline{Y_i})}_{\text{within}} + \underbrace{(\overline{Y_i} - \overline{\overline{Y}})}_{\text{between}}$$

The first term on the right of the above equation is an indication of *within-group variability*. The second term on the right is an indication of *between-group variability*. The intuition behind the ANOVA procedure is that if the between-group variability is large and the within-group variability is small, then we have evidence that the group means are different.

If we square both sides of the above equation and sum over all observations, we arrive at the following equation; the essence of ANOVA:

$$\sum_{i=1}^{k}\sum_{j=1}^{n_i}\left(Y_{ij}-\overline{\overline{Y}}\right)^2=\sum_{i=1}^{k}\sum_{j=1}^{n_i}\left(Y_{ij}-\overline{Y_i}\right)^2+\sum_{i=1}^{k}\sum_{j=1}^{n_i}\left(\overline{Y_i}-\overline{\overline{Y}}\right)^2$$

The squared sums above are referred to as:

$$SS_T = SS_W + SS_B$$

- SS_T: Sum of Squares Total,
- SS_W: Sum of Squares Within, and
- SS_B: Sum of Squares Between.

In addition the following definitions are important for understanding the ANOVA output:

1. The Between Mean Square:
$$MS_B = \frac{SS_B}{k-1}$$
2. The Within Mean Square:
$$MS_W = \frac{SS_W}{n-k}$$

The mean squares are estimates of the variability between and within groups. The ratio of these quantities is the test statistic.

2.4.2 F-Test in One-Way ANOVA

The null and alternative hypotheses are:

$$\begin{aligned} H_0 &: \alpha_i = 0 \text{ for all } i \\ H_1 &: \alpha_i \neq 0 \text{ for at least one } i \end{aligned}$$

The test statistic is given by

$$F = \frac{MS_B}{MS_W}$$

Under H_0, the statistic F follows an F distribution with $k-1$ and $n-k$ degrees of freedom.

2.4.3 Assumptions

These are the assumptions that will need to be validated.

1. The observations are independent of each other. This is usually a characteristic of the design of the experiment, and is not something we can always check from the data.
2. The errors are Normally distributed. Residuals can be calculated as follows:
$$Y_{ij} - \overline{Y_i}$$
The distribution of these residuals should be checked for Normality.

3. The variance within each group is the same. In ANOVA, the MS_W is a pooled estimate (across the groups) that is used; in order for this to be valid, the variance within each group should be identical. As in the 2-sample situation, we shall avoid separate hypotheses tests and proceed with the rule-of-thumb that if the ratio of the largest to smallest standard deviation is less than 2, we can proceed with the analysis.

Example 2.7 (Example: Heifers (Cont'd))**.** Let us now apply the F-test to the heifers dataset.

```
heifer_lm = ols('org ~ type', data=heifers).fit()
anova_tab = sm.stats.anova_lm(heifer_lm, type=3,)
anova_tab
```

	df	sum_sq	mean_sq	F	PR(>F)
type	5.0	0.590824	0.118165	7.972558	0.00009
Residual	28.0	0.415000	0.014821	NaN	NaN

At the 5% significance level, we reject the null hypothesis to conclude that the group means are significantly different from one another. This answers the first question we set out to.

To extract the estimated parameters, we can use the following code:

```
heifer_lm.summary()
```

Dep. Variable:	org	**R-squared:**	0.587
Model:	OLS	**Adj. R-squared:**	0.514
Method:	Least Squares	**F-statistic:**	7.973
Date:	Tue, 23 Sep 2025	**Prob (F-statistic):**	8.95e-05
Time:	09:06:05	**Log-Likelihood:**	26.655
No. Observations:	34	**AIC:**	-41.31
Df Residuals:	28	**BIC:**	-32.15
Df Model:	5		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	2.8950	0.050	58.248	0.000	2.793	2.997
type[T.Control]	-0.2917	0.070	-4.150	0.000	-0.436	-0.148
type[T.Enroflox]	-0.1850	0.070	-2.632	0.014	-0.329	-0.041
type[T.Fenbenda]	-0.0617	0.070	-0.877	0.388	-0.206	0.082
type[T.Ivermect]	0.1067	0.070	1.518	0.140	-0.037	0.251
type[T.Spiramyc]	-0.0400	0.079	-0.509	0.615	-0.201	0.121

Omnibus:	2.172	**Durbin-Watson:**	2.146
Prob(Omnibus):	0.338	**Jarque-Bera (JB):**	1.704
Skew:	-0.545	**Prob(JB):**	0.427
Kurtosis:	2.876	**Cond. No.**	6.71

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

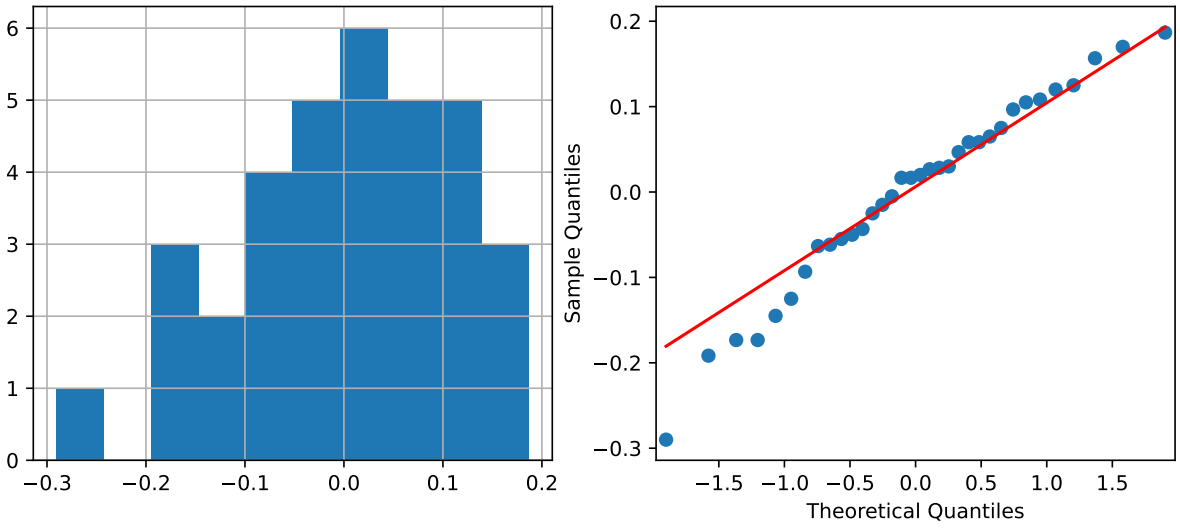
When estimating, Python sets one of the α_i to be equal to 0. We can tell from the output that the constraint has been placed on the coefficient for `Alfacyp` (since it is missing).

From the output, we can read off (the Intercept term) that the estimate for `Alfacyp` is precisely

$$2.895 + 0 = 2.895$$

To check the assumptions, we can use the following code:

```
f, axs = plt.subplots(1, 2, figsize=(10,4))
tmp = plt.subplot(121)
heifer_lm.resid.hist();
tmp = plt.subplot(122)
sm.qqplot(heifer_lm.resid, line="q", ax=tmp);
```



2.4.4 Comparing specific groups

The F-test in a One-Way ANOVA indicates if all means are equal, but does not provide further insight into which particular groups differ. If we had specified beforehand that we wished to test if two particular groups i_1 and i_2 had different means, we could do so with a t-test. Here are the details to compute a confidence interval in this case:

1. Compute the estimate of the difference between the two means:

$$\overline{Y_{i_1}} - \overline{Y_{i_2}}$$

2. Compute the standard error of the above estimator:

$$\sqrt{MS_W \left(\frac{1}{n_{i_1}} + \frac{1}{n_{i_2}} \right)}$$

3. Compute the $100(1-\alpha)$ confidence interval as:

$$\overline{Y_{i_1}} - \overline{Y_{i_2}} \pm t_{n-k,\alpha/2} \times \sqrt{MS_W \left(\frac{1}{n_{i_1}} + \frac{1}{n_{i_2}} \right)}$$

> **Note**
>
> If you notice from the summary statistics output for each group, the rule-of-thumb regarding standard deviations has not been satisfied. The ratio of largest to smallest standard deviations is slightly more than 2. Hence *we should not continue with ANOVA*; the pooled estimate of the variance may not be valid.
> However, we shall proceed with this dataset just to demonstrate the next few techniques, instead of introducing a new dataset.

Example 2.8 (Example: Enrofloxacin vs. Control)**.** Let us attempt to answer question (2), that we had set out earlier.

```
est1  = heifer_lm.params.iloc[2] - heifer_lm.params.iloc[1]
MSW = heifer_lm.mse_resid
df = heifer_lm.df_resid
q1 = -stats.t.ppf(0.025, df)

lower_ci = est1 - q1*np.sqrt(MSW * (1/6 + 1/4))
upper_ci = est1 + q1*np.sqrt(MSW * (1/6 + 1/4))
print("The 95% CI for the diff. between Enrofloxacin and control is" +
      f"({lower_ci:.3f}, {upper_ci:.3f}).")
```

```
The 95% CI for the diff. between Enrofloxacin and control is(-0.054, 0.268).
```

As the confidence interval contains the value 0, the binary conclusion would be to not reject the null hypothesis at the 5% level.

2.4.5 Contrast Estimation

A more general comparison, such as the comparison of a collection of l_1 groups with another collection of l_2 groups, is also possible. First, note that a linear contrast is any linear combination of the individual group means such that the linear coefficients add up to 0. In other words, consider L such that

$$L = \sum_{i=1}^{k} c_i \overline{Y_i}, \text{ where } \sum_{i=1}^{k} c_i = 0$$

Note that the comparison of two groups in Section 2.4.4 is a special case of this linear contrast.

Here is the procedure for computing confidence intervals for a linear contrast:

1. Compute the estimate of the contrast:

$$L = \sum_{i=1}^{k} c_i \overline{Y_i}$$

2. Compute the standard error of the above estimator:

$$\sqrt{MS_W \sum_{i=1}^{k} \frac{c_i^2}{n_i}}$$

3. Compute the $100(1-\alpha)$ confidence interval as:

$$L \pm t_{n-k,\alpha/2} \times \sqrt{MS_W \sum_{i=1}^{k} \frac{c_i^2}{n_i}}$$

Example 2.9 (Example: Enrofloxacin vs. Control)**.** As we mentioned earliers, let sub-group 1 consist of Ivermectin and Fenbendazole. Here is how we can compute a confidence interval for the difference between this sub-group, and Enrofloxacin.

```
c1 = np.array([-1, 0.5, 0.5])
n_vals = np.array([6, 6, 6,])
L = np.sum(c1 * heifer_lm.params.iloc[2:5])

MSW = heifer_lm.mse_resid
df = heifer_lm.df_resid
q1 = -stats.t.ppf(0.025, df)
se1 = np.sqrt(MSW*np.sum(c1**2 / n_vals))

lower_ci = L - q1*se1
upper_ci = L + q1*se1
print("The 95% CI for the diff. between the two groups is " +
      f"({lower_ci:.3f}, {upper_ci:.3f}).")
```

```
The 95% CI for the diff. between the two groups is (0.083, 0.332).
```

2.4.6 Multiple Comparisons

The procedures in the previous two subsections correspond to contrasts that we had specified before collecting or studying the data. If, instead, we wished to perform particular comparisons after studying the group means, or if we wish to compute all pairwise contrasts, then we need to adjust for the fact that we are conducting multiple tests. If we do not do so, the chance of making at least one false positive increases greatly.

Bonferroni

The simplest method for correcting for multiple comparisons is to use the Bonferroni correction. Suppose we wish to perform m pairwise comparisons, either as a test or by computing confidence intervals. If we wish to maintain the significance level of each test at α, then we should perform each of the m tests/confidence intervals at α/m.

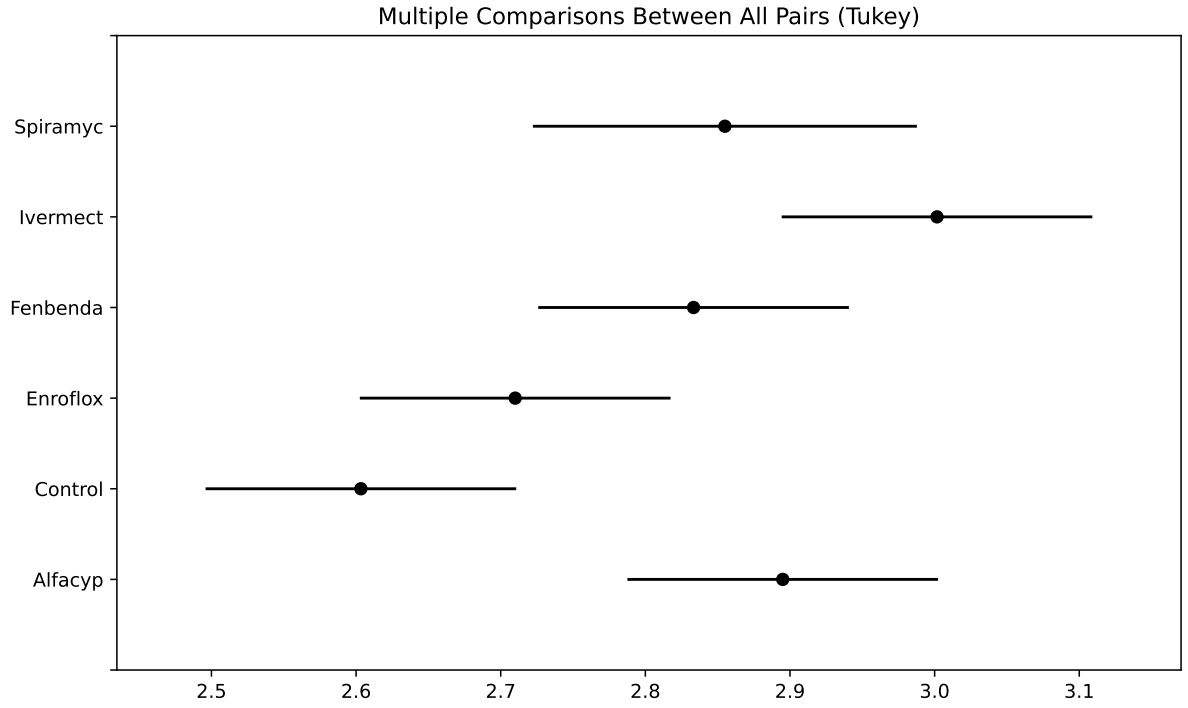
TukeyHSD

This procedure is known as Tukey's Honestly Significant Difference. It is designed to construct confidence intervals for **all** pairwise comparisons. For the same α-level, Tukey's HSD method provides shorter confidence intervals than a Bonferroni correction for all pairwise comparisons.

Example 2.10 (Example: Multiple Comparisons)**.** Let us apply Tukey's procedure to the heifers dataset.

```
cp = mc.MultiComparison(heifers.org, heifers.type)
tk = cp.tukeyhsd()
#print(tk)
```

```
tk.plot_simultaneous();
```



2.5 Categorical Variables

There are two sub-types of categorical variables:

- A categorical variable is *ordinal* if the observations can be ordered, but do not have specific quantitative values.
- A categorical variable is *nominal* if the observations can be classified into categories, but the categories have no specific ordering.

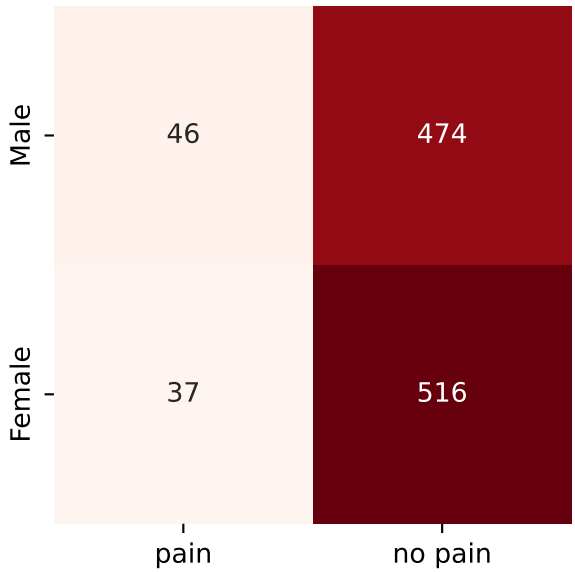
In this topic, we shall discuss techniques for identifying the presence, and for measuring the strength, of the association between two categorical variables.

Example 2.11 (Example: Chest Pain and Gender)**.** Suppose that 1073 NUH patients who were at high risk for cardiovascular disease (CVD) were randomly sampled. They were then queried on two things:

1. Had they experienced the onset of severe chest pain in the preceding 6 months? (yes/no)
2. What was their gender? (male/female)

The data would probably be summarised and presented in this format, which is known as a *contingency table*.

```
chest_array = np.array([[46, 474], [37, 516]])
#print(chest_array)

sns.heatmap(chest_array, annot=True, square=True, fmt='',
            xticklabels=['pain', 'no pain'],
            yticklabels=['Male', 'Female'],
            cmap='Reds', cbar=False, );
```



In the above table, the rows correspond to gender (Male, followed by Female) and the columns correspond to chest pain (from left to right: presence, then absence). Hence 516 of the 1073 patients were females who did not experience chest pain.

In a contingency table, each observation from the dataset falls in exactly one of the cells. The sum of all entries in the cells equals the number of independent observations in the dataset. All the techniques we shall touch upon in this chapter are applicable to contingency tables.

2.5.1 χ^2-Test for Independence

In the contingency table above, the two categorical variables are *Gender* and *Presence/absence of Pain*. With contingency tables, the main inferential task usually relates to assessing the association between the two categorical variables.

> **Note**
>
> If two categorical variables are **independent**, then the joint distribution of the variables would be equal to the product of the marginals. If two variables are not independent, we say that they are **associated**.

The χ^2-test uses the definition above to assess if two variables in a contingency table are associated. The null and alternative hypotheses are

$$\begin{aligned} H_0 &: \text{The two variables are indepdendent.} \\ H_1 &: \text{The two variables are not indepdendent.} \end{aligned}$$

Under the null hypothesis, we can estimate the joint distribution from the observed marginal counts. Based on this estimated joint distribution, we then compute *expected* counts (which may not be integers) for each cell. The test statistic essentially compares the deviation of *observed* cell counts from the expected cell counts.

Example 2.12 (Example: Chest Pain and Gender Expected Counts)**.** Continuing from the chest pain example, we can compute the estimated marginals using row and column proportions

```
col_prop,row_prop = sm.stats.Table(chest_array).marginal_probabilities

print(f"The proportion of males in the sample was {col_prop[0]:.3f}.")
print("The proportion of all patients who experienced chest pain " +
      f"was {row_prop[0]:.3f}.")
```

```
The proportion of males in the sample was 0.485.
The proportion of all patients who experienced chest pain was 0.077.
```

If we let X represent gender and Y represent chest pain, then we can estimate that:

$$\begin{aligned} \widehat{P}(X = \text{male}) &= 0.485 \\ \widehat{P}(Y = \text{pain}) &= 0.077 \end{aligned}$$

Consequently, *under* H_0 (independence), we would estimate

$$\widehat{P}(X = \text{male}, Y = \text{pain}) = 0.485 \times 0.077 \approx 0.04$$

From a sample of size 1073, the expected count for this cell is then

$$0.04 \times 1073 = 42.92$$

Using the approach above, we can derive a general formula for the expected count in each cell:

$$\text{Expected count} = \frac{\text{Row total} \times \text{Column total}}{\text{Total sample size}}$$

The formula for the χ^2-test statistic (with continuity correction) is:

$$\chi^2 = \sum \frac{|\text{expected} - \text{observed}|^2}{\text{expected count}}$$

The sum is taken over every cell in the table. Hence in a 2×2 table, as we have here, there would be 4 terms in the summation.

Example 2.13 (Example: Chest Pain and Gender χ^2 Test)**.** Let us see how we can apply and interpret the χ^2-test for the data in the chest pain example.

```
chisq_output = stats.chi2_contingency(chest_array, correction=False)

print(f"The p-value is {chisq_output.pvalue:.3f}.")
print(f"The test-statistic value is {chisq_output.statistic:.3f}.")
```

```
The p-value is 0.187.
The test-statistic value is 1.744.
```

Since the p-value is 0.1817, we would not reject the null hypothesis at significance level 5%. We do not have sufficient evidence to conclude that the variables are not independent.

To extract the expected cell counts, we can use the following code:

```
chisq_output.expected_freq
```

```
array([[ 40.22367195, 479.77632805],
       [ 42.77632805, 510.22367195]])
```

The test statistic in the χ^2-test compares the above table to the *observed* table.

> **Note**
>
> It is only suitable to use the χ^2-test when all *expected cell counts* are larger than 5. If this condition fails, one recommendation is to use a continuity correction, which modifies the test statistic to be
>
> $$\chi^2 = \sum \frac{(|\text{expected} - \text{observed}| - 0.5)^2}{\text{expected count}}$$
>
> However, I advocate using Fisher's exact test, which is also a test of independence, but measures deviations under slightly different assumptions. The test uses the exact distribution of the test statistic (not an approximation) so it will apply to small datasets as well.

```
stats.fisher_exact(chest_array)
```

```
SignificanceResult(statistic=1.3534040369483407, pvalue=0.2088776906675503)
```

2.5.2 Measures of Association

Measures of association quantify how two random variables vary together. When both variables are continuous, a common measure that is used is Pearson correlation. We shall return to this in topics such as regression. For now, we touch on bivariate measures of association for contingency tables (two categorical variables).

2.5.3 Odds Ratio

The most generally applicable measure of association, for 2x2 tables with nominal variables, is the Odds Ratio (OR). Suppose we have X and Y to be Bernoulli random variables with (population) success probabilities p_1 and p_2.

We define the odds of success for X to be

$$\frac{p_1}{1-p_1}$$

Similarly, the odds of success for random variable Y is $\frac{p_2}{1-p_2}$.

In order to measure the strength of their association, we use the *odds ratio*:

$$\frac{p_1/(1-p_1)}{p_2/(1-p_2)}$$

The odds ratio can take on any value from 0 to ∞.

- A value of 1 indicates no association between X and Y. If X and Y were independent, this is what we would observe.
- Deviations from 1 indicate stronger association between the variables.
- Note that deviations from 1 are not symmetric. For a given pair of variables, an association of 0.25 or 4 is the same - it is just a matter of which variable we put in the numerator odds.

Due to the above asymmetry, we often use the log-odds-ratio instead:

$$\log \frac{p_1/(1-p_1)}{p_2/(1-p_2)}$$

- Log-odds-ratios can take values from $-\infty$ to ∞.
- A value of 0 indicates no association between X and Y.
- Deviations from 0 indicate stronger association between the variables, and deviations are now symmetric; a log-odds-ratio of -0.2 indicates the same *strength* as 0.2, just the opposite direction.

To obtain a confidence interval for the odds-ratio, we work with the log-odds ratio and then exponentiate the resulting interval. Here are the steps:

1. The sample data in a 2x2 table can be labelled as $n_{11}, n_{12}, n_{21}, n_{22}$.
2. The *sample* odds ratio is
$$\widehat{OR} = \frac{n_{11} \times n_{22}}{n_{12} \times n_{21}}$$
3. For a large sample size, it can be shown that $\log \widehat{OR}$ follows a Normal distribution. Hence a 95% confidence interval can be obtained through
$$\log \frac{n_{11} \times n_{22}}{n_{12} \times n_{21}} \pm z_{0.025} \times ASE(\log \widehat{OR})$$

where

- the ASE (Asymptotic Standard Error) of the estimator is
$$\sqrt{\frac{1}{n_{11}} + \frac{1}{n_{12}} + \frac{1}{n_{21}} + \frac{1}{n_{22}}}$$

Example 2.14 (Example: Chest Pain and Gender Odds Ratio)**.** Let us compute the confidence interval for the odds ratio in the chest pain and gender example from earlier.

```
chest_tab2 = sm.stats.Table2x2(chest_array)

chest_tab2.summary()
```

	Estimate	SE	LCB	UCB	p-value
Odds ratio	1.353		0.863	2.123	0.188
Log odds ratio	0.303	0.230	-0.148	0.753	0.188
Risk ratio	1.322		0.872	2.004	0.188
Log risk ratio	0.279	0.212	-0.137	0.695	0.188

2.5.4 For Ordinal Variables

When both variables are ordinal, it is often useful to compute the strength (or lack) of any monotone trend association. It allows us to assess if

> As the level of X increases, responses on Y tend to increase toward higher levels, or responses on Y tend to decrease towards lower levels.

For instance, perhaps job satisfaction tends to increase as income does. In this section, we shall discuss a measure for ordinal variables, analogous to Pearson's correlation for quantitative variables, that describes the degree to which the relationship is monotone. It is based on the idea of a concordant or discordant pair of subjects.

- A **pair of subjects** is *concordant* if the subject ranked higher on X also ranks higher on Y.
- A **pair** is *discordant* if the subject ranking higher on X ranks lower on Y.
- A **pair** is *tied* if the subjects have the same classification on X and/or Y.

If we let

- C: number of concordant pairs in a dataset, and
- D: number of discordant pairs in a dataset.

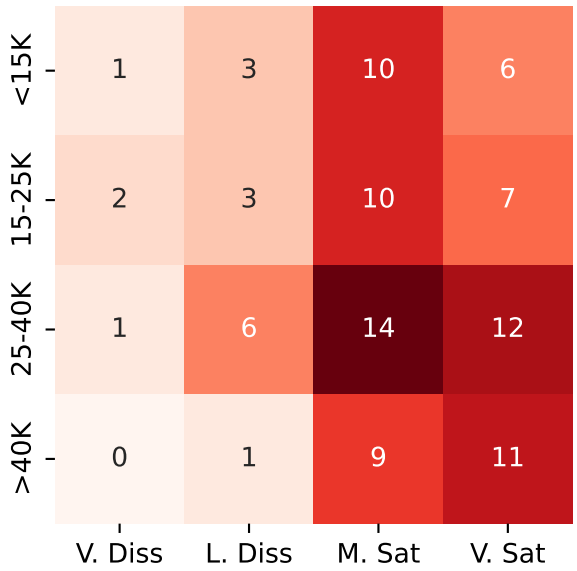
Then if C is much larger than D, we would have reason to believe that there is a strong positive association between the two variables. Here are two measures of association based on C and D:

1. Goodman-Kruskal γ is computed as
$$\gamma = \frac{C - D}{C + D}$$
2. Kendall τ_b is
$$\tau_b = \frac{C - D}{A}$$
where A is a normalising constant that results in a measure that works better with ties, and is less sensitive than γ to the cut-points defining the categories. γ has the advantage that it is more easily interpretable.

For both measures, values close to 0 indicate a very weak trend, while values close to 1 (or -1) indicate a strong positive (negative) association.

Example 2.15 (Example: Job Satisfaction by Income)**.** Consider the following table, obtained from Agresti (2012). The original data come from a nationwide survey conducted in the US in 1996.

```
us_svy_tab = np.array([[1, 3, 10, 6],
                       [2, 3, 10, 7],
                       [1, 6, 14, 12],
                       [0, 1,  9, 11]])
col_names = ['V. Diss', 'L. Diss', 'M. Sat', 'V. Sat']
row_names = ['<15K', '15-25K', '25-40K', '>40K']
sns.heatmap(us_svy_tab, annot=True, square=True, fmt='',
            xticklabels=col_names,
            yticklabels=row_names,
            cmap='Reds', cbar=False, );
```



In measuring the association between these two variables (job satisfaction and income), we are interested in answering questions such as:

> If individual A has higher income than individual B, is individual A more likely to be satisfied in his/her job?

γ and τ_b are measures that quantify this association. For the function in `scipy.stats` that computes this association, we need the data in "long format". Hence we unroll it manually before summoning the method.

```
dim1 = us_svy_tab.shape
x = []; y=[]
for i in range(0, dim1[0]):
    for j in range(0, dim1[1]):
        for k in range(0, us_svy_tab[i,j]):
```

```
            x.append(i)
            y.append(j)
```

```
stats.kendalltau(x, y, variant='b')
```

```
SignificanceResult(statistic=0.15235215134659688, pvalue=0.0860294855671433)
```

The output shows that $\tau_b = 0.15$ is positive, and is borderline significant at 5% level. We can conclude that there is a weak association between job satisfaction and income.

2.6 Summary

In this topic, the primary take-aways are the notions of hypothesis tests (HT) and confidence intervals (CI). Both of these approaches aim to uncover information about a population using a sample. I strongly advocate the choice of CI over HT, as the latter option only provides a binary decision. CIs, on the other hand, provide a range of values to provide a better understanding of the situation.

Using a small set of applications, we have demonstrated these techniques. We covered a common scenario where a researcher may need to compare the means between several groups. We then moved onto another common situation, where one might be faced with a contingency table. A common query is: What should we do if the assumptions of the test are not fulfilled? In those circumstances, one possibility is to turn to nonparametric versions of the test. For instance, the Kruskal-Wallis test is a HT for comparing the means of several groups whose distributions are not Normal.

In the section on categorical data, we introduced measures of association for categorical variables - odds ratio, and Kendall τ. When we have contingency tables, a common choice has been to display barcharts, or heat maps and use those to make decisions on. However, I encourage you to use the measures of association instead. They are intuitive to understand, and can allow you to compare sub-groups or variation of association over time.

With HT and with CI, there has been a tendency to manipulate the data or tests until the desired outcome has been reached. Do be watchful of this. Conducting multiple tests increases the false positive error rate, so please do avoid this as well. If you conduct multiple tests, adjust for multiple testing. Use statitsical inference techniques as a guide together with common sense and domain expertise.

2.7 References

2.7.1 Website References

1. Inference recap from Penn State:
    - Hypothesis testing recap
    - Confidence intervals recap
2. Tests for Normality More information on the Kolmogorov-Smirnov and Shapiro-Wilks Tests for Normality.

3. Overview of *t*-tests This page includes the rule of thumb about deciding when to use the equal variance test, and when to use the unequal variances version.

2.7.2 Documentation links

1. statsmodels ANOVA A more complete example on the application of ANOVA. Return to this when we complete the topic on regression.

2. Useful functions from scipy.stats: Under the "Hypothesis Tests" section of this page, you can find:

   - Kruskal Wallis - the nonparametric equivalent of ANOVA
   - Wilcoxon signed rank - the nonparametric equivalent of paired sample t-test
   - Mann Whitney test - the nonparametric equivalent of independent samples t test
   - Somer's D - measure of association for two categorical variables (one ordinal and one nominal).

3 Unsupervised Learning

3.1 Introduction

Suppose that we have a set of N observations $(x_1, x_2, \ldots, x_N)$ of a random p-vector X. The goal in unsupervised learning is to infer properties of the probability density of X. Note the primary difference with supervised learning - in that context, we had a set of labels $y_1, \ldots, y_N$ in addition to the x_i's. Here, we do not have labelled data.

In situations where x_i is vector of 3 or less, then graphical methods and numerical summaries such as correlations will suffice to help us understand the structure of the data. However, these methods breakdown as soon as p increases beyond 3. This topic introduces techniques that we can use, even when p is large, to :

1. Understand and interpret the main sources of variation in the data,
2. Identify "groups" or clusters within the data for further study.
3. Visualise high-dimensional data

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px

from itables import show
from ind5003 import clust
import folium
import geopandas

from sklearn import decomposition, preprocessing
from sklearn.metrics import pairwise_distances
from sklearn.manifold import MDS, TSNE
from sklearn.ensemble import IsolationForest

from scipy.cluster import hierarchy

from sentence_transformers import SentenceTransformer
```

Example 3.1 (Wine Quality Data)**.** The UCI Machine Learning Repository contains a dataset on Wine Quality. It consists of two tables - one corresponding to white wine and one corresponding to red wine. Each table contains the following columns:

1. fixed acidity
2. volatile acidity
3. citric acid

4. residual sugar
5. chlorides
6. free sulfur dioxide
7. total sulfur dioxide
8. density
9. pH
10. sulphates
11. alcohol
12. quality (score between 0 and 10)

Columns 1 - 11 are numeric variables, measured objectively on the wines. Column 12 is a subjective evaluation made by wine experts, based on sensory data. Each quality score is the median of at least 3 evaluations. Although this dataset was created for a supervised learning problem, we shall use it to practice unsupervised learning techniques. To do so, we shall ignore the column corresponding to quality in most sections until the end, when we try to interpret the findings.

Our first step is to read in the two tables and combine them into one.

```
wine_red = pd.read_csv("data/wine+quality/winequality-red.csv",
                       delimiter=";" )
wine_red['type'] = "red"
wine_white = pd.read_csv("data/wine+quality/winequality-white.csv",
                         delimiter=";")
wine_white['type'] = "white"

# remove spaces in column names:
col_names = ['fixed_acidity', 'volatile_acidity', 'citric_acid', 'residual_sugar',
             'chlorides', 'free_sulfur_dioxide', 'total_sulfur_dioxide',
             'density', 'pH', 'sulphates', 'alcohol', 'quality', 'type']
wine2 = pd.concat([wine_red, wine_white], ignore_index=True)
wine2.columns = col_names
```

Here is a brief overview of the data.

```
print(wine2.head())
```

```
   fixed_acidity  volatile_acidity  citric_acid  residual_sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free_sulfur_dioxide  total_sulfur_dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56
```

```
   alcohol  quality type
0      9.4        5  red
1      9.8        5  red
2      9.8        5  red
3      9.8        6  red
4      9.4        5  red
```

3.2 Principal Components Analysis

A Principal Components Analysis (PCA) explains the covariance matrix of a set of variables through a few *linear combinations* of these variables. The general objectives are

1. data reduction, into features that are uncorrelated with one another,
2. interpretation, and
3. visualisation.

3.2.1 Formal Set-up

Suppose that we have N observations of a random vector of length p. We can represent these values in a matrix with N rows and p columns:

$$\mathbf{X}_{N\times p} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ \cdots & \cdots & \cdots & \cdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,p} \end{bmatrix}$$

Let $\mathbf{x}_j = \begin{bmatrix} x_{1,j} & x_{2,j} & \cdots & x_{N,j} \end{bmatrix}^T$ correspond to column j in $\mathbf{X}$, for $j = 1, \ldots, p$. We represent the mean of column j with

$$\bar{x}_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

The first step in a PCA is to compute the covariance matrix of the data:

$$S_{p\times p} = \begin{bmatrix} s_1^2 & s_{1,2}^2 & \cdots & s_{1,p}^2 \\ \cdots & \cdots & \cdots & \cdots \\ s_{p,1}^2 & s_{p,2}^2 & \cdots & s_p^2 \end{bmatrix}$$

where $s_{m,n}^2 = \frac{1}{N-1}\sum_{k=1}^{N}(x_{k,m} - \bar{x}_m)(x_{k,n} - \bar{x}_n)$ is the sample covariance between columns m and n of matrix X, where $1 < m, n < p$. A PCA analysis yields coefficients a_i such that:

$$\mathbf{y}_i = a_{i,1}\mathbf{x}_1 + a_{i,2}\mathbf{x}_2 + \cdots + a_{i,p}\mathbf{x}_p, \quad i = 1, \ldots, p$$

In other words, $\mathbf{y}_i$ is a column vector of length N, formed from a linear combinations of the columns in the original $\mathbf{X}$ matrix. Each $\mathbf{y}_i$ is what we refer to as a principal component. From a symmetric $p \times p$ matrix, we can always compute p principal components, and these vectors will be uncorrelated with each other.

> **Note**
>
> However this does not help us much! We have not achieved any reduction!?

The value of PCA comes from the possibility that the *first few* principal components usually explain most of the variability in the data ($\sum_i^p s_i^2$). The last few principal components typically explain little of the variability in the data. Indeed, there is some loss of information when we drop them, but the benefit is that we can (hopefully) focus on much fewer dimensions than the original p (which could be in the hundreds, even). Moreover, as these components will be uncorrelated by design, they can be used as features to solve any issues of multicollinearity in our data.

Example 3.2 (Example: PCA on Wine Dataset)**.** While it is possible to extract principal components using either the covariance matrix or the correlation matrix, using the latter avoids situations where the primary principal component is simply driven by the scale of one or more columns in the original dataset. Here, we scale the first 11 columns (exclude quality and type) so that each column has mean 0 and variance 1.

```
X_raw = wine2.iloc[:, :-2]
scaler = preprocessing.StandardScaler().fit(X_raw)
X_scaled = scaler.transform(X_raw)
```

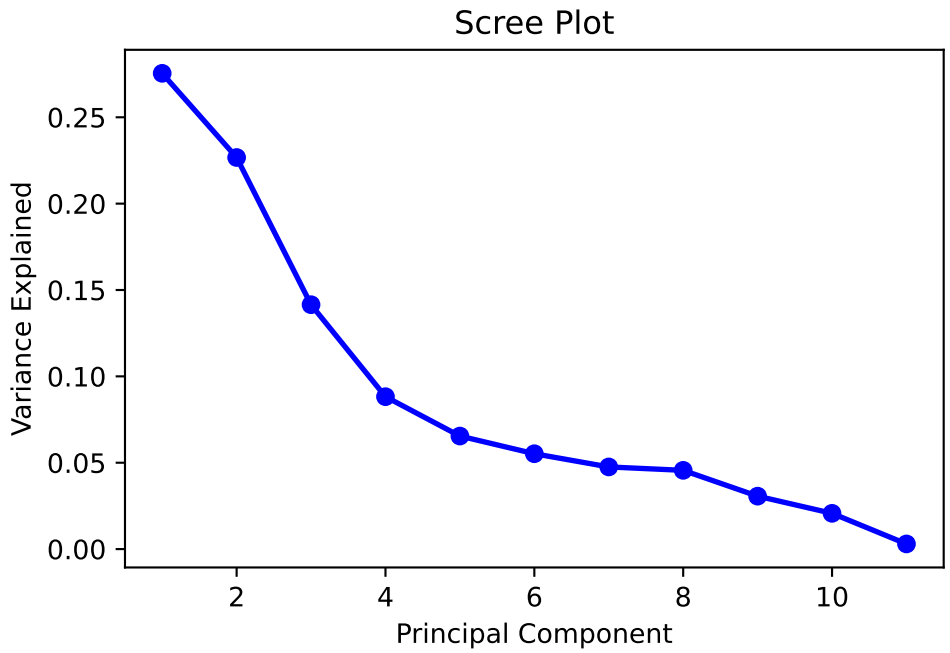
It is theoretically possible to extract 11 components from this X matrix. Let us proceed with that, and assess how many we should keep using a *scree* plot.

```
pca_full = decomposition.PCA(n_components=11)
pca_full.fit(X_scaled)
```

n_components	11
copy	True
whiten	False
svd_solver	'auto'
tol	0.0
iterated_power	'auto'
n_oversamples	10
power_iteration_normalizer	'auto'
random_state	None

A scree plots the variance explained by each subsequent principal component (on the y-axis) versus the order of the principal component. It indicates how much more value there is in including the subsequent principal component. Generally, we look for an "elbow" shape to inform us of how many to keep. From below, we would probably want to retain 4 or 5 principal components.

```
PC_values = np.arange(pca_full.n_components_) + 1
plt.plot(PC_values, pca_full.explained_variance_ratio_, 'o-', linewidth=2, color='blue')
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Variance Explained');
```



To find the amount of total variance explained, we can use the following command:

```
pca_full.explained_variance_ratio_.cumsum()
```

```
array([0.2754426 , 0.50215406, 0.64364015, 0.73187216, 0.79731533,
       0.85252548, 0.90008537, 0.94567722, 0.97631577, 0.99701538,
       1.        ])
```

It appears that 5 components are enough to explain 79.7% of the variance. Let us try to take a look at the $a_{i,j}$ coefficients matrix to interpret the principal components. This set of coefficients is also known as the loadings matrix.

```
pca = decomposition.PCA(n_components=5)
pca.fit(X_scaled)
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
loading_matrix = pd.DataFrame(loadings,
                              columns=['PC' + str(x+1) for x in range(0, 5)],
                              index=col_names[:-2])
```

In social sciences especially, it is a convention to drop loadings that are smaller than 0.3 in absolute value and then to interpret the remaining coefficients as well as possible.

```
loading_matrix2 = loading_matrix.copy()
loading_matrix2[loading_matrix.abs()  < 0.300] = 0.00
loading_matrix2.round(3).style.background_gradient(cmap='coolwarm_r',
                                                   vmin=-1, vmax=1)
```

Table 3.2

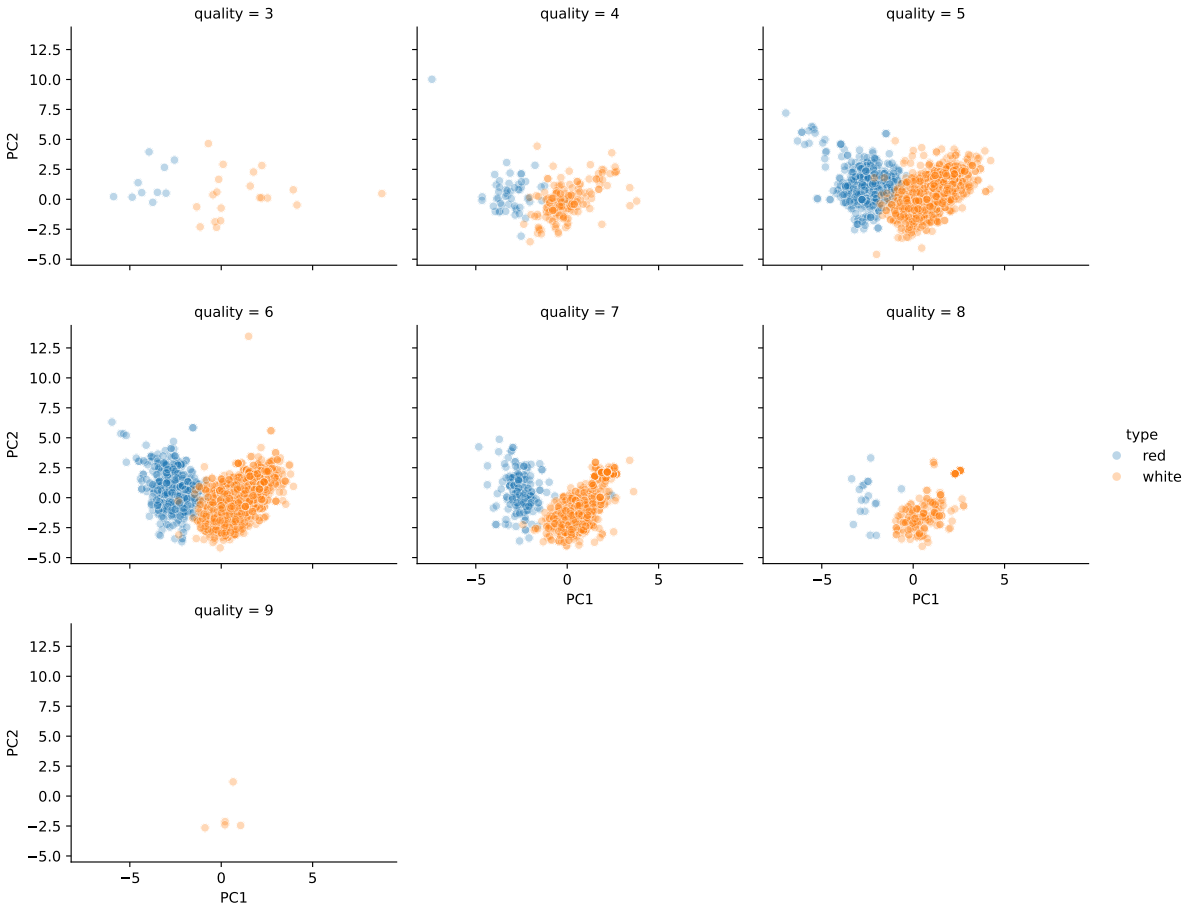
	PC1	PC2	PC3	PC4	PC5
fixed_acidity	-0.416000	0.531000	0.542000	0.000000	0.000000
volatile_acidity	-0.663000	0.000000	-0.383000	0.000000	0.000000
citric_acid	0.000000	0.000000	0.737000	0.000000	0.000000
residual_sugar	0.602000	0.521000	0.000000	0.000000	0.000000
chlorides	-0.505000	0.498000	0.000000	0.000000	0.521000
free_sulfur_dioxide	0.750000	0.000000	0.000000	0.352000	0.000000
total_sulfur_dioxide	0.848000	0.000000	0.000000	0.000000	0.000000
density	0.000000	0.922000	0.000000	0.000000	0.000000
pH	-0.381000	0.000000	-0.568000	0.408000	-0.385000
sulphates	-0.512000	0.303000	0.000000	0.631000	0.000000
alcohol	0.000000	-0.734000	0.326000	0.000000	0.000000

From above, if we focus on the top two principal components, we could interpret them as:

1. A combination of sugar and sulphur dioxides contrasted against acidity, chlorides and sulphates.
2. A contrast between density and alcohol.

In the following code, we **apply** the transformation to obtain the actual principal components ($\mathbf{y}_j$'s).

```
X_transformed = pca.transform(X_scaled)
X_transformed_df = pd.DataFrame(X_transformed,
                                columns=['PC1', 'PC2', 'PC3', 'PC4', 'PC5'])
X_transformed_df[['quality', 'type']] = wine2[['quality', 'type']]

sns.relplot(data=X_transformed_df, x='PC1', y='PC2', col='quality', col_wrap= 3,
            hue='type', marker='o', alpha=0.3, height=3, aspect=1.2);
```



Judging from the plots, red wines tend to be lower on PC1. Can we tie this back to the columns in the data to understand the difference between white and red wines more?

For white wines, there seems to be a linear relation between PC1 and PC2. However, for red wines, the range of PC1 values is quite narrow, and the PC1 values do not appear to suggest what the PC2 values could be.

Finally, neither PC1 nor PC2 appears to provide a clue on the the subjective quality of the wine.

3.3 Clustering

In the previous section, the goal was to reduce the dimensionality of the dataset. In this section, our goal is to segment the data. By assigning individual observations into groups (or clusters) such that those within each group are "closely related", we can gain an understanding of our data at a higher level.

There are many different clustering algorithms. You may have heard of K-means, a very popular one before. The one we are going to use here is very similar to it. It is known as **agglomerative hierarchical clustering**. Let's take a look at how it works first.

3.3.1 Hierarchical Clustering

Dissimilarity Measures Between Individual Observations

As we mentioned earlier, cluster analysis tries to identify groups such that those within a group a "similar" to one another. In order to proceed, we need to formalise this idea of similarity/dissimilarity.

As before, suppose that we have N observations $x_1, x_2, x_3, \dots, x_N$ and we wish to group them into K clusters. Each observation is typically a vector of p observations, so we may write $x_i = (x_{i,1}, x_{i,2}, \dots, x_{i,N})$.

Most clustering algorithms require a dissimilarity matrix as input, so we need function that can measure **pairwise dissimilarity**. One of the most common choices is the Euclidean distance (or rather the L2-norm) between x_i and x_j:

$$d(x_i,\ x_j) = \sqrt{\sum_{s=1}^{p}(x_{i,s} - x_{j,s})^2}$$

Another common choice is the $L1$-norm:

$$d(x_i,\ x_j) = \sum_{s=1}^{p} |x_{i,s} - x_{j,s}|$$

Dissimilarity Measures Between Clusters or Groups

For hierarchical clustering, we need to build on this choice of pairwise dissimilarity to obtain a measure of dissimilarity between groups. In other words, suppose we have two groups of points G and H, with N_G and N_H points within them respectively. We wish to use the pairwise dissimilarity between points in G and H, to compute a dissimilarity between G and H. We call this the **linkage method**, and there are several options for this too:

1. Single linkage takes the intergroup dissimilarity to be that of the closest (least dissimilar) pair.
$$d_S(G, H) = \min_{i \in G, j \in H} d(x_i,\ x_j)$$
2. Complete linkage takes the intergroup dissimilarity to be that of the furthest (most dissimilar) pair.
$$d_C(G, H) = \max_{i \in G, j \in H} d(x_i,\ x_j)$$
3. Average linkage utilises the average of all pairwise dissimilarities between the groups:
$$d_A(G, H) = \frac{1}{N_G N_H} \sum_{i \in G} \sum_{j \in H} d(x_i, x_j)$$
4. Ward linkage uses a more complicated distance to minimise the variance within groups. It usually returns more compact clusters than the others. Suppose that group G was formed by merging groups G_1 and G_2. Then the Ward distance between groups is
$$d_W(G, H) = \sqrt{\frac{|H| + |G_1|}{N_G + N_H} d_W(H, G_1)^2 + \frac{|H| + |G_2|}{N_G + N_H} d_W(H, G_2)^2 + \frac{H}{N_G + N_H} d_W(G_1, G_2)^2}$$

The choice of linkage can affect the final clusters we obtain, so it is important to choose carefully based on the subject matter. Here is a plot from sklearn, demonstrating the impact of the linkage on the clusters in toy datasets.

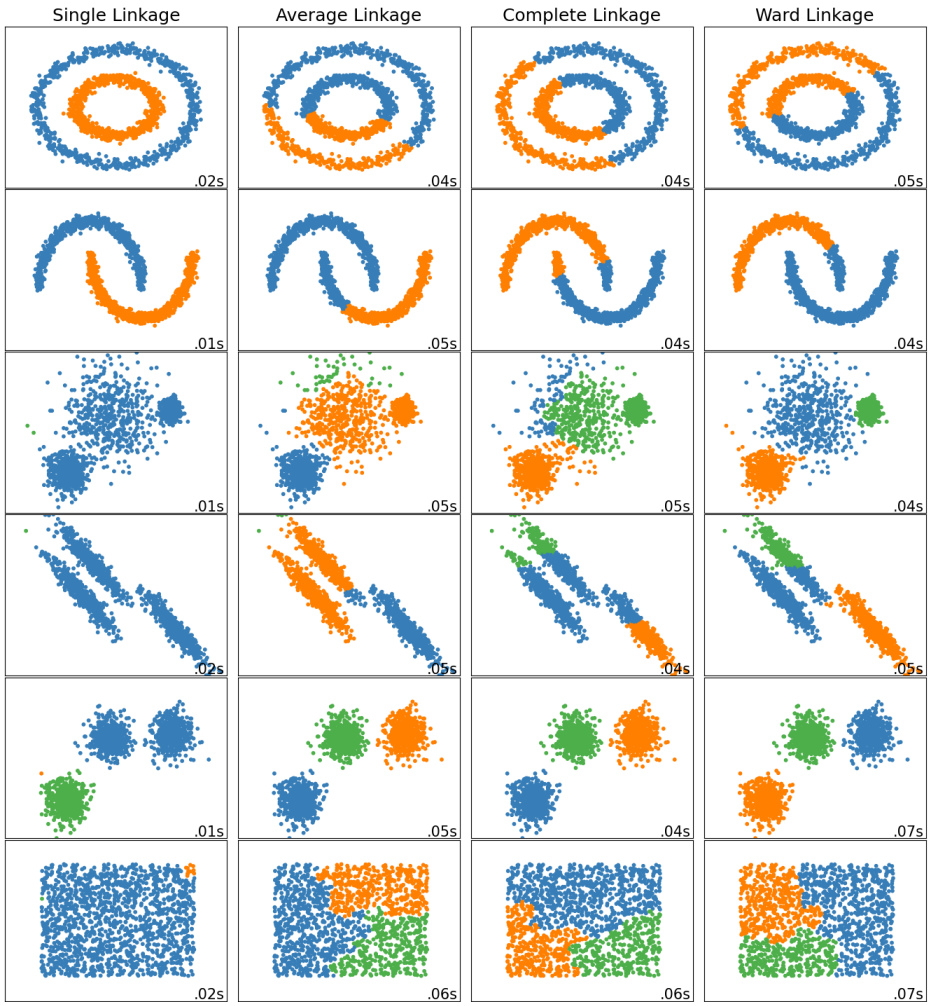


Figure 3.1: sklearn documentation

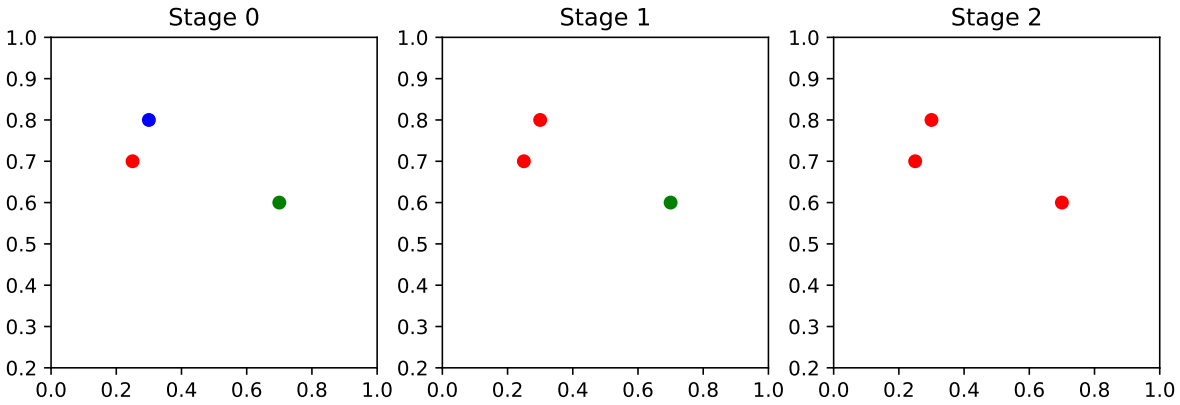
Agglomerative Hierarchical Clustering Algorithm

The output of the algorithm is a hierarchical representation of the data, where clusters at each level of the hierarchy are created by merging clusters at the next lower level. At the lowest level, each cluster contains a single observation. At the highest level there is only one cluster containing all of the data.

Starting at the bottom (with N clusters of singletons), we recursively merge a selected pair of clusters into a single cluster. This produces a grouping at the next higher level with one less cluster. The pair chosen for merging consist of the two groups with the smallest intergroup dissimilarity.

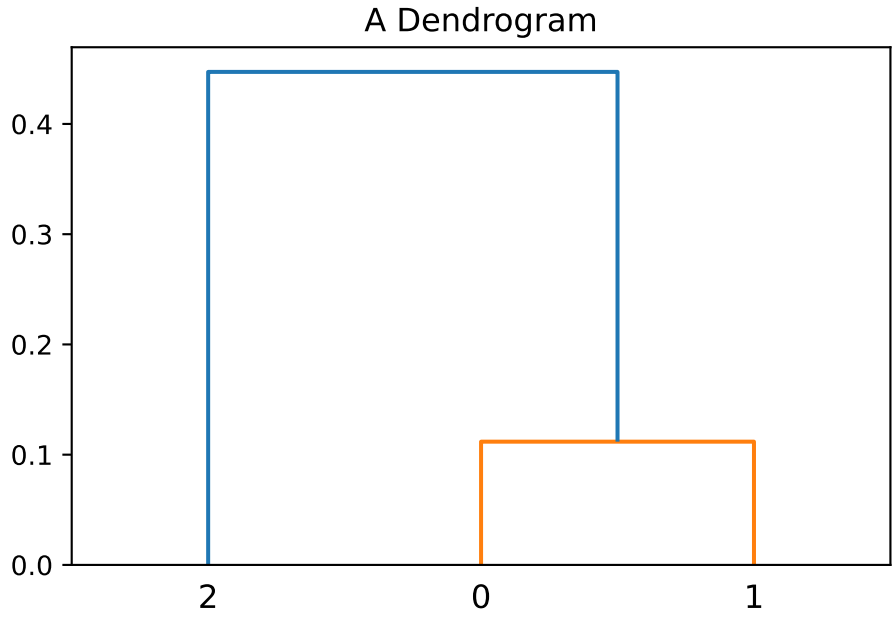
As you can tell, this algorithm does not require the number of clusters as an input. The final number of clusters can be based on a visualisation of this hierarchy of clusterings, through a dendrogram.

```
X = np.array([[.25,.7], [.3, .8], [.7, .6]])
fc_dict={'Stage 0': ['red', 'blue', 'green'], 'Stage 1':['red', 'red', 'green'],
        'Stage 2':['red']*3}

plt.figure(figsize=(10, 3))
for x,y in enumerate(fc_dict.items()):
    plt.subplot(1,3,x+1);
    plt.scatter(X[:,0], X[:,1], facecolor=y[1]);
    plt.ylim(0.2,1); plt.xlim(0,1);
    plt.title(y[0]);
```



As we can see the number of clusters changed from 3 to 2 and then to 1. Here is how we can visualise the hierarchy:

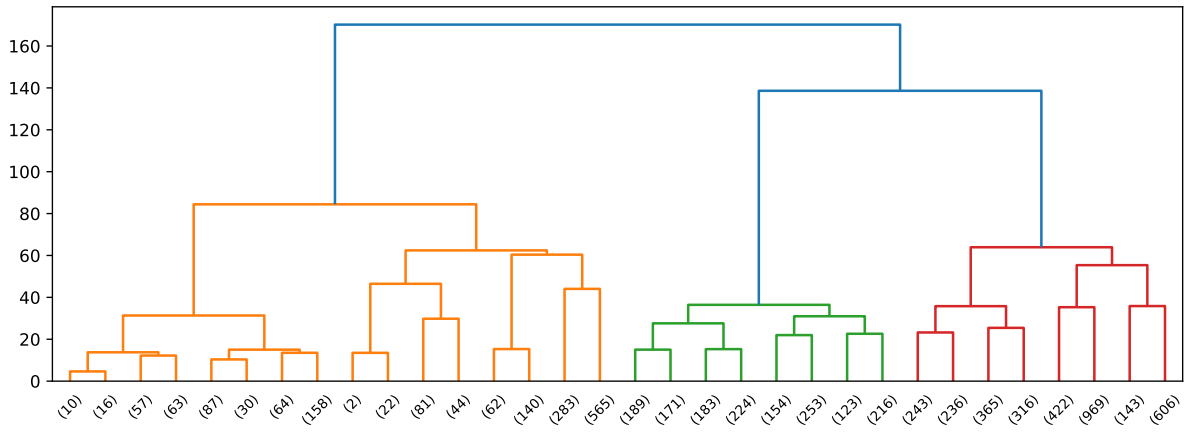
```
lm0 = hierarchy.linkage(X)
hierarchy.dendrogram(lm0,p=2)
plt.title('A Dendrogram');
```



The dendrogram shows that points with index 0 and 1 (the closest two points) merge at a small vertical distance (height of green lines), but the group containing them merges with point 2 at a much higher vertical distance (blue line on the left). This shows that points 0 and 1 are less dissimilar to one another than they are (as a group) to point 2. In other words, the *height of each node is proportional to the value of the intergroup dissimilarity between its two child nodes.*

Example 3.3 (Example: Hierarchical Clustering of Wine)**.** Now we apply this technique to the scaled wine dataset.

```
hc1 = hierarchy.linkage(X_transformed_df.iloc[:, :-2], method='ward')

plt.figure(figsize=(12,4))
hierarchy.dendrogram(hc1, p=4, truncate_mode='level');
```



From the dendrogram alone, it appears plausible that we can break the original set of data points into two, maybe 3, groups.

> **Note**
>
> Can we come up with a formal method of determining the optimal number of clusters?

3.3.2 Determining the optimal number of clusters

The Silhouette coefficient summarises the within similarity to the between similarity using the following formula:

$$S = \frac{b - a}{\max(a, b)}$$

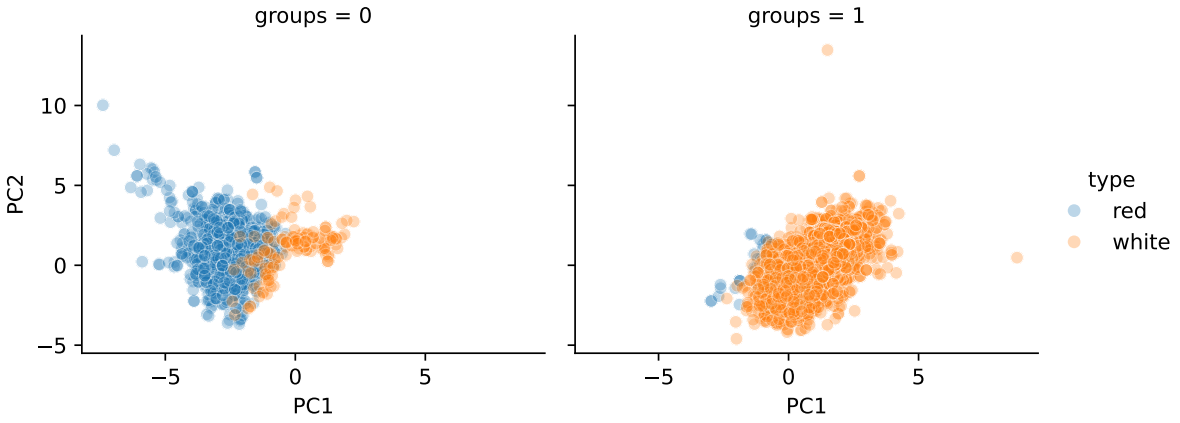
where b is the average distance between an observation and a cluster that it is not a part of. On the other hand, a is the mean distance within a cluster. This coefficient takes values between -1 and 1, with values closer to 1 indicating a more optimal clustering.

Example 3.4 (Example: Clustering Quality)**.** Here are the silhouette scores from the two clusterings of the wine data.

```
out = hierarchy.cut_tree(hc1, n_clusters=3).ravel()
X_transformed_df['groups'] = out

clust.compute_silhouette_scores(hc1, X_transformed_df.iloc[:, :-2], [2,3,4])
```

The silhouette coefficient values we are obtaining are not very good. However, out of the possible values we tried, $K = 2$ seems to be the best.

```
out = hierarchy.cut_tree(hc1, n_clusters=2).ravel()
X_transformed_df['groups'] = out

sns.relplot(data=X_transformed_df, x='PC1', y='PC2', col='groups',
            hue='type', marker='o', alpha=0.3, height=3, aspect=1.2);
```



As we can see, the groupings closely mirror the type of wine (red or white).

```
wine2.type.groupby(X_transformed_df.groups).describe()
```

groups	count	unique	top	freq
0	1684	2	red	1549
1	4813	2	white	4763

However, notice that there are a number of white wines grouped as 0 (with most of the other reds). It would be interesting to study what qualities of these wines led to them being grouped with the reds.

Also, consider what value the PCA brought to this problem. Go back and re-run the clustering algorithm with the scaled but untransformed data. Does the quality of clustering differ?

3.4 Outlier Detection

One efficient way of performing outlier detection in high-dimensional datasets is to use random forests. The `ensemble.IsolationForest` object in sklearn "isolates" observations by:

1. Randomly selecting features, and then randomly selecting a split value between the maximum and minimum values of the selected feature to form a decision tree.
    - In each tree, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.
2. Repeating step 1 to create a forest of trees.

For each observation, the average path length, over the forest of random trees, is a measure of how anomalous it is. Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Example 3.5 (Example: Isolation Forest with Taiwan Dataset)**.** Let us apply this technique to the Taiwan real estate dataset from the regression topic.

```
re2 = pd.read_csv("data/taiwan_dataset.csv")
X_re = re2.loc[:, ['trans_date', 'house_age', 'dist_MRT', 'num_stores',
                   'Xs', 'Ys', 'price']]
X_re_scaled = preprocessing.StandardScaler().fit_transform(X_re)
```

After reading in and scaling the data, we fit the `IsolationForest` estimator.

```
clf = IsolationForest(max_samples=300, max_features=2, contamination=0.01,
                      random_state=503)
clf.fit(X_re_scaled)
```

n_estimators	100
max_samples	300
contamination	0.01
max_features	2
bootstrap	False
n_jobs	None
random_state	503
verbose	0
warm_start	False

The `contamination` factor is the proportion of outliers in the dataset. `max_features` corresponds to the number of features to be drawn for *each* base estimator, while `max_samples` is the number of samples (observations) to draw from the original dataset for each base estimator.

```
id_outliers = pd.Series(clf.predict(X_re_scaled))
id_outliers.value_counts()
```

```
 1    409
-1      5
Name: count, dtype: int64
```

There are 5 points that have been identified as outliers (coded as -1). As analysts, we should do our best to understand what property, or combination of features, led to this.

```
re2['outliers'] = id_outliers
```

```
print(re2[['house_age', 'dist_MRT', 'num_stores', 'price',
          'Xs', 'Ys']].groupby(re2.outliers).describe())
```

```
         house_age                                                    \
             count       mean        std   min   25%   50%   75%   max
outliers
-1             5.0  25.900000   7.494331  17.4  18.0  30.9  31.5  31.7
 1           409.0  17.612469  11.401626   0.0   8.9  16.0  27.6  43.8

         dist_MRT                ...        Xs                Ys            \
            count         mean   ...       75%       max  count      mean
outliers                         ...
-1            5.0  6042.906600  ... -4.891503 -4.891503    5.0 -1.982369
 1          409.0  1023.261961  ...  0.995349  3.355106  409.0  0.024234


                std       min       25%       50%       75%       max
outliers
-1         0.618850 -2.903628 -2.095042 -2.095042 -1.423867 -1.394265
 1         1.377105 -4.129094 -0.683469  0.281186  0.990686  5.020761

[2 rows x 48 columns]
```

Here is the same map of the data points, but with colours to code the outlier points.

> **Note**
>
> If we drop the location parameters, would different points be identified as outliers?

3.5 Visualisation

3.5.1 MDS

Multidimensional Scaling is another technique for visualising high-dimensional data. Just like in hierarchical clustering, we begin with a square matrix consisting of all pairwise dissimilarities $d(x_i, x_j)$ between our N high-dimensional vectors. With a choice of k, we seek values $z_1, z_2, \ldots, z_N \in \mathbb{R}^k$ such that the following function is minimised:

$$S(z_1, \ldots, z_N) = \left[\sum_{i \neq j} \left(d(x_i, x_j) - ||z_i - z_j|| \right)^2 \right]^{1/2}$$

MDS is *not* the same as Principal Component Analysis (PCA):

- PCA maximises **variance**, orthogonal to earlier components.
- Principal components are ordered; MDS are not.
- Principal components are linear combinations of the original vectors; MDS output is not.

The goal of MDS is to find a lower dimensional set of vectors whose pairwise Euclidean distances are as close as possible to the dissimilarity matrix of the original vectors.

Example 3.6 (Example: MDS on Disease Symptoms)**.** The dataset `disease.csv` contains a list of symptoms that were reported for a set diseases. Each row in the dataframe corresponds to a particular disease, while each binary column indicates whether that particular symptom was frequently present for this disease.

```
disease = pd.read_csv("data/disease.csv")
```

Our goal is to visualise the 41 diseases - diseases with "similar" symptoms should be plotted "close" to one another. However, the data begets the natural question: How can we define dissimilarity between the symptom lists of two diseases?

For this purpose, we shall use the Jaccard similarity index. For two disease symptom sets A and B, the Jaccard dissimilarity is defined to be

$$J = 1 - \frac{|A \cap B}{|A \cup B|}$$

If there are no common symptoms between the two diseases, then the intersection between the two sets would be the empty set. In that situation, J would take on the maximum value of 1. If the two symptom lists are identical, then J takes on the smallest possible value of 0.

```
disease_names = disease.disease.to_list()
symptoms = disease.columns.to_list()[:-1]

X = disease.iloc[:, 0:-1].to_numpy()

symptom_text = []

for i in range(0, X.shape[0]):
    symptom_text.append(','.join([symptoms[x] for x in np.where(X[i] == 1)[0]]))
disease['symptom_text'] = symptom_text

# pdist2 is 41x41
pdist2 = pairwise_distances(X!=0, metric='jaccard')
disease.loc[disease.disease.isin(['GERD', 'Heart attack']), 'symptom_text'].to_list()
```

```
['stomach_pain,acidity,ulcers_on_tongue,vomiting,cough,chest_pain',
 'vomiting,chest_pain,breathlessness,sweating']
```

Based on the set of symptoms for GERD and Heart attack, we have the following calculation:

$$J = 1 - \frac{2}{8} = 0.75$$

Now we turn to the MDS transformation.

```
embedding = MDS(n_components=2, normalized_stress='auto', dissimilarity='precomputed',
                random_state=42, max_iter=500, verbose=0)
X_transformed = embedding.fit_transform(pdist2)
X_transformed_df = pd.DataFrame(X_transformed, columns=['X', 'Y'])
X_transformed_df['disease'] = disease_names

fig = px.scatter(X_transformed_df, x='X', y='Y', text='disease', hover_name=symptom_text,
                 width=1024, height=960)
fig.update_traces(textposition='top center')
#fig.show()
```

```
/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/manifold

The default value of `n_init` will change from 4 to 1 in 1.9.

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html
```

To verify if the plot makes intuitive sense, we should inspect the points that occur nearby to one another. Here are some example codes we can use:

```
disease.loc[disease.disease.isin(['Typhoid', 'Dengue', 'Malaria']),
            'symptom_text'].to_list()
# disease.loc[disease.disease.isin(['Common Cold', 'Pneumonia', 'Bronchial asthma']),
#              'symptom_text'].to_list()
# disease.loc[disease.disease.isin(['GERD', 'Heart attack', 'Drug reaction']),
#             'symptom_text'].to_list()
```

```
['chills,vomiting,nausea,high_fever,diarrhoea,headache,sweating,muscle_pain',
 'skin_rash,chills,vomiting,nausea,loss_of_appetite,high_fever,fatigue,headache,back_pain,musc
 'chills,vomiting,nausea,abdominal_pain,high_fever,fatigue,diarrhoea,headache,constipation,tox
```

It is important to remember that above, we requested the MDS algorithm to return us z-coordinates in $\mathbf{R}^2$ - two-dimensional Euclidean space, that replicate the dissimilarity matrix from the higher-dimensional data. This was for convenience, since the two dimensional plane is easier to plot. One problem is that, due to information loss, it is possible that in 2D, two points appear near, but in fact, they may be far apart on a third dimension.

With `plotly`, we can make interactive 3d plots, which can go some way to alleviating this problem.

> **Warning**
>
> Stay away from non-interactive 3d plots!

3.5.2 t-SNE

t-SNE is an fast, iterative algorithm for *visualising* high-dimensional data.

Once again, suppose we have N *data points* $x_i \in \mathbf{R}^p$. We would like to choose N *map points* $y_i \in \mathbf{R}^2$ to represent them. Here is how the algorithm works:

1. Compute pairwise similarity between the *data points*, using a Gaussian (Normal) kernel.
2. Iteratively update *map points* so that their pairwise similarity is as close as possible to the original data points.

The innovation of these algorithm is that the similarity between map points is computed using a t-distribution instead of Gaussian. The t-distribution has fatter tails than the Normal. This ensures that data points that are not close in $\mathbf{R}^D$ are pushed apart in the map points space.

There are a couple of important parameters in this algorithm.

1. *Perplexity*: This is a parameter that provides a guide on how many neighbours a point has. It is recommended to try different values between 5 and 50 and assess if results are meaningful and consistent.
2. *Number of iterations*: This is the number of adjustments to make to the map points before stopping. The difference between map point similarity and data point similarity is measured using Kullback-Leibler Divergence. When this no longer drops quickly, we can stop the t-SNE algorithm.

Although you will find t-SNE used in many analyses, you should be aware of certain caveats when using it. Please take a look at the link in the references for plots that expound on the following points.

1. t-SNE is for visualisation and exploration, not for clustering.
2. Distances in the map space are not reflective of true distances between datapoints.
3. t-SNE preserves small pairwise distances so that local relationships are preserved.
4. In the map space, distances between clusters (between far-away points) might not mean much (unlike MDS).
5. Make sure you try with different perplexity values, and check that the algorithm has converged.

Example 3.7 (Example: Twitter Dataset)**.** The UCI Machine Learning repository contains tweets pertaining to health news from more than 15 major news agencies in 2015. In this example, we are going to encode each BBC tweet in to a numerical vector of length 384. Then we are going to visualise it using t-SNE.

```
model = SentenceTransformer('sentence-transformers/all-MiniLM-L12-v2')
#sentences = ["This is an example sentence. can we move on?", "Each sentence is converted"]
#embeddings = model.encode(sentences)
#embeddings.shape
```

```
bbchealth_df = pd.read_table('data/health+news+in+twitter/Health-Tweets/bbchealth.txt',
                             delimiter='|', names=['id', 'datetime', 'tweet'])
bbchealth_tweets = bbchealth_df.tweet
print(bbchealth_tweets[10])
```

```
Have GP services got worse? http://bbc.in/1Ci5c22
```

Above, we have an example of a tweet. We are going to strip off the URL at the end before encoding each (short) sentence.

```
t1 = bbchealth_tweets.str.replace(' http.*$', '', regex=True)
t2 = t1.str.replace('^VIDEO:', '', regex=True)
t2_l = t2.to_list()

embeddings = model.encode(t2_l)
```

Example 3.8 (Example: Twitter Dataset t-SNE Output)**.** The following code generates an interactive plot based on the t-SNE visualisation.

```
tsne1 = TSNE(n_components=2, init="random", perplexity=10, verbose=0,
             random_state=43, max_iter=5000)
X_transformed2 = tsne1.fit_transform(embeddings)
```

The plot is only visible in the HTML version of the textbook.

3.6 References

3.6.1 Website references

1. Clustering performance evaluation There are many other ways of assessing what the optimal number of clusters should be.
2. More information on isolation forests
3. Wikipedia entry on Jaccard Similarity (or index): This page introduces variants of the index, which may be relevant when one has *counts* of the number of times each item appears in the set.
4. Caveats when using t-SNE
5. DataCamp on t-SNE: This tutorial consists of a worked-through example on a churn dataset. It includes a comparison with PCA.

3.6.2 Video references

1. t-SNE, explained by Josh Starmer
2. A general video on high-dimensional space: A nice explainer from Google.

4 Natural Language Processing

4.1 Introduction

In our world, Natural Language Processing (NLP) is used in several scenarios. For example,

- phones and handheld computers support predictive text and handwriting recognition;
- web search engines give access to information locked up in unstructured text;
- machine translation allows us to understand texts written in languages that we do not know;
- text analysis enables us to detect sentiment in tweets and blogs.

But as we begin to explore Natural Language, we realise that it is an extremely difficult subject. Here are some specific points to note:

1. Some words mean different things in different contexts, but us humans know which meaning is being used.
    - He **served** the **dish**.
2. In the following two sentences, the word "by" has different meanings:
    - The lost children were found by the lake.
    - The lost children were found by the search party.
3. In the following cases, we (humans) can resolve what "they" is referring to, but it is not easy to generate a simple rule that a computer can follow.
    - The thieves stole the paintings. They were subsequently recovered.
    - The thieves stole the paintings. They were subsequently arrested.
4. How can we get a computer to understand the following tweet?:
    - "Wow. Great job st@rbuck's. Best cup of coffee ever."

```
import numpy as np
import pandas as pd

from itables import show
from IPython.display import YouTubeVideo, display, HTML
import ipywidgets as widgets
import pprint

import gensim
from gensim.parsing.preprocessing import *
import gensim.downloader as api
from nltk.stem import PorterStemmer, WordNetLemmatizer
```

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn import manifold

from transformers import pipeline

import pyLDAvis
import pyLDAvis.gensim_models as gensimvis
import matplotlib.pyplot as plt
import plotly.express as px
```

> **Note**
>
> Can you catch all three jokes in the movie clip below?

```
video = YouTubeVideo("NfN_gcjGoJo", width=640, height=480)
display(video)
```

4.2 Definitions

Before we go on, it would be useful to establish some terminology:

- A **corpus** is a collection of documents.
  - Examples are a group of movie reviews, a group of essays, a group of paragraphs, or just a group of tweets.
  - Plural of corpus is **corpora**.
- A **document** is a single unit within a corpus.
  - Depending on the context, examples are a single sentence, a single paragraph, or a single essay.
- **Terms** are the elements that make up the document. They could be individual words, bigrams or trigrams from the sentences. These are also sometimes referred to as **tokens**.
- The **vocabulary** is the set of all terms in the corpus.

Consider the sentence:

```
I am watching television.
```

The process of splitting up the document into tokens is known as tokenisation. The result for the above sentence would be

```
'I', 'am', 'watching', 'television', '.'
```

How we tokenize and pre-process things will affect our final results. We shall discuss this more in a minute.

4.3 Overview of Applications

Here are some of the use-cases that we shall discuss:

1. *Topic Modeling*: This is an unsupervised technique that allows us to identify the salient topics of a new document automatically. This could be useful in a customer feedback setting, because it would allow quick allocation or prioritisation of resources. This approach requires one to decide on the number of topics. It typically also requires some study of the topics in order to interpret and verify them.
2. *Information Retrieval*: This is also an unsupervised approach. If the new document can be considered a "query", then this can be used to retrieve and prioritise documents that are relevant to the query.
3. *Sentiment Analysis*: Using a lexicon of words and their tagged sentiments, we can assess whether the sentiment in a document is mostly positive or negative.

4.4 Text Pre-processing

Example 4.1 (Example: Wine Reviews Dataset)**.** A dataset containing wine reviews is accessible from Kaggle. We shall work with one of the csv files. It contains 130,000 rows, although some are duplicates.

```
rng = np.random.default_rng(5001)

wine_reviews = pd.read_csv("data/winemag-data-130k-v2.csv", index_col=0)
wine_reviews.drop_duplicates(inplace=True)

pp = pprint.PrettyPrinter(indent=4, compact=True,)
for x in rng.choice(wine_reviews.description, size=5):
    pprint.pp(x)
```

```
('Bell pepper and sharp red fruit aromas provide a shaky start, which is '
 'followed by cranberry, tart cherry and other pointed flavors. The feel is '
 'racy and tight, with gritty acids. Airing does improve it somewhat. Tasted '
 'twice; this is a review of the better bottle. Cabernet, Merlot, Cab Franc '
 'and Carmenère is the blend. From Brazil.')
('Consistent with previous releases, this Michel Rolland effort is a soft, '
 'silky, smoky wine that introduces itself with round cherry fruit and then '
 'charges ahead with layers of licorice, citrus, coffee and rock that enliven '
 'the finish. There is plenty of tart raspberry fruit to open, and the '
 "balancing acids to give the wine a tight core. It's a very polished and "
 'appealing balance of forward, approachable fruit married to more elegant, '
 'ageworthy tannins and acids.')
("This is a light and soft selection, with an upfront gamy note that's framed "
 'by soft strawberry, rhubarb, red cherry and currant fruit tones on the nose '
 "and mouth. Overall, it's short and direct; drink up.")
('Fresh, clean and easy, this would taste great at an outdoor pool party or '
 'during a sunny lunch outdoors. It delivers fresh crispness, with lingering '
 'tones of green apple and passion fruit.')
('This easy-drinking wine has a bouquet of honeydew melon and lime juice. '
 'Flavors of lemon, tangerine, guava and white peach with a soft hint of '
 'baking spice continue into the finish, which is marked by flavors of stone '
 'fruits and nutmeg.')
```

The "descriptions" column contains the review for a particular wine by a user, whose name and twitter handle are provided. Also included is information such as the price, originating county, region of the wine, and so on. In our activity, we are going to apply NLP techniques to the wine reviews.

4.4.1 Pre-processing Text with Gensim

Text documents consist of sentences of varying lengths. Usually, the first step to analysing a document is to break it up into pieces. This process is known as **tokenizing**. When tokenizing a document, we can do it at several levels of resolution: at the sentence, line, word or even punctuation level.

Tokenizing can easily done using the `.split()` within built-in python. But after that, we need to further pre-process the tokens.

The `gensim` package includes a module for pre-processing text strings. Here is a list of some of the functions there:

- `strip_multiple_whitespaces`
- `strip_non_alphanum`
- `strip_numeric`
- `strip_punctuation`
- `strip_short`

Since what we are about to do in the initial part of our activity is based on frequency counts of tokens, apart from some of the above steps, we are also going to remove common "filler" words that could end up skewing the eventual probability distributions of counts. These filler words are known as stop words. They were identified by linguists, and they vary from model to model, from Python package to package, and of course, from language to language.

Whether stop-word removal is meaningful or not also depends on your particular application. At times, it is only done in order to speed up the training of a model. However, it is possible to change the entire meaning of a sentence by removing stop-words.

For us, we are going to apply this list of filters to each review:

1. `strip_punctuation()`,
2. `strip_multiple_whitespaces()`,
3. `strip_numeric()`,
4. `remove_stopwords()`,
5. `strip_short()`,
6. `lemmatize()`

Lemmatizing a word is to reduce it to its root word. You will come across **stemming** whenever you read about lemmatizing. In both cases, we wish to reduce a word to its root word so that we do not have to deal with multiple variations of a token, such as ate, eating, and eats.

When we stem a word, the prefix and/or suffix will be removed according to a set of rules. Since it is primarily rule-based, the resulting word may not be an actual English word.

Like stemming, lemmatizing also aims to reduce a word to its root form. However, it differs from stemming in that the final word must be a proper English language word. For this purpose, the algorithm has to be supplied with a lexicon or dictionary, along with the text to be lemmatized.

Here is an example that demonstrates the differences.

```
porter = PorterStemmer()
wn = WordNetLemmatizer()

demo_sentence = 'Cats and ponies have a meeting'.split()
demo_sentence
```

```
['Cats', 'and', 'ponies', 'have', 'a', 'meeting']
```

```
#import nltk
#nltk.download('wordnet')

[porter.stem(x) for x in demo_sentence]
```

```
['cat', 'and', 'poni', 'have', 'a', 'meet']
```

```
[wn.lemmatize(x) for x in demo_sentence]
```

```
['Cats', 'and', 'pony', 'have', 'a', 'meeting']
```

Now let us go ahead and perform the pre-processing on the wine reviews.

```
CUSTOM_FILTER = [lambda x: x.lower(), strip_punctuation,
                 strip_multiple_whitespaces, strip_numeric,
                 remove_stopwords, strip_short]
#CUSTOM_FILTER[1]
all_review_strings = wine_reviews.description.values
#all_review_strings[:3]
all_strings_tokenized = [preprocess_string(x, CUSTOM_FILTER) for x in all_review_strings]
```

Here is an example of the pre-processed review:

```
pp.pprint(all_strings_tokenized[1])
```

```
[   'ripe', 'fruity', 'wine', 'smooth', 'structured', 'firm', 'tannins',
    'filled', 'juicy', 'red', 'berry', 'fruits', 'freshened', 'acidity',
    'drinkable', 'certainly', 'better']
```

At this point in time, what we have is a list of lists. Each sub-list contains the tokens for a particular wine_review. For instance, the original review for row 234 was:

```
pp.pprint(wine_reviews.description.values[233])
pp.pprint(all_strings_tokenized[233])
```

```
('There is an odd, piercing edge to the aromas, a mix of acetic acid and '
 "pungent herb, with a hint of diesel. Somehow it's not off-putting, just "
 'atypical. The light, tart fruit is a mix of rhubarb and cranberry, very '
 'earthy and tasting of dirt and bark in the finish. This could be quite '
 'pleasant with a hearty, rustic dish such as beef Bourgogne.')
[   'odd', 'piercing', 'edge', 'aromas', 'mix', 'acetic', 'acid', 'pungent',
    'herb', 'hint', 'diesel', 'putting', 'atypical', 'light', 'tart', 'fruit',
    'mix', 'rhubarb', 'cranberry', 'earthy', 'tasting', 'dirt', 'bark',
    'finish', 'pleasant', 'hearty', 'rustic', 'dish', 'beef', 'bourgogne']
```

`gensim` does not have a lemmatizer, so we use the nltk lemmatizer on each token.

```
preprocessed_corpus = [[wn.lemmatize(w) for w in dd ] for dd in all_strings_tokenized]
```

4.5 Representation of Text

Tokenisation of the corpus is merely the first step in processing natural language. All mathematical algorithms work on numerical representations of the data. Hence the next step is to convert the text into numeric representations. In natural language, there are two common ways of representing text:

1. Sparse vectors, using tf-idf or PPMI, or
2. Dense embeddings, which could result from word2vec, GLoVe, or from neural models.

4.5.1 Sparse embeddings with Tf-idf

In this section, we demonstrate how we can use Tf-idf (Term frequency-Inverse document frequency) to create vector representations of documents. Consider the following set of three simple text documents. Each document is a single sentence.

```
raw_docs =[
  "Here are some very simple basic sentences.",
  "They won't be very interesting , I'm afraid. ",
  """
  The point of these basic examples is to learn how basic text
  counting works on *very simple* data, so that we are not afraid when
  it comes to larger text documents. The sentences are here just to provide words.
  """]
```

As the name tf-idf suggests, our first step should be to compute the frequency of each term (token) within each document.

```
vectorizer1 = CountVectorizer(stop_words='english', min_df=1)
X1 = vectorizer1.fit_transform(raw_docs)

print(pd.DataFrame(X1.toarray(),
  columns=vectorizer1.get_feature_names_out()).iloc[:, :10])
```

```
   afraid  basic  comes  counting  data  documents  examples  interesting  \
0       0      1      0         0     0          0         0            0
1       1      0      0         0     0          0         0            1
2       1      2      1         1     1          1         1            0

   just  larger
0     0       0
1     0       0
2     1       1
```

The counts indicate the number of times each feature (or words) were present in the document. As you might observe, longer documents tend to contain larger counts (see document 3, which has many more 1's and even a couple of 2's. Thus, instead of dealing with counts, we shall convert each row into a vector of length 1. Words that appear in all documents will be weighted

down by this transformation, since these do not help to distinguish the document from others. This transformation is known as the TF-IDF transformation.

Instead of the raw counts, we define:

- N to be the number of documents ($N = 3$ in the little example above).
- $tf_{i,j}$ to be the frequency of term i in document j.
- df_i to be the frequency of term i across all documents.
- $w'_{i,j}$ to be:

$$w'_{i,j} = tf_{i,j} \times \left[\log \left(\frac{1+N}{1+df_i} \right) + 1 \right] \tag{4.1}$$

Then the final $w_{i,j}$ for term i in document j is the normalised version of $w'_{i,j}$ across the terms that document.

Consider the word "sentences", in document id 02 (the third document).

- $N = 3$
- $tf_{i,j} = 1$
- $df_i = 2$

Thus

$$w'_{ij} = 1 \times \log((1+3)/(1+2)) = 1.287 \tag{4.2}$$

```
vectorizer2 = TfidfVectorizer(stop_words='english', norm=None)
X2 = vectorizer2.fit_transform(raw_docs)

print(pd.DataFrame(X2.A,
  columns=list(vectorizer2.get_feature_names_out())).iloc[:, :10].round(3))
```

```
   afraid  basic  comes  counting   data  documents  examples  interesting  \
0   0.000  1.288  0.000     0.000  0.000      0.000     0.000        0.000
1   1.288  0.000  0.000     0.000  0.000      0.000     0.000        1.693
2   1.288  2.575  1.693     1.693  1.693      1.693     1.693        0.000

    just  larger
0  0.000   0.000
1  0.000   0.000
2  1.693   1.693
```

The final step normalises the weights across each document, so now the weight for sentences in document 00 is higher than the weight in document 02 since document 00 is shorter.

```
vectorizer3 = TfidfVectorizer(stop_words='english')
X3 = vectorizer3.fit_transform(raw_docs)

print(pd.DataFrame(X3.A,
  columns=list(vectorizer3.get_feature_names_out())).iloc[:, :10].round(3))
```

```
   afraid  basic  comes  counting   data  documents  examples  interesting  \
0   0.000  0.577  0.000     0.000  0.000      0.000     0.000        0.000
1   0.474  0.000  0.000     0.000  0.000      0.000     0.000        0.623
2   0.170  0.340  0.223     0.223  0.223      0.223     0.223        0.000

    just  larger
0  0.000   0.000
1  0.000   0.000
2  0.223   0.223
```

The above matrix is known as a **document-term matrix**, since the columns are defined by terms, and each row is a document. At this point, we can use each **row** as a vector representation of each document. If necessary, for this corpus, we could even represent each term using its corresponding **column**.

Note that some books/software use a slightly different convention - they may work with the *term-document* matrix. However, the idea is the same. Take a look at the following term-document matrix, assembled from the complete works of Shakespeare:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 4.1: Jurafsky and Martin (2025)

If we intend to represent each document as a numeric vector, the columns, highlighted by the red boxes, would be a natural choice. Suppose we only focus on the coordinates corresponding to the words `battle` and `fool`. Then a visualisation of the documents would look like this:

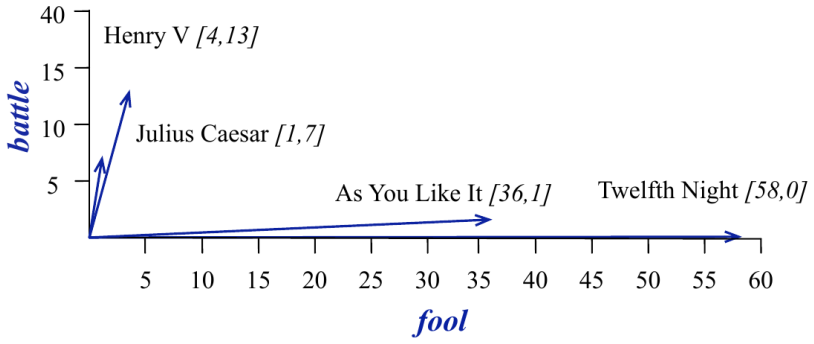


Figure 4.2: Jurafsky and Martin (2025)

Visually, it is easy to tell that "Henry V" and "Julius Caesar" are similar (they point in the same direction) as opposed to "As You Like It" and "Twelfth Night". But it is also easy to see *why* - the former two contain similar high counts of `battle` compared to the latter two, which are comedies.

Tf-idf are a normalised version of the above raw counts; they provide a numerical representation of documents, adjusting for document length and words that are common across all documents in a corpus.

4.5.2 Cosine similarity

In order to quantify the similarity (or nearness) of vector representations in NLP, the common method used is cosine similarity. Suppose that we have a vector representation of two documents $\mathbf{v}$ and $\mathbf{w}$. If the vocabulary size is N, then each of the vectors is of length N. Since we are dealing with counts the coordinate values of each vector will be non-negative. We use the angle θ between the vectors as a measure of their similarity:

$$\cos\theta = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

Geometrically, cosine similarity measures the size of the angle between vectors:

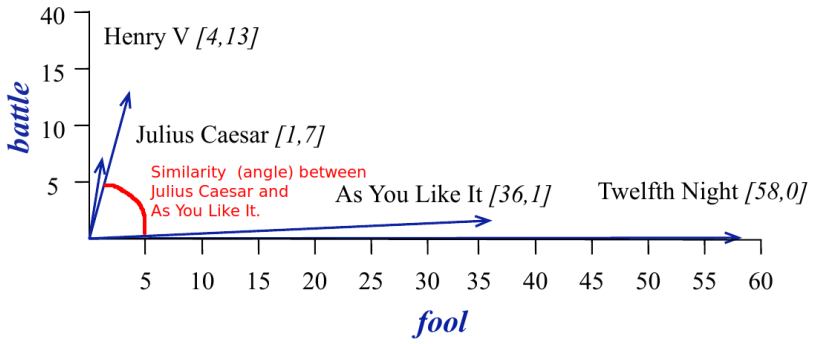


Figure 4.3: Jurafsky and Martin (2025)

4.5.3 Dense Embeddings

One of the drawbacks of sparse vectors is that they are very long (the length of the vocabulary), and most entries in the vector will be 0. As a result, researchers worked on methods that would pack the information in the vectors into shorter ones. Instead of working on representations of the documents, the methods aimed to create representations of each token (or word) in the vocabulary. These are referred to as *embeddings*.

Here, we shall discuss word2vec (Mikolov et al. (2013)), but take note that there are others. GLoVe (Pennington, Socher, and Manning (2014)) was invented soon after, but the most common embeddings used today arise from Deep Learning models. The most widely used version is BERT (see the video references below, as well as Devlin et al. (2019)).

The approach in word2vec deviates considerably from tf-idf, in that the goal is to obtain a numeric representation of a word, *in the context of it's surrounding words*. Consider this statement:

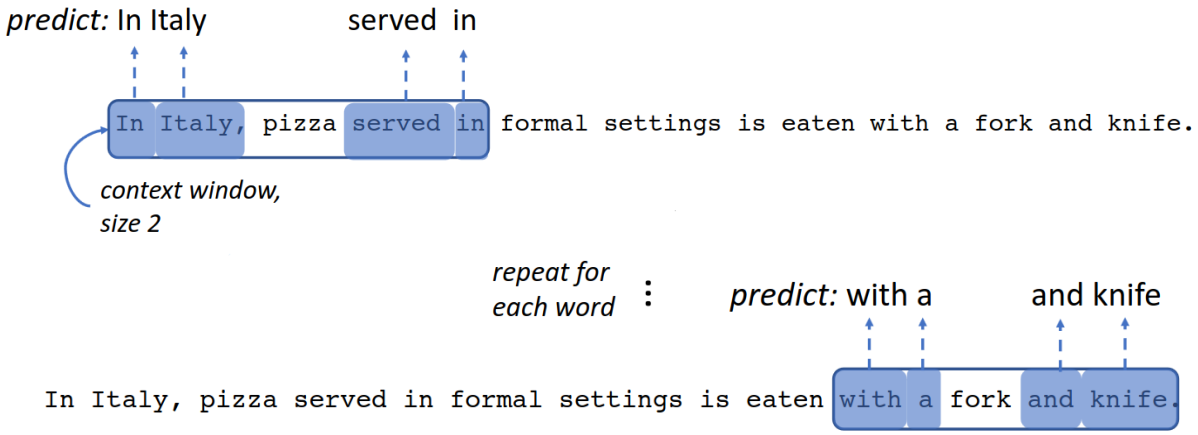
> 13% of the United States population eats pizza on any given day. Mozzarella is commonly used on pizza, with the highest quality mozzarella from Naples. In Italy, pizza served in formal > settings is eaten with a fork and knife.

The words `eats`, `served` and `mozzarella` appear close to `pizza`. Hence another word that appears in similar contexts, should be *similar* to `pizza`. Examples could be certain baked dishes or even `salad`.

To achieve such a representation, word2vec runs a self-supervised algorithm, with two tasks:

1. **Primary task:** To "learn" a numeric vector that represents each word.
2. **Pretext task (stepping stone):** To train a classifier that, when given a word w, predicts nearby context words c.

Self-supervised algorithms differ from supervised algorithms in that there are no labels that need to be created. The pre-text task trains a model to perform predictions, based on a sliding window context:



Starting with an initial random vector for each word, the algorithm updates the vectors as it proceeds through the corpus, finally ending up with an embedding for each word that reflects its semantic value, based on neighbouring words.

In NLP, the quality of an embedding can be evaluated using an analogy task:

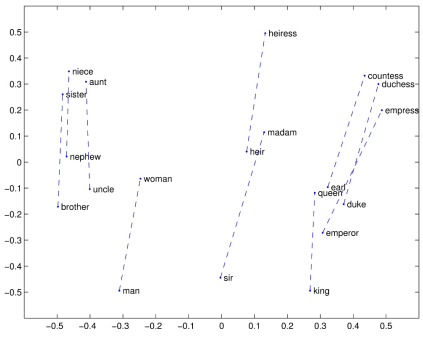
> Given X, Y and Z, find W such that W is related to to Z in the same way that X is related to Y.

For instance, if we are given the pair `man`:`king`, and the word `woman`, then the embedding should return `queen`, since `woman`:`queen` in the same way that `man` is related to `king`. Geometrically, the answer to the analogy is obtained by adding (king - man) to woman. The nearest embedding to the result, is returned as the answer.

On the left are examples of the types of analogy pairs that word2vec is able to solve, while on the right, we have visualisations of GLoVe.

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

(a) word2vec



(a) GloVE

Here's how we can use `gensim` code to conduct the analogy task.

```
# load pre-trained word-vectors from gensim-data
word_vectors = api.load("glove-wiki-gigaword-100")

# Check the "most similar words", using the default "cosine similarity" measure.
result = word_vectors.most_similar(positive=['woman', 'king'], negative=['man'])
most_similar_key, similarity = result[0]  # look at the first match
print(f"{most_similar_key}: {similarity:.4f}")
```

```
queen: 0.7699
```

```
print(word_vectors.doesnt_match("breakfast cereal dinner lunch".split()))
#similarity = word_vectors.similarity('woman', 'man')
```

```
cereal
```

4.6 Visualisation with t-SNE

When compared with sparse embeddings, dense embeddings are compact. However, a more important difference is that dense vectors represent the semantic meaning of the words. This means that vectors that are close to each other are similar in meaning. Let us use t-SNE to visualise the GloVe embeddings.

There are a total of 400,000 vectors in the embedding. Even with t-SNE that will be difficult to make sense of. Hence for now, we sample a set of 100 to visualise them.

```
rng1 = np.random.default_rng(1111)

nn = 1000
id = rng1.choice(len(word_vectors), size=(nn,), replace=False)

X = np.zeros((nn, 100))
for ii in np.arange(nn):
    #X[ii,] = glove_vectors.get_vector(id[ii])
    X[ii,] = word_vectors.get_vector(ii)
labels = pd.Series([word_vectors.index_to_key[x] for x in np.arange(nn)])

tsne1 = manifold.TSNE(n_components=2, init="random", perplexity=10, metric='cosine', verbose=0
X_transformed2 = tsne1.fit_transform(X)
```

You should get the same plot as us since we have set the same seed at the start of the cell, and when we initialise the transformer. Explore the resulting plot - notice how months of the year appear close together at the bottom left. Around the left as well, the calendar years appear as a group.

(The figure below only appears in the html version of the text)

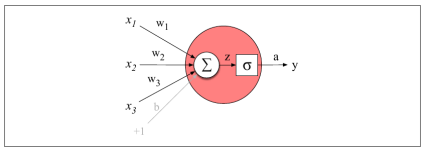
> **Note**
>
> Would we be able to make such a plot using tf-idf? Why or why not?

4.7 Neural Language Models

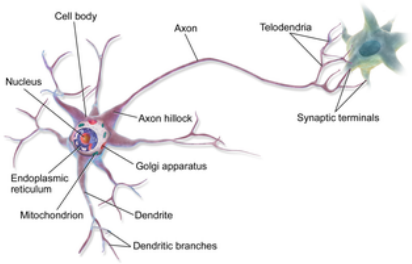
Language is complex. It is incredible how we can understand such long paragraphs of texts with such ease. We somehow seem to have learnt complicated sets of grammar and syntax just by listening to others speak. To get a machine to learn language has not been easy. It is only recently that Large Language Models such as chatGPT have demonstrated that it is possible for machines to converse with humans just as we do to one another.

Neural Models (or deep learning models) have been the key to this. In this subsection, we provide a very brief overview of their characteristics that allow them to achieve impressive performance on a range of language-related tasks.

The basic unit of a neural model is the neural unit (on the left). It consists of weights and a non-linear activation function. Given an input vector, the weights are multiplied by the input vector, summed and then fed through the activation function to generate an output.



(a) Jurafsky and Martin (2025)



(a) Neuron figure from https://en.wikipedia.org/wiki/Neuron

Neural models are made up of many neural units, organised into layers. The first neural models were Feed-Forward Networks. Due to the virtue of being able to incorporate many parameters, and due to semi-supervised learning, they were already a huge improvement over earlier models. Here is a simple set up, with one hidden layer for training a language model (used to predict the next word). It can also be used to learn embeddings.

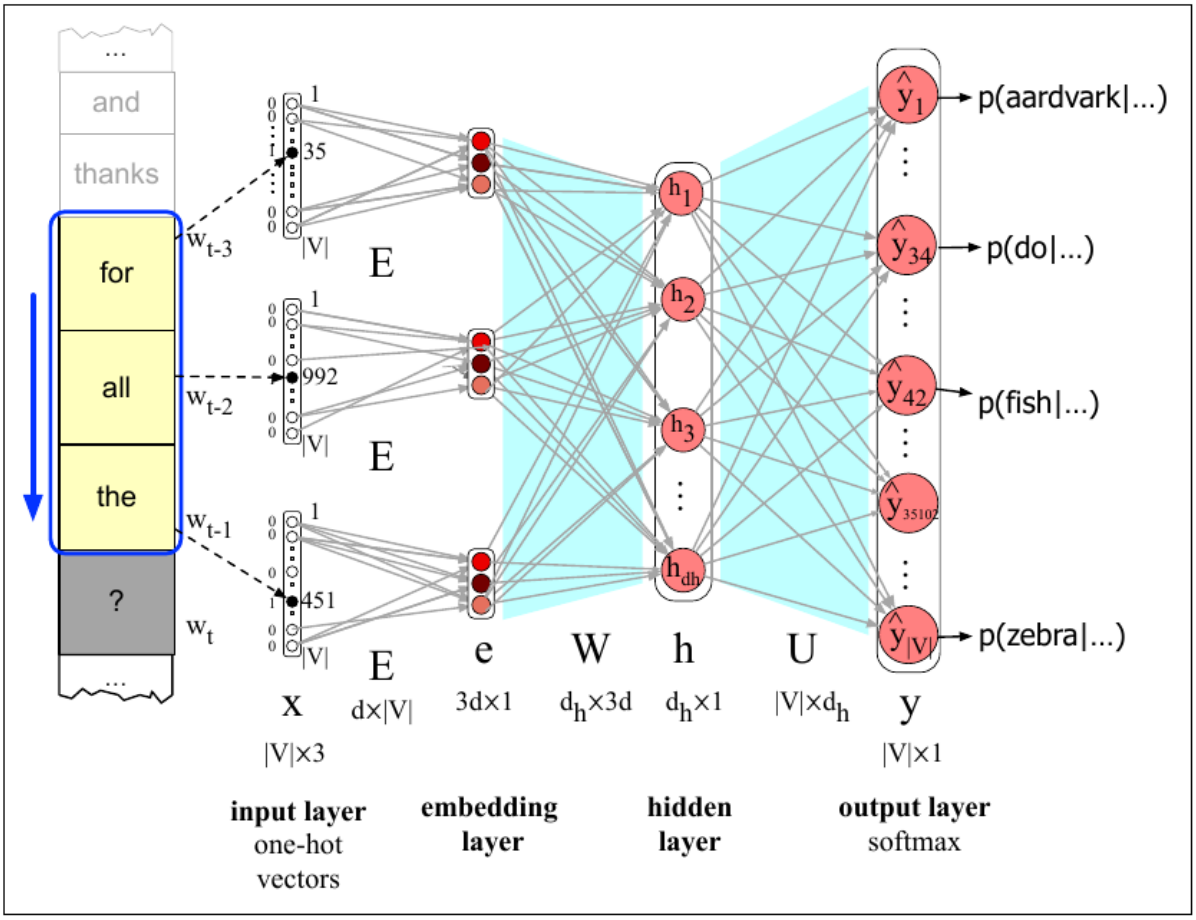


Figure 4.8: Jurafsky and Martin (2025), FFN

The next evolution in neural models was the ability to incorporate words in the recent history. For humans, this comes naturally. For instance, we know that this is grammatically correct:

> The flights the airline was cancelling **were** full.

For neural models to have this ability, it was necessary to incorporate the hidden layers from recent words when processing the current word. Recurrent Neural Networks (RNNs) and Long-Short Term Memory (LSTM) networks had these features, but they were very slow to train. The major breakthrough came with the invention of the transformer architecture. The self-attention layer of these networks gave a word access to *all* preceding words in the training window, instead of just one. Most importantly. the training of these networks could be parallelised!

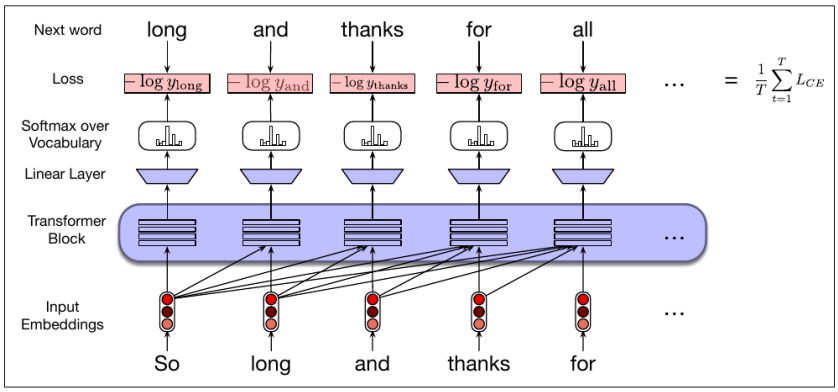


Figure 4.9: Jurafsky and Martin (2025)

Here are some examples where transformers excel:

> The keys to the cabinet *are* on the table.
>
> The chicken crossed the road because *it* wanted to get to the other side.
>
> I walked along the pond, and noticed that one of the trees along the *bank* had fallen into the water after the storm.

In the final sentence, the word `bank` has two meanings - how will a model know to decide the correct one? With transformers, because the full context of a word is captured along with it, it is possible to perform this disambiguation.

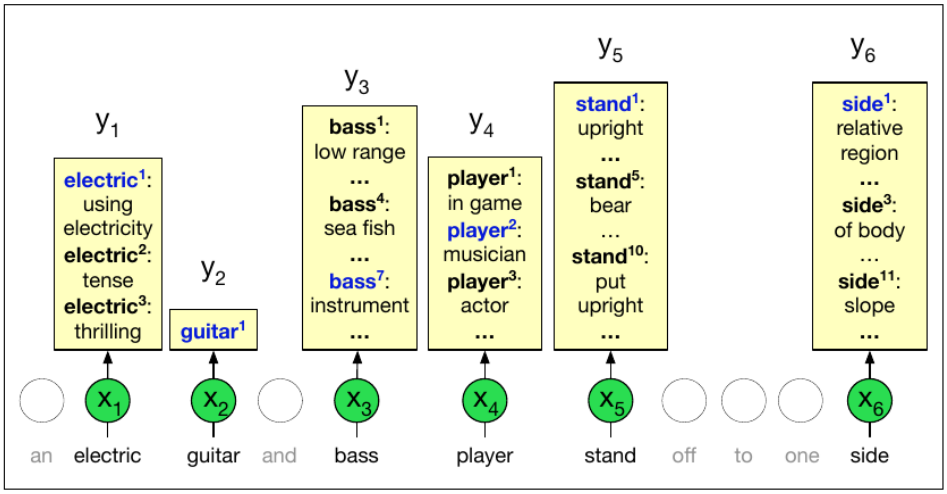


Figure 4.10: Jurafsky and Martin (2025)

4.8 Applications

Hugging Face has spent a considerable effort to make Neural Language Models accessible and available to all with minimal coding. For starters, they have ensured that all their models are described in a standardised manner with model cards. Here is an example of a model card for BERT.

Moreover, they have developed easy to use pipelines. For NLP, the following tasks have mature pipelines:

- feature-extraction (obtaining the embedding of a text)
- ner
- question-answering
- sentiment-analysis
- summarization
- text-generation
- translation, and
- zero-shot-classification.

4.8.1 Sentiment Analysis

In this subsection, we shall utilise one of their sentiment analysis models on the wine reviews dataset. This is a transformer-based neural language model (BERT) that has been fine-tuned with data labelled with sentiments. All we have to do is feed in the sentence, and we will obtain a confidence score, and a sentiment label.

```
classifier = pipeline("sentiment-analysis",
  model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")

classifier(["I love this course!", "I absolutely detest this course."])
```

```
Device set to use cpu
```

```
[{'label': 'POSITIVE', 'score': 0.9998835325241089},
 {'label': 'NEGATIVE', 'score': 0.9973570704460144}]
```

Example 4.2 (Example: Wine Reviews Dataset)**.** The number of reviews we have is close to 120,000. Hence, computing the sentiments for each and every one will take a long time. Instead, we shall compute the sentiments for a sample (of size 20, where possible) from each variety of wine.

The following snippet samples 20 reviews from each wine type.

```
tmp_df = pd.DataFrame(columns= wine_reviews.columns)
# w = widgets.IntProgress(
#     value=0,
#     min=0,
#     max=len(wine_reviews.variety.unique()),
#     description='Progress: ',
#     bar_style='', # 'success', 'info', 'warning', 'danger' or ''
#     style={'bar_color': 'lightblue'},
#     orientation='horizontal'
# )
# display(w)

for x,vv in wine_reviews.groupby(wine_reviews.variety):
```

```
    grp_len = vv.shape[0]
    if(grp_len >= 20):
        vv = vv.sample(n=20, random_state=99)
    tmp_df = pd.concat([tmp_df, vv], ignore_index=True)
    # w.value += 1

review_list = list(tmp_df.description)
```

```
/tmp/ipykernel_12830/2593812303.py:17: FutureWarning:

The behavior of DataFrame concatenation with empty or all-NA entries is deprecated. In a futur
```

The next snippet computes the sentiment scores for those sampled reviews.

```
tmp_df['score'] = 0.00
tmp_df['label'] = ''
# w = widgets.IntProgress(
#     value=0,
#     min=0,
#     max=tmp_df.shape[0],
#     description='Progress: ',
#     bar_style='', # 'success', 'info', 'warning', 'danger' or ''
#     style={'bar_color': 'lightblue'},
#     orientation='horizontal'
# )
# display(w)
for i,rr in enumerate(review_list):
    tmp = classifier(rr)[0]
    tmp_df.loc[i, 'score'] = tmp['score']
    tmp_df.loc[i, 'label'] = tmp['label']
    #w.value = i
```

```
sent_counts = pd.crosstab(tmp_df.variety, tmp_df.label,margins=True)
sent_counts['proportion'] = sent_counts.POSITIVE/sent_counts.All
```

```
sent_counts.head()
```

label variety	NEGATIVE	POSITIVE	All	proportion
Abouriou	0	3	3	1.000000
Agiorgitiko	0	20	20	1.000000
Aglianico	0	20	20	1.000000
Aidani	0	1	1	1.000000
Airen	1	2	3	0.666667

These are the reviews for one of the varieties that had a proportion of positive reviews close to 50%.

```
for x in wine_reviews[wine_reviews.variety == 'Tempranillo Blanco'].description.values:
    pp.pprint(x)
```

```
("Gold in color and lightly oxidized on the nose, and it's still young. Smells "
 'heavy and creamy, like hay. Feels flat, with pickled flavors and mealy apple '
 'on the finish. Runs plump, sweet and seems like an imposter for Chardonnay.')
('Oily, stalky, bready aromas are a bit tired. This has a chunky feel offset '
 'by citric acidity. Briny, salty flavors of citrus fruits and lees are '
 "lasting. For varietal Tempranillo Blanco, this isn't bad.")
('Maderized in color, this wine has a yeasty, creamy nose with baked '
 "white-fruit aromas and caramel. It's OK in feel, with pickled, mildly briny "
 'flavors of apple and apricot. The finish is showing some oxidization, '
 'leading to a chunky, fleshy feel.')
("Waxy peach aromas seem slightly oxidized. It's round and citrusy on the "
 'palate, but in a monotone way that fades to pithy white fruits and mealy '
 "citrus. Shows some flashes of uniqueness and class; mostly it's wayward and "
 'slightly bitter.')
('Forget the high price on this Tempranillo Blanco. Looking at the wine alone, '
 "it's briny and stalky on the nose, with wiry lemon-like acids that push sour "
 "orange flavors. Overall it's monotone, briny and citrusy.")
("A maderized color is apropos for the wine's fully mature, nutty nose. This "
 'is big and cidery feeling, with apple and orange flavors. A finish of '
 'vanilla, nuttiness and oxidation matches the color and aromas of this '
 'interesting but midlevel Tempranillo Blanco.')
('Green grassy aromas are modest and watery. This feels oily, but with decent '
 'acidity. Oxidized flavors of stone fruits finish wheaty, bland and eggy.')
('Rough, stalky, yeasty aromas are all over the map. Lemony acidity renders '
 'this tight as a drum, while bitter, stalky flavors finish wheaty and bitter. '
 'This Tempranillo Blanco is barely worth a go; the pleasure factor is at base '
 'level.')
('Green aromas of herbs and tomatillo are harsh, rubbery and outweigh peach '
 'and other stone-fruit scents. This Tempranillo Blanco is plump and fair on '
 'the palate, while flavors of apple and peach are briny and finish with '
 'controlled bitterness.')
```

> **Note**
>
> Do you agree with the classifications above? What would you investigate next?

4.8.2 Information Retrieval

In the NLP context, Information Retrieval (IR) refers to the task of returning the most relevant set of documents, when given a query string. Search engines, e.g. Google, are trained to perform fast and accurate IR. Typically, a long list of documents is returned, with the most relevant one on top.

> **Note**
>
> Pause for a moment, and consider how you would assess the performance of such a search engine.

Example 4.3 (Example: Wine Reviews Dataset)**.** A simple way to perform IR is to use cosine similarity to compute how close the given query vector is to the individual documents in the corpus.

The next snippet initialises a model for retrieving similar documents.

```
dct = gensim.corpora.Dictionary(all_strings_tokenized)
bow_corpus = [dct.doc2bow(text) for text in all_strings_tokenized]
tfidf = gensim.models.TfidfModel(dictionary=dct)
```

NLP corpora are typically very large. Before we can find matching documents, we build a similarity index, so that matches are returned quicker. We try something simple at first:

Which documents/reviews are similar to the first one?

```
index = gensim.similarities.Similarity(None,
  corpus=tfidf[bow_corpus], num_features=len(dct))
sims = index[tfidf[bow_corpus[0]]]
```

These are the most similar reviews to *review id 0*. Of course, the first review itself is there! Let's retrieve and print all the reviews similar to the first one.

```
#np.argsort(-sims)[:10]
for x in np.argsort(-sims)[:5]:
    pp.pprint(all_review_strings[x])
```

```
('Aromas include tropical fruit, broom, brimstone and dried herb. The palate '
 "isn't overly expressive, offering unripened apple, citrus and dried sage "
 'alongside brisk acidity.')
("The nose isn't very expressive but reveals white flower and tropical fruit. "
 'The simple palate delivers pineapple and lemon zest alongside brisk acidity.')
("The nose isn't very expressive but the palate eventually reveals raw red "
 'berry, espresso, brimstone and grilled rosemary alongside astringent and '
 'rather drying tannins.')
('This opens with aromas of pressed acacia flowers, ripe stone fruits and '
 "dried sage. The palate isn't overly sweet, offering dried apricot, "
 'wildflower honey and toasted almond notes.')
('Subdued aromas of Spanish broom and brimstone float from the glass. The '
 'vertical palate offers yellow apple, citrus zest and mineral alongside crisp '
 'acidity.')
```

Now we try a new query of our own: "acidic chardonnay". First we preprocess it, like we did the original documents.

```
q1 = [wn.lemmatize(x) for x in preprocess_string('acidic white chardonnay', CUSTOM_FILTER)]
sims = index[tfidf[dct.doc2bow(q1)]]
```

Now we print the top 5 most similar reviews to our query.

```
q1_results = np.argsort(-sims)[:10]
#q1_results
#pp.pprint(wine_reviews.description.values[q1_results])
for x in q1_results[:5]:
    pp.pprint(all_review_strings[x])
```

```
('A standard Chardonnay, dry and nicely acidic, with citrus, pear, vanilla, '
 'lees and oak flavors.')
('Dry and acidic, this Chardonnay has a herbaceous earthiness, plus flavors of '
 'orange and pear.')
'This is thin and acidic, with flavors of sour cherry candy and spice.'
'This is acidic and sweet, with a medicinal taste.'
('Pungent up front, with green herb, white pepper and citrus aromas, this is '
 'zesty and acidic on the palate, with a monotonous lemon flavor on the '
 'finish. It turns more tart and acidic as it airs.')
```

> **Note**
>
> Try your favourite tastes, see if you discover a wine you like/dislike

4.8.3 Topic Modeling

The LDA (Latent Dirichlet Allocation) model assumes the following intuitive generative process for the documents:

1. There is a set of K topics that the documents come from. Each document contains words from several topics. There is a probability mass function on the topics for each document.
2. For each topic, there is a probability mass function for the distribution of words in that topic.

At the end of LDA topic modeling, we will be able to tell, for a particular (new or old) document: the weight combination of the topics for that document. For each topic, we would be able to tell the terms that are salient.

LDA only gives us the probabilistic weights - we have to interpret them ourselves. Suppose we decide to split the corpus into 10 topics. Let us investigate what these topics consist of.

```
lda1 = gensim.models.LdaModel(corpus= bow_corpus, num_topics=10, id2word=dct)
reviews_vis_data = gensimvis.prepare(lda1, bow_corpus, dct)

pp.pprint(lda1.show_topics())
```

```
[   (   0,
        '0.024*"flavors" + 0.022*"wine" + 0.019*"cabernet" + 0.017*"cherry" + '
        '0.017*"fruit" + 0.013*"blend" + 0.013*"black" + 0.012*"oak" + '
        '0.012*"tannins" + 0.010*"sauvignon"'),
    (   1,
        '0.070*"wine" + 0.037*"tannins" + 0.030*"fruit" + 0.024*"drink" + '
        '0.023*"black" + 0.022*"ripe" + 0.020*"rich" + 0.019*"firm" + '
        '0.018*"years" + 0.018*"fruits"'),
    (   2,
        '0.039*"flavors" + 0.028*"pinot" + 0.026*"acidity" + 0.023*"wine" + '
        '0.022*"chardonnay" + 0.018*"vanilla" + 0.017*"oak" + 0.016*"sweet" + '
        '0.015*"rich" + 0.015*"noir"'),
    (   3,
        '0.043*"wine" + 0.025*"fruit" + 0.015*"bright" + 0.012*"expression" + '
        '0.012*"vineyard" + 0.011*"grapes" + 0.010*"aromas" + 0.010*"almond" + '
        '0.009*"white" + 0.009*"shows"'),
    (   4,
        '0.035*"palate" + 0.034*"aromas" + 0.030*"cherry" + 0.028*"black" + '
        '0.023*"tannins" + 0.018*"red" + 0.017*"plum" + 0.017*"spice" + '
        '0.015*"berry" + 0.014*"nose"'),
    (   5,
        '0.032*"palate" + 0.026*"finish" + 0.024*"apple" + 0.022*"aromas" + '
        '0.021*"flavors" + 0.019*"lemon" + 0.019*"citrus" + 0.018*"white" + '
        '0.018*"acidity" + 0.016*"peach"'),
    (   6,
        '0.075*"wine" + 0.041*"acidity" + 0.034*"drink" + 0.024*"ripe" + '
        '0.021*"fruity" + 0.020*"crisp" + 0.019*"flavors" + 0.019*"fruits" + '
        '0.018*"texture" + 0.017*"red"'),
    (   7,
        '0.021*"spice" + 0.018*"wine" + 0.013*"pair" + 0.012*"succulent" + '
        '0.010*"meat" + 0.010*"elegant" + 0.009*"fruit" + 0.008*"brunello" + '
        '0.006*"barolo" + 0.006*"closed"'),
    (   8,
        '0.054*"flavors" + 0.037*"finish" + 0.031*"aromas" + 0.017*"fruit" + '
        '0.017*"berry" + 0.016*"oak" + 0.013*"feels" + 0.013*"herbal" + '
        '0.012*"like" + 0.012*"sweet"'),
    (   9,
        '0.029*"light" + 0.020*"dry" + 0.019*"fruit" + 0.017*"wine" + '
        '0.015*"fresh" + 0.015*"flavors" + 0.014*"nose" + 0.014*"palate" + '
        '0.013*"easy" + 0.012*"tart"')]
```

The output provides the most common terms that define each topic (remember: each topic is defined as a *probability distribution over the vocabulary*).

> **Note**
>
> What name would you give each topic?

We can also find out which *topics* a particular document is distributed over. For instance, the output below shows that words in document 0 are predominantly drawn from topics 2 and 5.

```
lda1.get_document_topics(bow_corpus[0])
```

```
[(4, 0.2845675), (5, 0.67732245)]
```

```
pp.pprint(all_review_strings[0])
```

```
('Aromas include tropical fruit, broom, brimstone and dried herb. The palate '
 "isn't overly expressive, offering unripened apple, citrus and dried sage "
 'alongside brisk acidity.')
```

A delightful visualisation from pyLDAvis allows us to understand the "distance" between topics, and the frequent words from each topic easily.

```
pyLDAvis.display(reviews_vis_data)
```

4.9 Interpretation of Neural Models

Neural models have achieved impressive performance on a number of language-related tasks. However, one criticism of them is that they are "black-box" models; we do not fully grasp how they work. This can lead to a mistrust of such models, with good reason. If we do not fully know how these models work, we would not know when they are might fail, or we might not know the reason when they do fail (or make an incorrect prediction). For this reason, a huge amount of research effort is currently directed towards understanding and interpreting neural models.

One approach is to identify which examples in the training set were most influential for predictions regarding particular test instances. For incorrect predictions, this could give us intuition on why the model is failing, and guide us to ways to fix it. Here is an example where it was possible to pinpoint why a model yielded incorrect sentiment prediction.

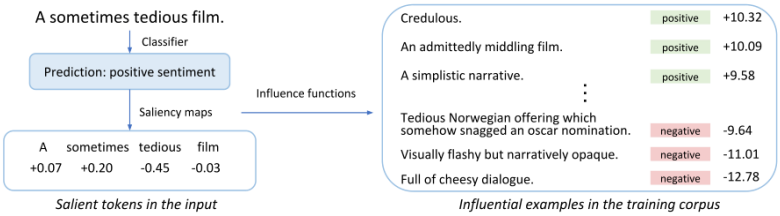


Figure 4.11: Han, Wallace, and Tsvetkov (2020)

Another approach is to identify which parts of the test sentence itself were important to the eventual prediction. Imagine perturbing the test sentence in some ways, and studying how the prediction changed. In one study of a Question-Answering model, the question was modified by dropping the least important word, until the question was answered incorrectly. In this case, the study revealed something pathological about the model:

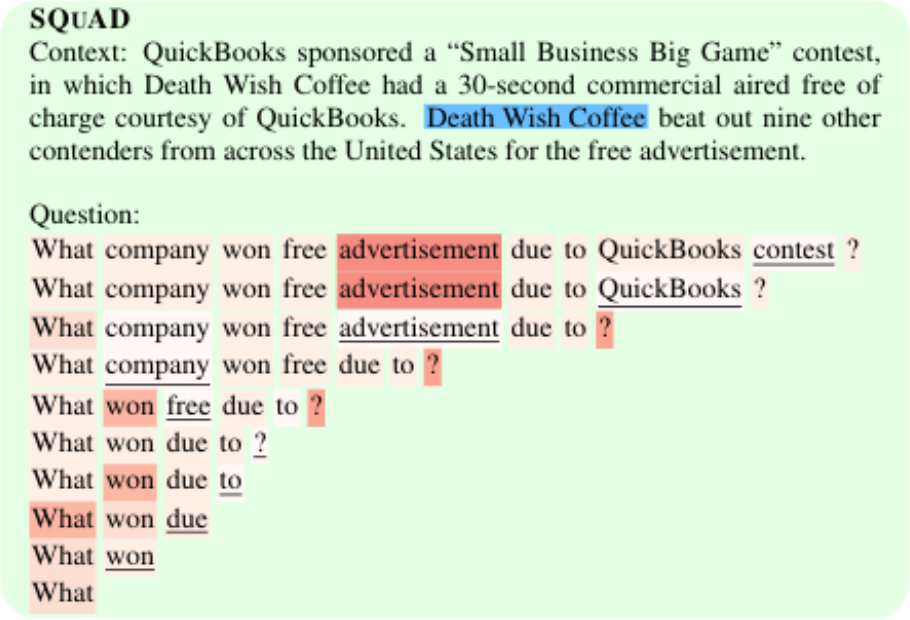


Figure 4.12: Sun et al. (2021)

For transformers in particular, a great deal of study has focused on the weights that the attention layers pick up. By relating these to linguistic information, researchers try to infer the precise language-related information that models retain. For instance, it has been found that BERT learns parts of speech. It is also able to identify the dependencies between words.

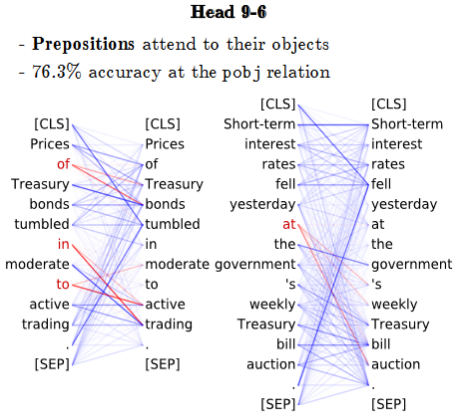


Figure 4.13: Clark et al. (2019)

4.10 References

4.10.1 Video explainers

1. Transformer models and BERT: A very good video from Google Cloud Tech on current neural models (11:37)
2. Introduction to RNN
3. Introduction to BERT

4.10.2 Website References

1. Hugging Face course on transformers
2. Gensim documentation: Contains tutorials as well.
3. Using sklearn to perform LDA: We can also use scikit-learn to perform LDA.
4. Visualising LDA: Contains sample notebooks for the visualisation.

5 Linear Regression

5.1 Introduction

Regression analysis is a technique for investigating and modeling the relationship between variables like X and Y. Here are some examples:

1. Within a country, we may wish to use per capita income (X) to estimate the life expectancy (Y) of residents.
2. We may wish to use the size of a crab claw (X) to estimate the closing force that it can exert (Y).
3. We may wish to use the height of a person (X) to estimate their weight (Y).

In all the above cases, we refer to X as the *explanatory* or *independent* variable. It is also sometimes referred to as a *predictor*. Y is referred to as the *response* or *dependent* variable. In this topic, we shall first introduce the case of simple linear regression, where we model the Y on a single X. In later sections, we shall model the Y on multiple X's. This latter technique is referred to as multiple linear regression.

Regression models are used for two primary purposes:

1. To understand how certain explanatory variables affect the response variable. This aim is typically known as *estimation*, since the primary focus is on estimating the unknown parameters of the model.
2. To predict the response variable for new values of the explanatory variables. This is referred to as *prediction*.

In this topic, we shall focus on the estimation aim, since prediction models require a paradigm of their own, and are best learnt alongside a larger suite of models e.g. decision trees, support vector machines, etc. We shall cover prediction in the topic of supervised learning.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import folium
import geopandas
from itables import show

import statsmodels.api as sm
from statsmodels.formula.api import ols

from scipy import stats
```

Example 5.1 (Example: Taiwan Real Estate)**.** For this tutorial, we shall work with a data set from the UCI machine learning repository. It contains real estate prices in the Xindian district of Taiwan. Our goal is to answer the following question:

> How well can we explain real-estate prices in Taiwan?

Here is a brief description of the columns in the dataset:

- `trans_date`: The date of the transaction. As you can see, this has been coded to be a numerical value, so 2013.5 refers to June 2013.
- `house_age`: Age of the house in years.
- `dist_MRT`: Distance the the nearest MRT (in metres)
- `num_stores`: Number of convenience stores within walking distance
- `lat`, `long`: Latitude and longitude
- `price`: House price per unit area (10,000 New Taiwan Dollars per Ping, which is about $3.3m^2$)
- `X`, `Y`, `Xs`, `Ys`: Projected coordinates

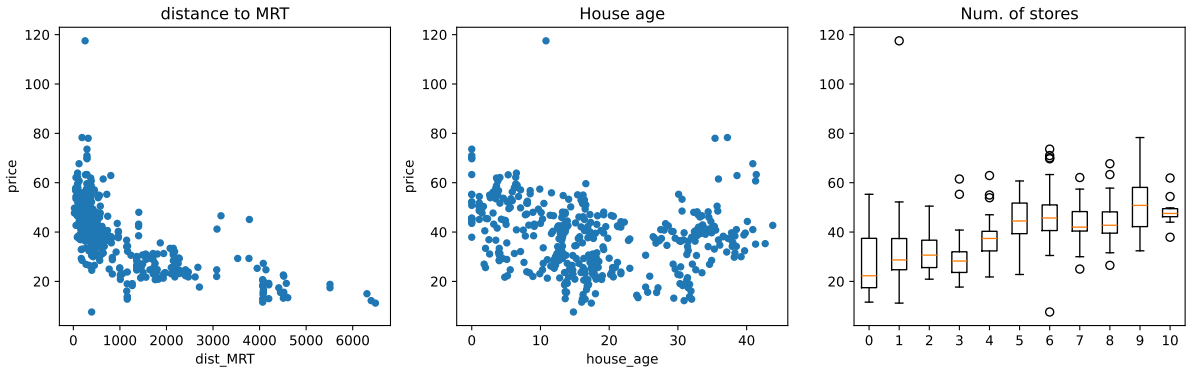
```
re2 = pd.read_csv("data/taiwan_dataset.csv")
re2.head()
```

	id	trans_date	house_age	dist_MRT	num_stores	lat	long	price	X	Y
0	1	2012.916667	32.0	84.87882	10	24.98298	121.54024	37.9	506501.554580	2.
1	2	2012.916667	19.5	306.59470	9	24.98034	121.53951	42.2	506433.292007	2.
2	3	2013.583333	13.3	561.98450	5	24.98746	121.54391	47.3	506862.973688	2.
3	4	2013.500000	13.3	561.98450	5	24.98746	121.54391	54.8	506862.973688	2.
4	5	2012.833333	5.0	390.56840	5	24.97937	121.54245	43.1	506732.307872	2.

The following interactive map is available on the HTML version.

Let us first explore the dataset with Python. How do this plots aid in your understanding of the dataset?

```
plt.figure(figsize=(15,4));
ax=plt.subplot(131)
re2.plot(x='dist_MRT', y='price', kind='scatter', ax=ax, title='distance to MRT')

ax=plt.subplot(132)
re2.plot(x='house_age', y='price', kind='scatter', ax=ax, title='House age')

ax= plt.subplot(133)
z = re2.num_stores.unique()
z.sort()
tmp2 = np.array([re2.price[re2.num_stores == x].to_numpy() for x in z], dtype=object)
ax.boxplot(tmp2, tick_labels=z); ax.set_title('Num. of stores');
```



5.2 Simple Linear Regression

5.2.1 Formal Set-up

The simple linear regression model is applicable when we have observations (X_i, Y_i) for n individuals. For now, let's assume both the X and Y variables are quantitative.

The simple linear regression model is given by

$$Y_i = \beta_0 + \beta_1 X_i + e_i$$

where

- β_0 is intercept term,
- β_1 is the slope, and
- e_i is an error term, specific to each individual in the dataset.

β_0 and β_1 are unknown constants that need to be estimated from the data. There is an implicit assumption in the formulation of the model that there is a linear relationship between Y_i and X_i. In terms of distributions, we assume that the e_i are i.i.d Normal.

$$e_i \sim N(0, \sigma^2), \ i = 1 \ldots, n$$

The constant variance assumption is also referred to as homoscedascity (homo-skee-das-city). The validity of the above assumptions will have to be checked after the model is fitted. All in all, the assumptions imply that:

1. $E(Y_i|X_i) = \beta_0 + \beta_1 X_i$, for $i = 1, \ldots, n$.
2. $Var(Y_i|X_i) = Var(e_i) = \sigma^2$, for $i = 1, \ldots, n$.
3. The Y_i are independent.
4. The Y_i's are Normally distributed.

5.2.2 Estimation

Before deploying or using the model, we need to estimate optimal values to use for the unknown β_0 and β_1. We shall introduce the method of Ordinary Least Squares (OLS) for the estimation. Let us define the *error Sum of Squares* to be

$$SS_E = S(\beta_0, \beta_1) = \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i)^2$$

Then the OLS estimates of β_0 and β_1 are given by

$$\arg\min_{\beta_0, \beta_1} \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i)^2$$

The minimisation above can be carried out analytically, by taking partial derivative with respect to the two parameters and setting them to 0.

$$\begin{aligned} \frac{\partial S}{\partial \beta_0} &= -2 \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i) = 0 \\ \frac{\partial S}{\partial \beta_1} &= -2 \sum_{i=1}^{n} X_i (Y_i - \beta_0 - \beta_1 X_i) = 0 \end{aligned}$$

Solving and simplifying, we arrive at the following:

$$\begin{aligned} \hat{\beta}_1 &= \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^{n} (X_i - \bar{X})^2} \\ \hat{\beta}_0 &= \bar{Y} - \hat{\beta}_0 \bar{X} \end{aligned}$$

where $\bar{Y} = (1/n) \sum Y_i$ and $\bar{X} = (1/n) \sum X_i$.

If we define the following sums:

$$\begin{aligned} S_{XY} &= \sum_{i=1}^{n} X_i Y_i - \frac{(\sum_{i=1}^{n} X_i)(\sum_{i=1}^{n} Y_i)}{n} \\ S_{XX} &= \sum_{i=1}^{n} X_i^2 - \frac{(\sum_{i=1}^{n} X_i)^2}{n} \end{aligned}$$

then a form convenient for computation of $\hat{\beta}_1$ is

$$\hat{\beta}_1 = \frac{S_{XY}}{S_{XX}}$$

Once we have the estimates, we can use the estimated model to compute *fitted* values for each observation, corresponding to our best guess of the mean of the distributions from which the observations arose:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_i, \quad i = 1, \ldots, n$$

As always, we can form residuals as the deviations from fitted values.

$$r_i = Y_i - \hat{Y}_i$$

Residuals are our best guess at the unobserved error terms e_i. Squaring the residuals and summing over all observations, we can arrive at the following decomposition, which is very similar to the one in the ANOVA model:

$$\underbrace{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}_{SS_T} = \underbrace{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}_{SS_{Res}} + \underbrace{\sum_{i=1}^{n}(\hat{Y}_i - \bar{Y})^2}_{SS_{Reg}}$$

where

- SS_T is known as the total sum of squares.
- SS_{Res} is known as the residual sum of squares.
- SS_{Reg} is known as the regression sum of squares.

In our model, recall that we had assumed equal variance for all our observations. We can estimate σ^2 with

$$\hat{\sigma^2} = \frac{SS_{Res}}{n-2}$$

Our distributional assumptions lead to the following for our estimates $\hat{\beta}_0$ and $\hat{\beta}_1$:

$$\hat{\beta}_0 \sim N(\beta_0,\, \sigma^2(1/n + \bar{X}^2/S_{XX})) \tag{5.1}$$
$$\hat{\beta}_1 \sim N(\beta_1,\, \sigma^2/S_{XX}) \tag{5.2}$$

The above are used to construct confidence intervals for β_0 and β_1, based on t-distributions.

5.2.3 Hypothesis Test for Model Significance

The first test that we introduce here is to test if the coefficient β_1 is significantly different from 0. It is essentially a test of whether it was worthwhile to use a simple linear regression, instead of a simple mean to represent the data.

The null and alternative hypotheses are:

$$H_0 : \beta_1 = 0$$
$$H_1 : \beta_1 \neq 0$$

The test statistic is

$$F_0 = \frac{SS_{Reg}/1}{SS_{Res}/(n-2)}$$

Under the null hypothesis, $F_0 \sim F_{1,n-2}$.

It is also possible to perform this same test as a t-test, using the result earlier. The statement of the hypotheses is equivalent to the F-test. The test statistic

$$T_0 = \frac{\hat{\beta}_1}{\sqrt{\hat{\sigma}^2/S_{XX}}}$$

Under H_0, the distribution of T_0 is t_{n-2}. This t-test and the earlier F-test in this section are *identical*. It can be proved that $F_0 = T_0^2$; the obtained p-values will be identical.

5.2.4 Coefficient of Determination, R^2

The coefficient of determination R^2 is defined as

$$R^2 = 1 - \frac{SS_{Res}}{SS_T} = \frac{SS_{Reg}}{SS_T}$$

It can be interpreted as the proportion of variation in Yi, explained by the inclusion of X_i. Since $0 \leq SS_{Res} \leq SS_T$, we can easily prove that $0 \leq R^2 \leq 1$. The larger the value of R^2 is, the better the model is.

When we get to the case of multiple linear regression, take note that simply including more variables in the model will increase R^2. This is undesirable; it is preferable to have a parsimonious model that explains the response variable well.

Example 5.2 (Example: Price vs. House Age)**.** As a first model, we fit price (Y) against house age (X_1). From the plot above, we already suspect this may not be ideal, but let us use it as a starting point.

```
lm_house_age_1 = ols('price ~ house_age', data=re2).fit()
print(lm_house_age_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.044
Model:                            OLS   Adj. R-squared:                  0.042
Method:                 Least Squares   F-statistic:                     19.11
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           1.56e-05
Time:                        09:16:30   Log-Likelihood:                -1658.3
No. Observations:                 414   AIC:                             3321.
Df Residuals:                     412   BIC:                             3329.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     42.4347      1.211     35.042      0.000      40.054      44.815
house_age     -0.2515      0.058     -4.372      0.000      -0.365      -0.138
==============================================================================
Omnibus:                       48.404   Durbin-Watson:                   1.957
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              119.054
```

```
Skew:                          0.589   Prob(JB):                     1.40e-26
Kurtosis:                      5.348   Cond. No.                         39.0
==============================================================================

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

From the output, we can tell that the *estimated* model for Price (Y) against Housing age (X_1) is:

$$Y = 42.43 - 0.25X_1$$

The estimates are $\hat{\beta}_0 = 42.43$ and $\hat{\beta}_1 = -0.25$. The output includes the 95% confidence intervals for β_0 and β_1. The R^2 is 0.044, which means that means that only 4.4% of the variation in Y is explained by X. This is extremely poor, even though the p-value for the F-test is very small (0.000016).

A simple interpretation of the model is as follows:

> For every 1 year increase in house age, there is an average associated *decrease* in price of $0.25 \times 10,000$ New Taiwan Dollars.

Note that this interpretation has to be taken very cautiously, especially when there are other explanatory variables in the model.

Example 5.3 (Example: Price vs. House Age Estimated Line)**.** In linear regression, we almost always wish to use the model to understand what the mean of future observations would be. In this case, we may wish to use the model to understand how the Price changes as house age increases. This is because, based on our formulation,

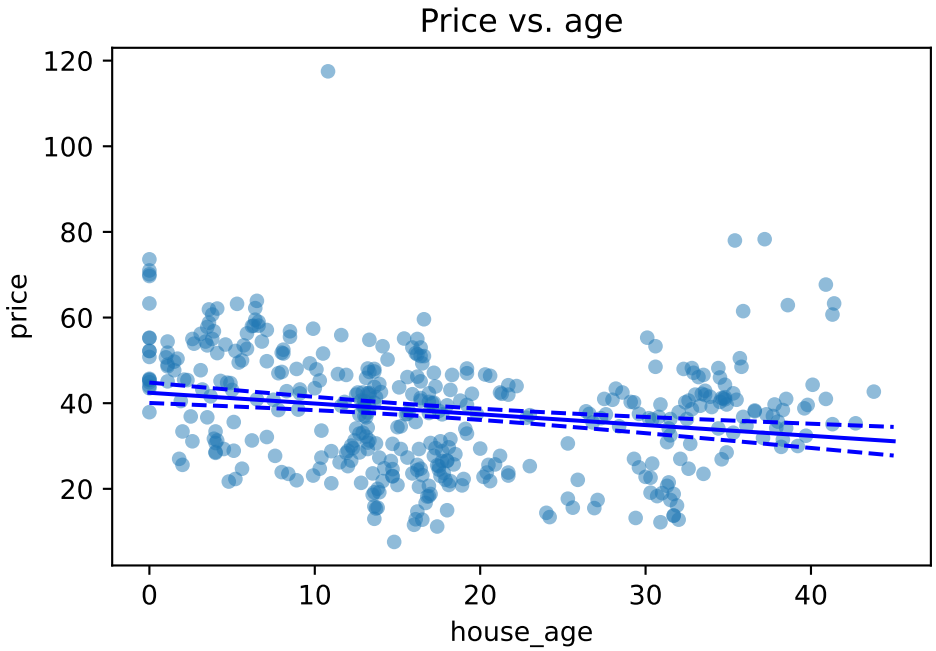
$$E(Y|X) = \beta_0 + \beta_1 X$$

After estimating the parameters, we would have:

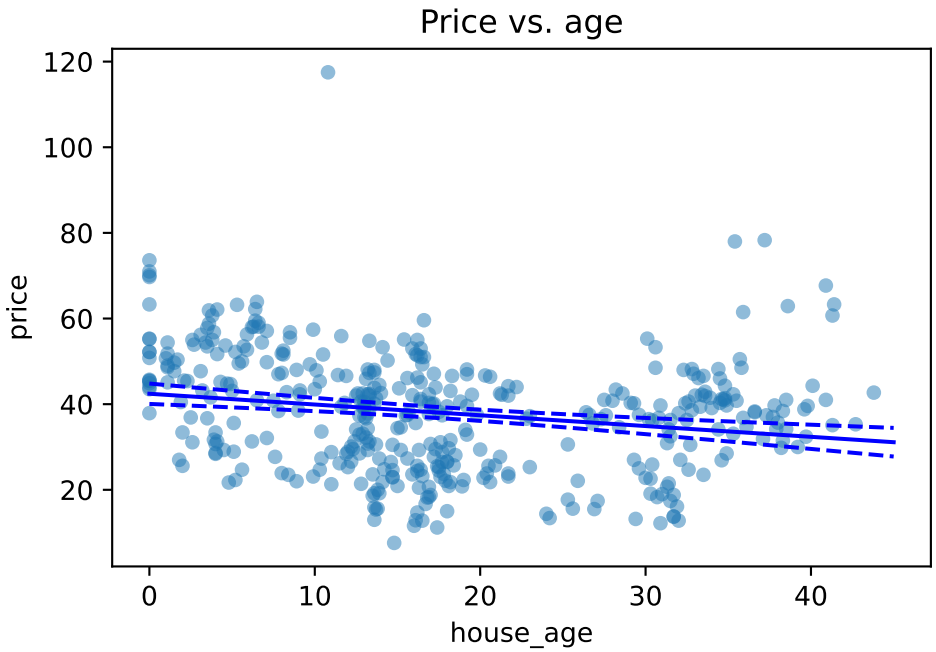
$$\widehat{E(Y|X)} = \hat{\beta}_0 + \hat{\beta}_1 X$$

Thus we can vary the values of X to study how the mean of Y changes. Here is how we can do so for the model that we have just fit.

```
new_df = sm.add_constant(pd.DataFrame({'house_age' : np.linspace(0, 45, 100)}))
predictions_out = lm_house_age_1.get_prediction(new_df)

ax = re2.plot(x='house_age', y='price', kind='scatter', alpha=0.5 )
ax.set_title('Price vs. age');
ax.plot(new_df.house_age, predictions_out.conf_int()[:, 0].reshape(-1),
        color='blue', linestyle='dashed');
ax.plot(new_df.house_age, predictions_out.conf_int()[:, 1].reshape(-1),
        color='blue', linestyle='dashed');
ax.plot(new_df.house_age, predictions_out.predicted, color='blue');
```




```
new_df = sm.add_constant(pd.DataFrame({'house_age' : np.linspace(0, 45, 100)}))
predictions_out = lm_house_age_1.get_prediction(new_df)

ax = re2.plot(x='house_age', y='price', kind='scatter', alpha=0.5 )
ax.set_title('Price vs. age');
ax.plot(new_df.house_age, predictions_out.conf_int()[:, 0].reshape(-1),
        color='blue', linestyle='dashed');
ax.plot(new_df.house_age, predictions_out.conf_int()[:, 1].reshape(-1),
        color='blue', linestyle='dashed');
ax.plot(new_df.house_age, predictions_out.predicted, color='blue');
```



5.3 Multiple Linear Regression

5.3.1 Formal Setup

When we have more than 1 explanatory variable, we turn to multiple linear regression - a more general version of what we have been dealing with so far. We still assume that we have observed information from n individuals, but for each one, we now observe a vector of values:

$$Y_i,\, X_{1,i},\, X_{2,i}, \dots ,\, X_{p-1,i}, X_{p,i}$$

In other words, we observe p independent variables and 1 response variable for each individual in our dataset. The analogous equation to the earlier model is

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \cdots + \beta_p X_{p,i} + e$$

It is easier to write things with matrices for multiple linear regression:

$$\mathbf{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix},\ \mathbf{X} = \begin{bmatrix} 1 & X_{1,1} & X_{2,1} & \cdots & X_{p,1} \\ 1 & X_{1,2} & X_{2,2} & \cdots & X_{p,2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & X_{1,n} & X_{2,n} & \cdots & X_{p,n} \end{bmatrix},\ \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix},\ e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

With the above matrices, we can re-write the regression model as

$$\mathbf{Y} = \mathbf{X}\beta + \mathbf{e}$$

We retain the same distributional assumptions as in simple linear regression.

5.3.2 Estimation

Similar to estimation in the earlier case, we can define SS_E to be

$$SS_E = S(\beta_0, \beta_1, \dots, \beta_p) = \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_{1,i} - \cdots - \beta_p X_{p,i})^2$$

Minimising the above cost function leads to the OLS estimates:

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

The fitted values can be computed with

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

Residuals are obtained as

$$\mathbf{r} = \mathbf{Y} - \hat{\mathbf{Y}}$$

Finally, we estimate σ^2 using

$$\hat{\sigma^2} = \frac{SS_{Res}}{n-p} = \frac{\mathbf{r}'\mathbf{r}}{n-p}$$

5.3.3 Adjusted R^2

In the case of multiple linear regression, R^2 is calculated exactly as in simple linear regression, and its interpretation remains the same:

$$R^2 = 1 - \frac{SS_{Res}}{SS_T}$$

However, note that R^2 can be inflated simply by adding more terms to the model (even insignificant terms). Thus, we use the adjusted R^2, which penalizes us for adding more and more terms to the model:

$$R^2_{adj} = 1 - \frac{SS_{Res}/(n-p)}{SS_T/(n-1)}$$

5.3.4 Hypothesis Tests

The F-test in the multiple linear regression helps determine if our regression model provides any advantage over the simple mean model. The null and alternative hypotheses are:

$$\begin{aligned} H_0 &: \beta_1 = \beta_2 = \cdots = \beta_p = 0 \\ H_1 &: \beta_j \neq 0 \text{ for at least one } j \in \{1, 2, \ldots, p\} \end{aligned}$$

The test statistic is

$$F_1 = \frac{SS_{Reg}/p}{SS_{Res}/(n-p-1)}$$

Under the null hypothesis, $F_0 \sim F_{p,n-p-1}$.

It is also possible to test for the significance of individual β terms, using a t-test. The output is typically given for all the coefficients in a table. The statement of the hypotheses pertaining to these tests is:

$$\begin{aligned} H_0 &: \beta_j = 0 \\ H_1 &: \beta_j \neq 0 \end{aligned}$$

However, note that these t-tests are partial because it should be interpreted as a test of the contribution of β_j, *given that all other terms are already in the model.*

Example 5.4 (Example: Price vs. House Age and Distance to MRT)**.** For our first example on multiple linear regression, let us regress price (Y) on house age (X_1) and distance to MRT (X_2).

```
lm_age_mrt_1 = ols('price ~ house_age + dist_MRT', data=re2).fit()
print(lm_age_mrt_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.491
Model:                            OLS   Adj. R-squared:                  0.489
Method:                 Least Squares   F-statistic:                     198.3
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           5.07e-61
Time:                        09:16:31   Log-Likelihood:                -1527.9
No. Observations:                 414   AIC:                             3062.
Df Residuals:                     411   BIC:                             3074.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     49.8856      0.968     51.547      0.000      47.983      51.788
house_age     -0.2310      0.042     -5.496      0.000      -0.314      -0.148
dist_MRT      -0.0072      0.000    -18.997      0.000      -0.008      -0.006
==============================================================================
Omnibus:                      161.397   Durbin-Watson:                   2.130
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1297.792
Skew:                           1.443   Prob(JB):                    1.54e-282
Kurtosis:                      11.180   Cond. No.                     3.37e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.37e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The estimated equation for the model is

$$Y = 49.89 - 0.23X_1 - 0.0072X_2$$

At 5%-level, the estimates for both β_1 and β_2 are significantly different from 0. Moreover, the adjusted R^2 is now a more respectable 0.49.

Example 5.5 (Example: Broken Line Regression)**.** Although multiple linear regression is usually carried out with distinct variables, it is possible to include functions of the same variable. Suppose we define:

- X_1: House age
- X_2: Distance to MRT
- X_3 to be as follows:

$$X_3 = \begin{cases} 0 & \text{if } X_1 \leq 25, \\ X_1 - 25 & \text{if } X_1 > 25 \end{cases}$$

This allows a non-linear function of house age to be included in the model. It is similar to including a polynomial term, but this is simpler to interpret.

```
re2['x3'] = [(x - 25) if x > 25 else 0 for x in re2.house_age]

lm_age_mrt_2 = ols('price ~ house_age + x3 + dist_MRT', data=re2).fit()
print(lm_age_mrt_2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.527
Model:                            OLS   Adj. R-squared:                  0.523
Method:                 Least Squares   F-statistic:                     152.0
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           3.24e-66
Time:                        09:16:31   Log-Likelihood:                -1512.9
No. Observations:                 414   AIC:                             3034.
Df Residuals:                     410   BIC:                             3050.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     52.9935      1.090     48.598      0.000      50.850      55.137
house_age     -0.6047      0.079     -7.674      0.000      -0.760      -0.450
x3             1.1553      0.209      5.533      0.000       0.745       1.566
dist_MRT      -0.0065      0.000    -16.644      0.000      -0.007      -0.006
==============================================================================
Omnibus:                      165.669   Durbin-Watson:                   2.120
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1558.270
Skew:                           1.436   Prob(JB):                         0.00
Kurtosis:                      12.060   Cond. No.                     3.95e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.95e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

We've managed to improve the adjusted R^2 by a little, but let's focus on understanding the estimated relationship. To help us out, let's fix the distance-to-MRT to be two values: 300m, and 1500m. Roughly speaking, these values correspond to the 1st and 3rd quartiles of the distance values. For a given distance value, we then have two sub-cases:

1. House age less than or equal to 25 years, and
2. House age more than 25 years.

The full equation is as follows:

$$Y = 53.00 - 0.60X_1 - 0.0065X_2 + 1.16X_3$$

The special cases correspond to:

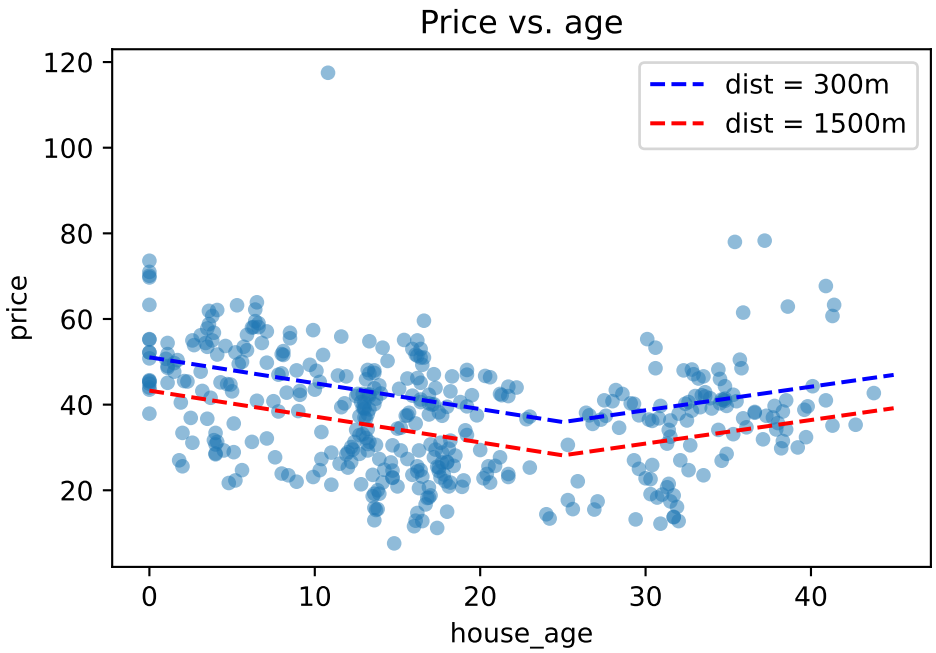
1. $X_2 = 300$:

    1. $X_1 <= 25$:
        - $Y = 51.05 - 0.60X_1$
    2. $X_1 > 25$:
        - $Y = 22.05 + 0.56X_1$
2. $X_2 = 1500$:
    1. $X_1 <= 25$:
        - $Y = 43.25 - 0.60X_1$
    2. $X_1 > 25$:
        - $Y = 14.25 + 0.56X_1$

Visually, here is what the estimated lines look like:

```
new_df = sm.add_constant(pd.DataFrame({'house_age' : np.linspace(0, 45, 100)}))
new_df['x3'] = [(x - 25) if x > 25 else 0 for x in new_df.house_age]
new_df2 = new_df.copy()

new_df['dist_MRT'] = 300
new_df2['dist_MRT'] = 1500

predictions_out = lm_age_mrt_2.get_prediction(new_df)
predictions_out2 = lm_age_mrt_2.get_prediction(new_df2)
```

```
ax = re2.plot(x='house_age', y='price', kind='scatter', alpha=0.5 )
ax.set_title('Price vs. age');
ax.plot(new_df.house_age, predictions_out.predicted_mean,
        color='blue', linestyle='dashed', label='dist = 300m');
ax.plot(new_df2.house_age, predictions_out2.predicted_mean,
        color='red', linestyle='dashed', label='dist = 1500m');
ax.legend();
```



5.4 Including a Categorical Variable

The explanatory variables in a linear regression model do not need to be continuous. Categorical variables can also be included in the model. In order to include them, they have to be coded using dummy variables.

For instance, suppose that we wish to include gender in a model as X_3. There are only two possible genders in our dataset: Female and Male. We can represent X_3 as an indicator variable, with

$$X_{3,i} = \begin{cases} 1 & \text{individual } i \text{ is male} \\ 0 & \text{individual } i \text{ is female} \end{cases}$$

The model (without subscripts for the n individuals) is then:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + e$$

For females, the value of X_3 is 0. Hence the model reduces to

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + e$$

On the other hand, for males, the model reduces to

$$Y = (\beta_0 + \beta_3) + \beta_1 X_1 + \beta_2 X_2 + e$$

The difference between the two models is in the intercept. The other coefficients remain the same.

In general, if the categorical variable has a levels, we will need $a - 1$ columns of indicator variables to represent it. This is in contrast to machine learning models which use one-hot encoding. The latter encoding results in columns that are linearly dependent if we include an intercept term in the model.

Example 5.6 (Example: Price vs. Num Stores and Distance to MRT)**.** For this example, let us work with a reduced model in order to understand how things work. Price will remain the dependent variable, but we shall use distance to MRT (quantitative) and number of nearby convenience stores. However, we shall recode the number of stores as low (or high) corresponding to whether or not there were 4 stores or less (resp. more than 5).

```
re2['num_stores_cat'] = ['low' if x <= 4 else 'high' for x in re2.num_stores]

lm_cat_1 = ols('price ~ dist_MRT + num_stores_cat', re2).fit()
print(lm_cat_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.502
Model:                            OLS   Adj. R-squared:                  0.500
Method:                 Least Squares   F-statistic:                     207.3
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           5.54e-63
Time:                        09:16:31   Log-Likelihood:                -1523.3
```

```
No. Observations:                 414   AIC:                             3053.
Df Residuals:                     411   BIC:                             3065.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================================
                             coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------------------
Intercept                  47.8055      0.696     68.679      0.000      46.437      49.174
num_stores_cat[T.low]      -7.4086      1.171     -6.325      0.000      -9.711      -5.106
dist_MRT                   -0.0055      0.000    -11.913      0.000      -0.006      -0.005
==============================================================================
Omnibus:                      190.015   Durbin-Watson:                   2.138
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             2327.960
Skew:                           1.618   Prob(JB):                         0.00
Kurtosis:                      14.157   Cond. No.                     4.31e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 4.31e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The categorical variable has been coded (by Python) as follows:

$$X_4 = \begin{cases} 1, & \text{if there were 4 nearby stores or less} \\ 0, & \text{otherwise} \end{cases}$$

As a result, we have estimated two models:

- Corresponding to a large number of nearby stores: $Y = 47.81 - 0.0055X_2$
- Corresponding to a small number of nearby stores: $Y = 40.40 - 0.0055X_2$

5.4.1 Including an Interaction Term

A more complex model arises from an interaction between two terms. Here, we shall consider an interaction between a continuous variable and a categorical explanatory variable. Suppose that we have three predictors: height (X_1), weight (X_2) and gender (X_3). As spelt out in the previous section, we should use indicator variables to represent X_3 in the model.

If we were to include an *interaction* between gender and weight, we would be allowing for a males and females to have separate coefficients for X_2. Here is what the model would appear as:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_2 X_3 + e$$

Remember that X_3 will be 1 for males and 0 for females. The simplified equation for males would be:

$$Y = (\beta_0 + \beta_3) + \beta_1 X_1 + (\beta_2 + \beta_4) X_2 + e$$

For females, it would be:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + e$$

Both the intercept and coefficient of X_2 are different, for each value of X_3. Recall that in the previous section, only the intercept term was different.

Example 5.7 (Example: Interaction between Number of Stores and Distance to MRT).

```
lm_cat_2 = smf.ols('price ~ num_stores_cat + dist_MRT + dist_MRT:num_stores_cat', data=df).fit()
print(lm_cat_2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.538
Model:                            OLS   Adj. R-squared:                  0.534
Method:                 Least Squares   F-statistic:                     158.9
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           2.54e-68
Time:                        09:16:31   Log-Likelihood:                -1508.0
No. Observations:                 414   AIC:                             3024.
Df Residuals:                     410   BIC:                             3040.
Df Model:                           3
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      0.
------------------------------------------------------------------------------------------------
Intercept                      54.8430      1.425     38.499      0.000      52.043      57
num_stores_cat[T.low]         -14.9553      1.759     -8.504      0.000     -18.412     -11
dist_MRT                       -0.0278      0.004     -6.948      0.000      -0.036      -0
dist_MRT:num_stores_cat[T.low]  0.0226      0.004      5.602      0.000       0.015       0
==============================================================================
Omnibus:                      215.525   Durbin-Watson:                   2.117
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             3236.645
Skew:                           1.844   Prob(JB):                         0.00
Kurtosis:                      16.192   Cond. No.                     1.10e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.1e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Notice that we now have the largest adjusted R^2 out of all of the models we have fit so far.

The model that we have fit consists of two models:

1. For the case that the number of stores is 4 or less: $Y = 39.88 - 0.005X_2$
2. For the case that the number of stores is more than 4: $Y = 54.84 - 0.028X_2$.

> **Note**
>
> Can you interpret the result above based on your intuition or understanding of real estate prices?

5.5 Residual Analysis

Recall from earlier that residuals are computed as

$$r_i = Y_i - \hat{Y}_i$$

Residual analysis is a standard approach for identifying how we can improve a model. In the case of linear regression, we can use the residuals to assess if the distributional assumptions hold. We can also use residuals to identify influential points that are masking the general trend of other points. Finally, residuals can provided some direction on how to improve the model.

5.5.1 Standardised Residuals

It can be shown that the variance of the residuals is in fact not constant! Let us define the hat-matrix as

$$\mathbf{H} = \mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'$$

The diagonal values of $\mathbf{H}$ will be denoted h_{ii}, for $i = 1, \ldots, n$. It can then be shown that

$$Var(r_i) = \sigma^2(1 - h_{ii}), \quad Cov(r_i, r_j) = -\sigma^2 h_{ij}$$

As such, we use the standardised residuals when checking if the assumption of Normality has been met.

$$r_{i,std} = \frac{r_i}{\hat{\sigma}\sqrt{1 - h_{ii}}}$$

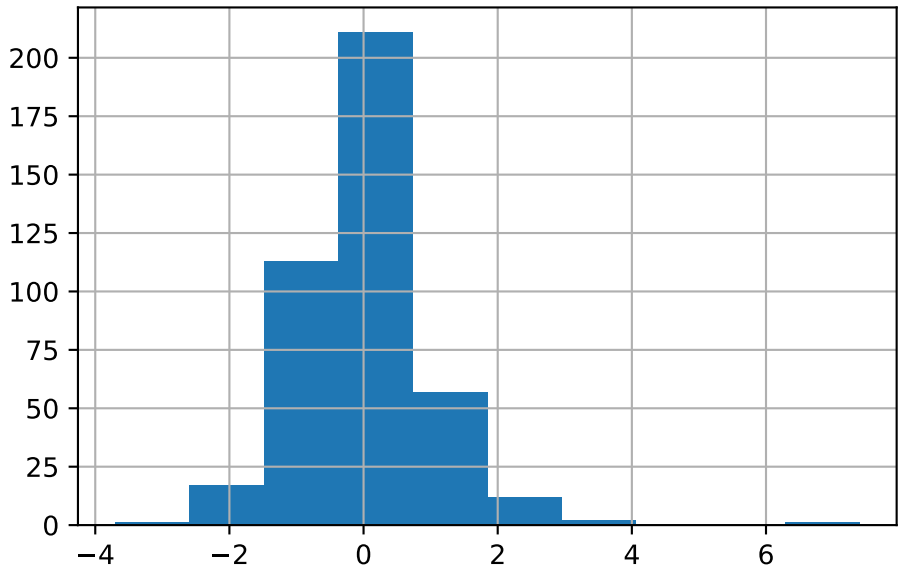
If the model fits well, standardised residuals should look similar to a $N(0, 1)$ distribution. In addition, large values of the standardised residual indicate potential outlier points.

By the way, h_{ii} is also referred to as the *leverage* of a point. It is a measure of the potential influence of a point (on the parameters, and future predictions). h_{ii} is a value between 0 and 1. For a model with p parameters, the average h_{ii} should be should be p/n. We consider points for whom $h_{ii} > 2 \times p/n$ to be high leverage points.

In the literature and in textbooks, you will see mention of residuals, standardised residuals and studentised residuals. While they differ in definitions slightly, they typically yield the same information. Hence we shall stick to standardised residuals for our course.

Example 5.8 (Example: Normality Check for `lm_age_mrt_1`)**.** One of the first checks for Normality is to create a histogram. If the residuals adhere to a Normal distribution, we should observe a symmetric bell-shaped distribution.

```
r_s = pd.Series(lm_age_mrt_1.resid_pearson)
r_s.hist();
```

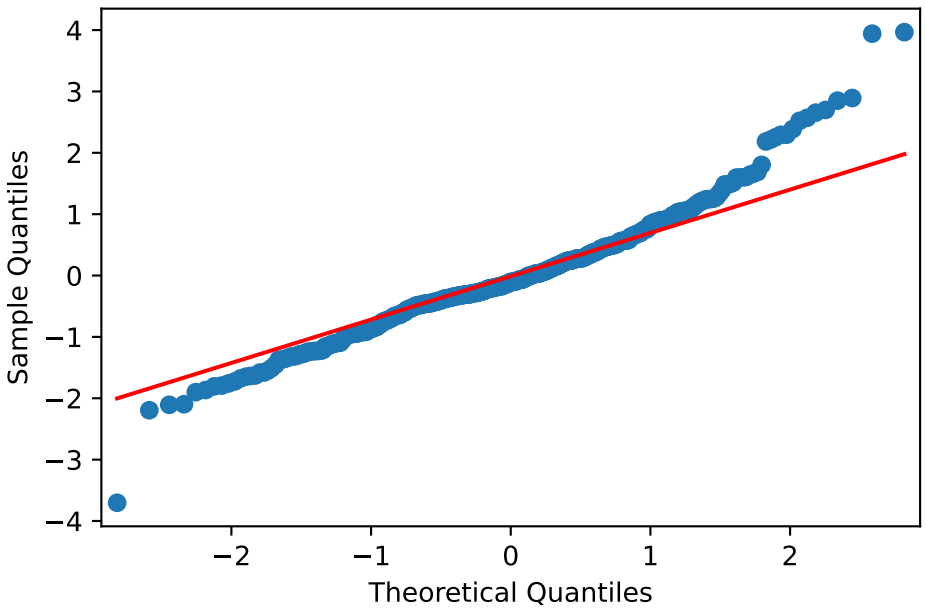


From above, it appears that the distribution is slightly skewed, and there is one noticeable outlier. A second graphical diagnostic plot that we make is a QQ-plot.

A QQ-plot plots the standardized sample quantiles against the theoretical quantiles of a N(0; 1) distribution. If they fall on a straight line, then we would say that there is evidence that the data came from a normal distribution.

Especially for unimodal datasets, the points in the middle will fall close to the line. The value of a QQ-plot is in judging if the tails of the data are fatter or thinner than the tails of the Normal.

```
sm.qqplot(r_s[r_s < 6], line="q");
```



Overall, the residuals do indicate a lack of Normal behaviour. Non-normality in the residuals should lead us to view the hypothesis tests with caution. The estimated models are still valid, in the sense that they are optimal. Estimation of the models did not require the assumption of Normality. So far, we have only focused on inspecting the R^2 to assess the model quality.

5.5.2 Scatterplots

To understand the model fit better, a set of scatterplots are typically made. These are plots of standardised residuals (on the y-axis) against

- fitted values
- explanatory variables, one at a time.
- potential variables.

Residuals are meant to contain only the information that our model cannot explain. Hence, if the model is good, the residuals should only contain random noise. There should be no apparent pattern to them. If we find such a pattern in one of the above plots, we would have some clue as to how we could improve the model.

We typically inspect the plots for the following patterns:

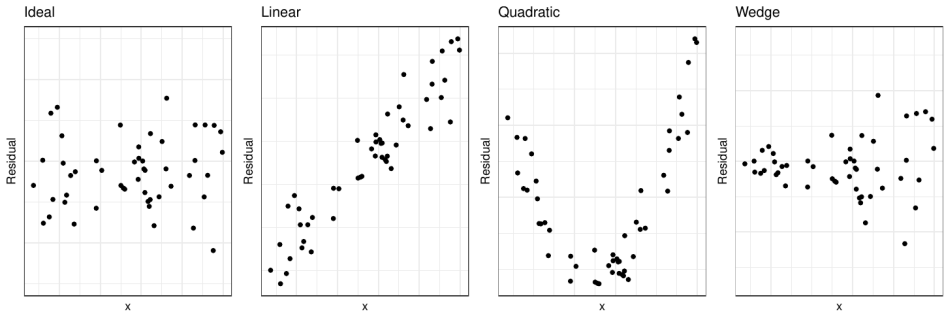


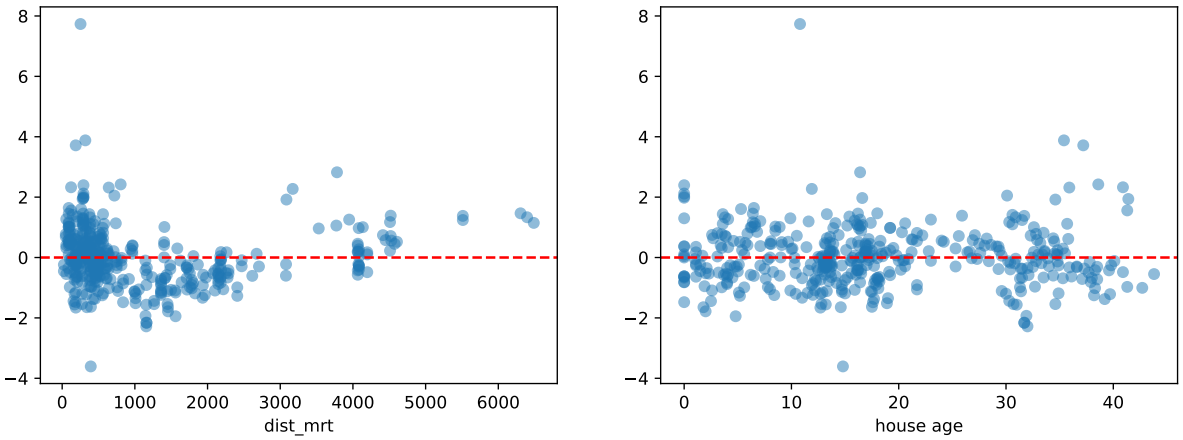
Figure 5.1: Residuals

1. A pattern like the one on the extreme left is ideal. Residuals are randomly distributed around zero; there is no pattern or trend in the plot.
2. The second plot is something rarely seen. It would probably appear if we were to plot residuals against a *new* variable that is not currently in the model. If we observe this plot, we should then include this variable in the model.
3. This plot indicates we should include a quadratic term in the model.
4. The wedge shape (or funnel shape) indicates that we do not have homoscedascity. The solution to this is either a transformation of the response, or weighted least squares. You will cover these in your linear models class.

Example 5.9 (Example: Residual Plots for `lm_age_mrt_2`)**.** Let us extract and create the residual plots for the second model that we had fit, earlier.

```
plt.figure(figsize=(12,4));
r_s = lm_age_mrt_2.resid_pearson
ax=plt.subplot(121)
ax.scatter(re2.dist_MRT, r_s, alpha=0.5)
ax.set_xlabel('dist_mrt')
```

```
ax.axhline(y=0, color='red', linestyle='--')

ax=plt.subplot(122)
ax.scatter(re2.house_age, r_s, alpha=0.5)
ax.set_xlabel('house age');
ax.axhline(y=0, color='red', linestyle='--');
```



While the plot for house age looks acceptable (points are evenly scattered about the red dashed line), the one for distance shows some curvature. This is something we can try to fix, using a transformation of the x-variable.

5.5.3 Influential Points

The influence of a point on the inference can be judged by how much the inference changes with and without the point. For instance to assess if point i is influential on coefficient j:

1. Estimate the model coefficients with all the data points.
2. Leave out the observations (Y_i, X_i) one at a time and re-estimate the model coefficients.
3. Compare the β's from step 2 with the original estimate from step 1.

While the above method assesses influence on parameter estimates, Cook's distance performs a similar iteration to assess the influence on the fitted values. Cook's distance values greater than 1 indicate possibly influential points.

There are several ways to deal with influential points. First, we can remove the influential point (or sets of points) and asssess how much the model changes. Based on our understanding of the domain, we can then decide to keep or remove those points. A second approach is to create a dummy variable that identifies those points (individually). Fitting the subsequent model allows all points to be used in estimating standard errors, but quantifies an adjustment for those points. A third approach is to use a *robust linear model*. This is a model that automatically reduces the influence of aberrant points. This is a good topic to know about - do read up on it if you are keen!

Example 5.10 (Example: Influential Points for `lm_age_mrt_2`)**.** The influence of a point on the inference can be judged by how much the inference changes with and without the point. For instance to assess if point i is influential on coefficient j:

1. Estimate the model coefficients with all the data points.
2. Leave out the observations (Y_i, X_i) one at a time and re-estimate the model coefficients.
3. Compare the β's from step 2 with the original estimate from step 1.

While the above method assesses influence on parameter estimates, Cook's distance performs a similar iteration to assess the influence on the fitted values. Cook's distance values greater than 1 indicate possibly influential points.

```
infl = lm_age_mrt_2.get_influence()
infl_df = infl.summary_frame()
```

```
print(infl_df.head())
```

```
   dfb_Intercept  dfb_house_age    dfb_x3  dfb_dist_MRT   cooks_d  \
0       0.004477      -0.015423  0.004746      0.015011  0.000249
1      -0.004122       0.022294 -0.024375     -0.017434  0.000236
2       0.016558       0.009214 -0.017881     -0.018001  0.000405
3       0.037359       0.020789 -0.040346     -0.040616  0.002054
4      -0.037629       0.024643 -0.013610      0.006609  0.000375

   standard_resid  hat_diag  dffits_internal  student_resid    dffits
0       -0.350312  0.008050        -0.031558      -0.349937 -0.031524
1        0.319229  0.009195         0.030754       0.318879  0.030720
2        0.638887  0.003955         0.040257       0.638426  0.040228
3        1.438604  0.003955         0.090648       1.440488  0.090767
4       -0.463313  0.006946        -0.038748      -0.462869 -0.038711
```

5.6 Transformation

Example 5.11 (Example: Log-transformation)**.** As we observed in the residual plots, the distance-to-MRT variable displays a slight curvature. We can fix this by taking a log-transformation of the variable before fitting the model.

Below, we include the code to perform this fitting.

```
re2['ldist'] = np.log(re2.dist_MRT)

lm_age_mrt_3 = ols('price ~ house_age + x3 + num_stores + ldist', data=re2).fit()
print(lm_age_mrt_3.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.597
Model:                            OLS   Adj. R-squared:                  0.593
Method:                 Least Squares   F-statistic:                     151.6
Date:                Tue, 23 Sep 2025   Prob (F-statistic):           2.07e-79
Time:                        09:16:31   Log-Likelihood:                -1479.4
No. Observations:                 414   AIC:                             2969.
```

```
Df Residuals:                      409   BIC:                             2989.
Df Model:                            4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     84.0709      4.009     20.971      0.000      76.190      91.951
house_age     -0.4941      0.075     -6.578      0.000      -0.642      -0.346
x3             0.8599      0.197      4.355      0.000       0.472       1.248
num_stores     0.7815      0.201      3.886      0.000       0.386       1.177
ldist         -6.6595      0.559    -11.910      0.000      -7.759      -5.560
==============================================================================
Omnibus:                      233.692   Durbin-Watson:                   2.059
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             3847.453
Skew:                           2.030   Prob(JB):                         0.00
Kurtosis:                      17.372   Cond. No.                         213.
==============================================================================

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

Now, take some to investigate the following issues:

1. Interpret the coefficient for `dist_MRT`.
2. Have the issues with the residuals been fixed?
3. What is the difference between how this model uses `num_stores`, and how `lm_cat_1` uses it?
4. How did we choose 25 as the breakpoint for house-age? Is it the ideal one?

5.7 Summary, Further topics

Linear regression is a very flexible model. It is quick to fit, easily generalisable and much more interpretable than other models. These are some of the reasons why it is still one of the most widely used models in industry.

In our short session, we have touched on several practical tips for using regression models. However, take note that regression models can be generalised in many other ways. Here are some models you may want to read up on:

- Assuming correlated errors instead of independent error terms
- Using splines to include non-linear functions of explanatory variables.
- Kernel regression is an even more modern method for including higher-order terms, but at this point we start to lose interpretability
- Constrained regression, when we know certain coefficients should be positive, for instance.
- Robust linear models to automagically take care of wild outliers.

Good reference textbooks for this topic are Draper (1998) and Hastie, Tibshirani, and Friedman (2009).

5.8 References

5.8.1 Website References

1. Taiwan dataset from UCI machine learning repository
2. Stats models documentation
3. Diagnostics
4. On residual plots

6 Time Series Analysis

6.1 Exploring Time Series Data

One of the first things that we do when we are given a time series is visualise it. The visualisation is meant to give us an indication of what kinds of techniques would be suitable for forecasting it. In this section, we shall learn several methods to visualise a time series dataset.

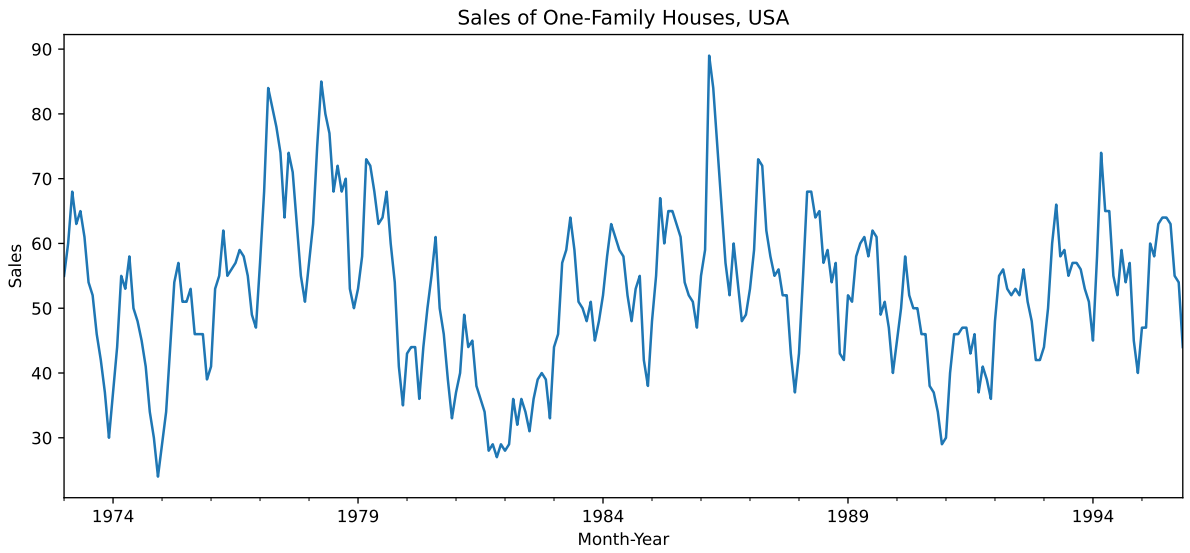
When we visualise a time series, we look out for the following features:

1. **Trend**: A trend exists when there is a long-term increase or decrease in the data.
2. **Level**: The level of a series refers to its height on the ordinate axis.
3. **Seasonal**: A seasonal pattern exists when a series is influenced by factors such as quarters of the year, the month, the day of the week, or time of day. Seasonality is always of a **fixed and known** period.
4. **Cyclic**: A cyclic pattern exists when there are rises and falls that are not of a fixed period.

```
import pandas as pd
import numpy as np
import datetime, calendar
from statsmodels.tsa.seasonal import seasonal_decompose,STL
from statsmodels.tsa.statespace.tools import diff
from statsmodels.tsa.stattools import acf
from statsmodels.tsa.forecasting.theta import ThetaModel
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace import exponential_smoothing

from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt, STLForecast
import pmdarima as pm

from scipy.cluster import hierarchy
from scipy.spatial.distance import pdist,squareform

from ind5003 import ts

import matplotlib.pyplot as plt
import seaborn as sns
```

Example 6.1 (Example: Basic Plot of Housing Data)

```
hsales = pd.read_csv('Data/hsales.csv', parse_dates=[0])
hsales.set_index('date', inplace=True)
hsales.index.freq = 'MS'
hsales.plot(title='Sales of One-Family Houses, USA', legend=False, figsize=(12,5))
plt.xlabel('Month-Year'); plt.ylabel('Sales');
```



We observe a strong seasonality within each year. There is some indication of cyclic behaviour every 6 – 10 years. There is no apparent monotonic trend over this period. With pandas objects, we can resample the series. This allows us to compute summaries over time windows that could be of business importance. For instance, we might be interested in a breakdown by quarters instead of months.

With the `resample()` function, we can perform both downsampling (reducing the frequency of observations) or upsampling (via interpolation or nearest neighbour imputation).

```
hsales.resample('14D').interpolate().head()
#hsales.head()
```

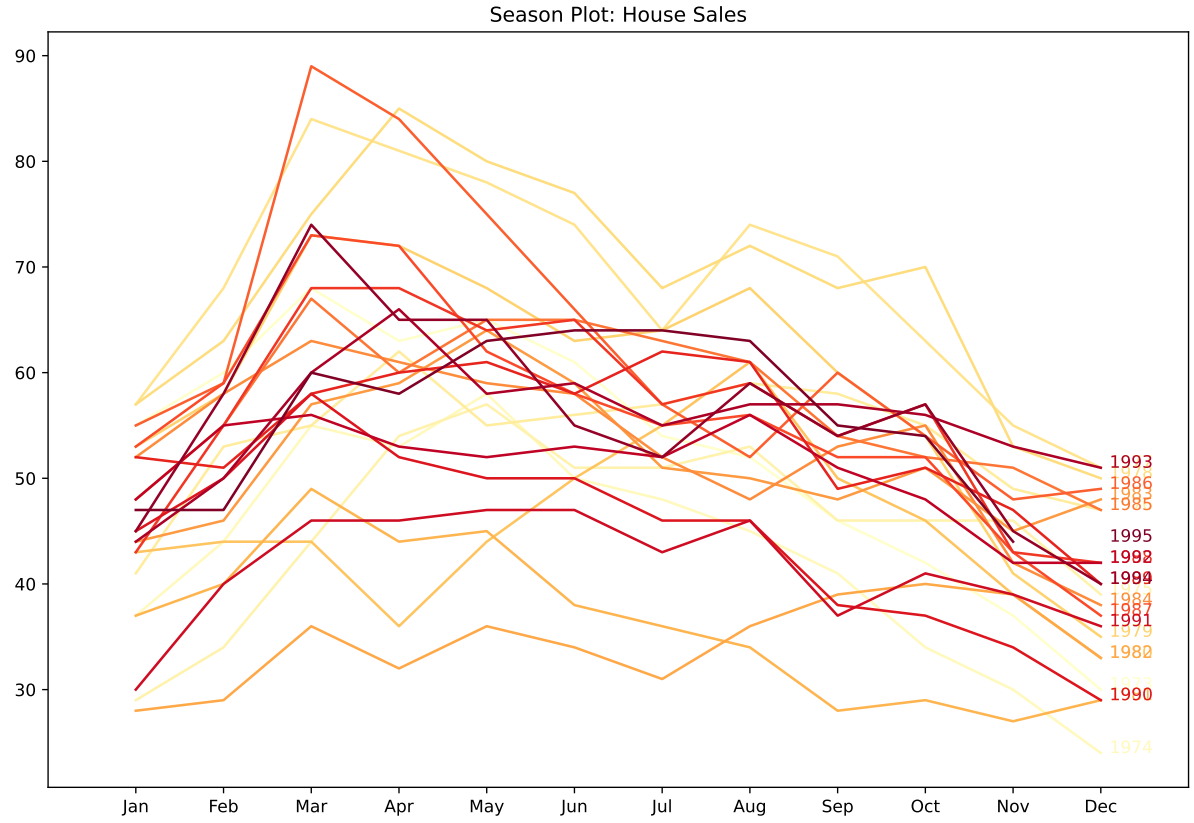
	hsales
date	
1973-01-01	55.000000
1973-01-15	54.820513
1973-01-29	54.641026
1973-02-12	54.461538
1973-02-26	54.282051

> **Note**
>
> See the page on date-offset objects for details and options on the strings that you can put within the resample method.

Example 6.2 (Example: Season Plots of Housing Data)**.** A season plot allows us to visualise what happens within a season. In this case, each line in the graph below traces the behaviour of the series from the beginning of the season till the end (from Jan to Dec).

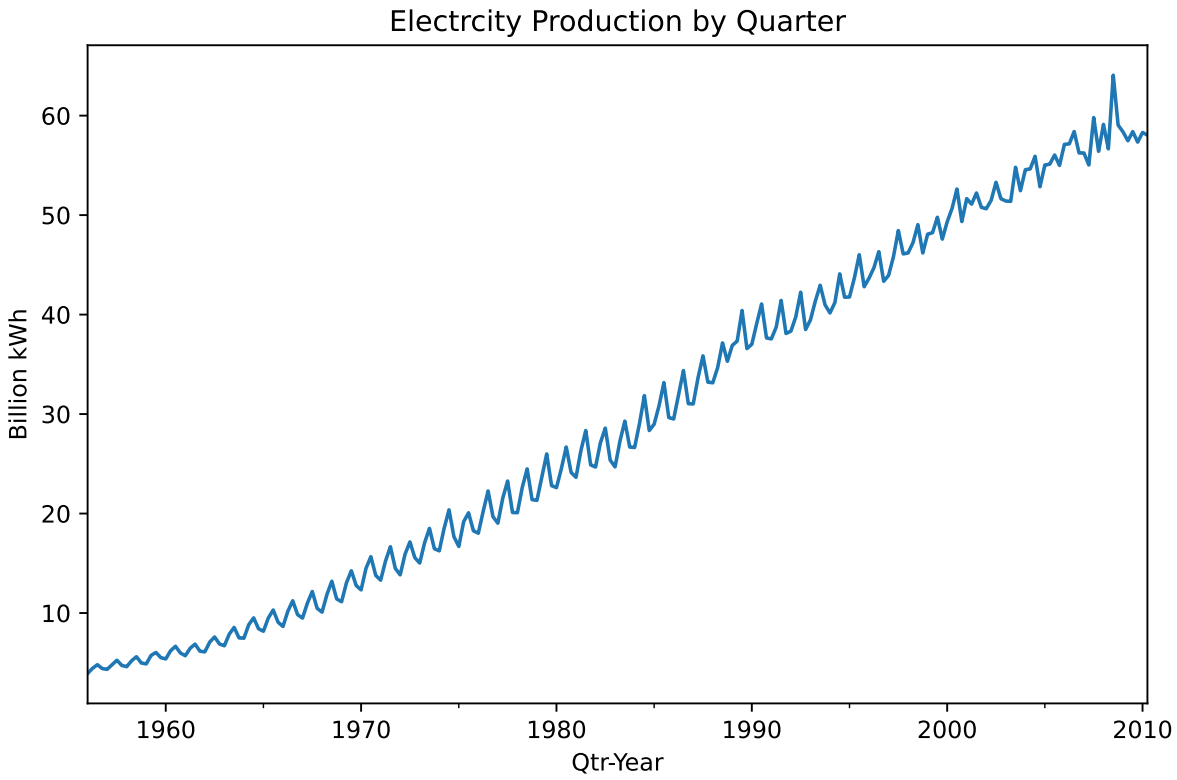
```
hsales.loc[:, 'year'] = hsales.index.year
hsales.loc[:, 'month'] = hsales.index.month
```

```
yrs = np.sort(hsales.year.unique())
color_ids = np.linspace(0, 1, num=len(yrs))
colors_to_use = plt.cm.YlOrRd(color_ids)

plt.figure(figsize=(12, 8))

for i,yr in enumerate(yrs):
    df_tmp = hsales.loc[hsales.year == yr, :]
    plt.plot(df_tmp.month, df_tmp.hsales, color=colors_to_use[i]);
    plt.text(12.1, df_tmp.hsales.iloc[-1], str(yr), color=colors_to_use[i])
plt.title('Season Plot: House Sales')
plt.xlim(0, 13)
plt.xticks(np.arange(1, 13), calendar.month_abbr[1:13]);
```



Example 6.3 (Example: Basic Time Plot, Australian Quarterly Electricity Production).

```
qau = pd.read_csv('data/austelec.csv', parse_dates=['date'])
qau.set_index('date', inplace=True)
qau.index.freq = 'QS'

qau.plot(figsize=(8,5), title='Electrcity Production by Quarter', legend=False)
plt.xlabel('Qtr-Year'); plt.ylabel('Billion kWh');
```

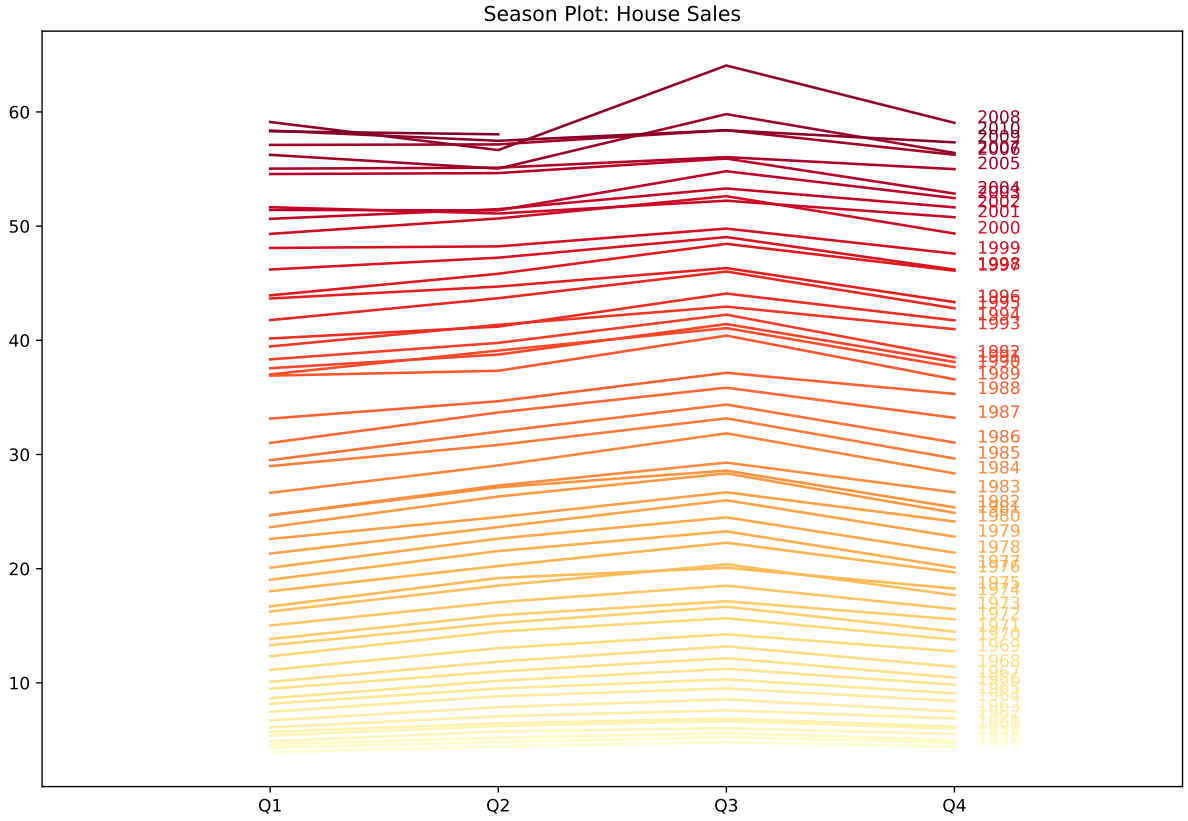


There is a strong increasing trend. There is strong seasonality, and there is no evidence of cyclic behaviour.

In general, it looks like there is a peak around Feb to March, after which sales descend until the next January. Using a colour map with colours that we can remember would allow us to identify a trend **across** seasons. In this case, there isn't one, but if you re-do this plot for the Electricity series, you would see a clear association between the colour of line and level of each series within a year.

```
qau2 = qau.copy()

qau2.loc[:, 'year'] = qau2.index.year
qau2.loc[:, 'qtr'] = qau2.index.quarter
```

```
yrs = np.sort(qau2.year.unique())
color_ids = np.linspace(0, 1, num=len(yrs))
colors_to_use = plt.cm.YlOrRd(color_ids)

plt.figure(figsize=(12, 8))

for i,yr in enumerate(yrs):
    df_tmp = qau2.loc[qau2.year == yr, :]
    plt.plot(df_tmp.qtr, df_tmp.kWh, color=colors_to_use[i]);
    plt.text(4.1, df_tmp.kWh.iloc[-1], str(yr), color=colors_to_use[i])
plt.title('Season Plot: House Sales')
plt.xlim(0, 5)
plt.xticks(np.arange(1, 5), ['Q1', 'Q2', 'Q3', 'Q4']);
```



Most time series models are autoregressive in nature. This means that they attempt to predict future observations based on previous ones. The observations from the past could be from the time series that we are interested in, or they could be from other time series that we believe are related.

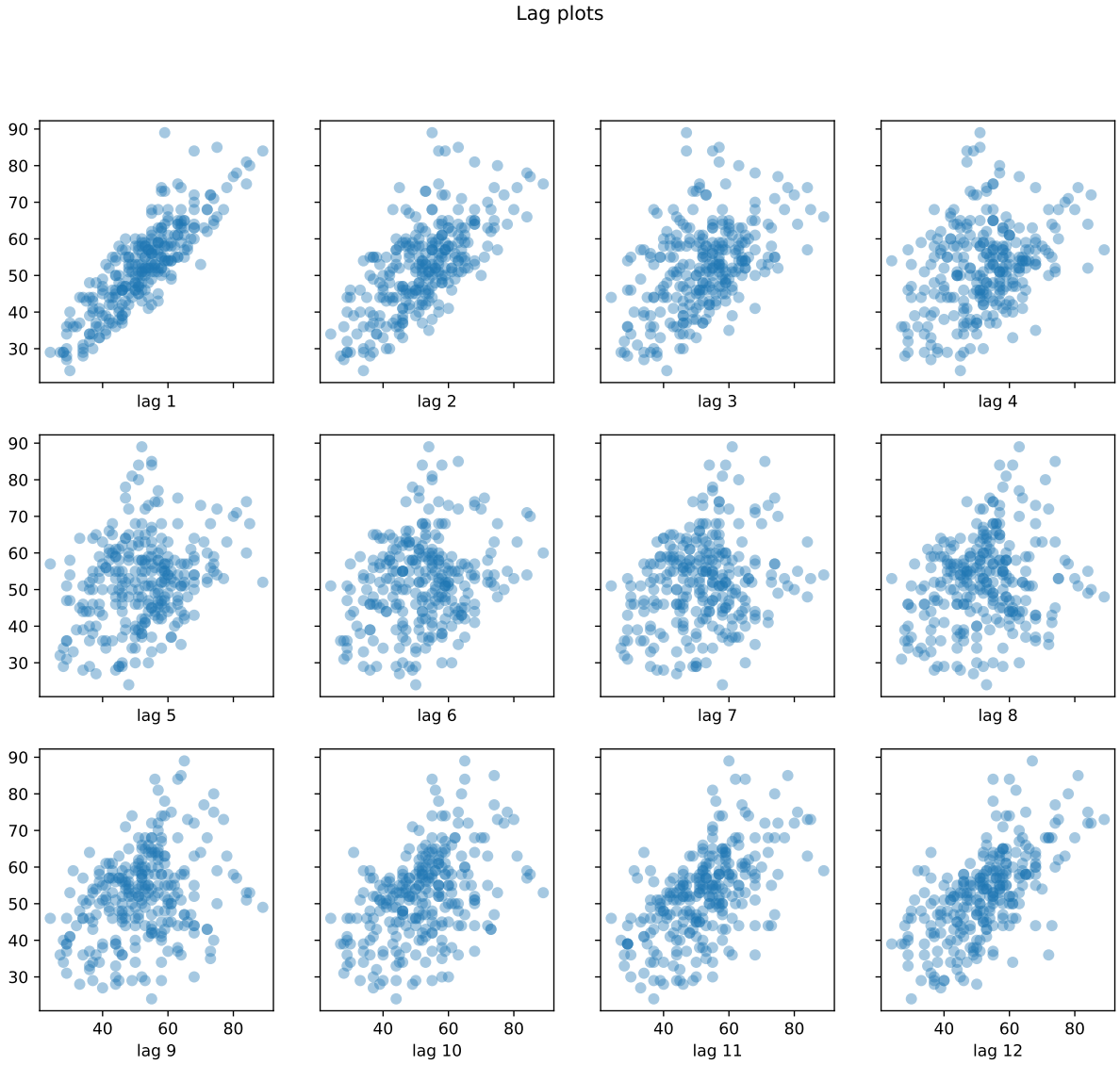
When beginning with model fitting, we would want to have some idea about the extent to which past observations affect the current one. In other words, how many of the past observations should we include when forecasting new observations? Lag plots and autocorrelation functions (acf) are the tools that we use to answer this question.

If we denote the observation of a time series as y_t, then a lag plot at lag $k, k > 0$ is a scatter plot of y_t against y_{t-k}. It allows us to visually assess if the observations that are k units of time apart are associated with one another. We typically plot several lags at once to observe this relationship.

Example 6.4 (Explore: Lag plots of House Sales Data).

```
f, aa = plt.subplots(nrows=3, ncols=4, sharex=True, sharey=True)
f.set_figheight(10)
f.set_figwidth(12)

y = hsales.hsales.values
for i in np.arange(0, 3):
    for j in np.arange(0, 4):
        lag = i*4 + j + 1
        aa[i,j].scatter(y[:-lag], y[lag:], alpha=0.4)
```

```
        aa[i,j].set_xlabel("lag " + str(lag))
f.suptitle('Lag plots');
```



6.2 Decomposing Time Series Data

In this section, our goal is still primarily exploratory, but it will also return information that we can use to model our data later on. We aim to break the time series readings into:

- a trend-cycle component,
- a seasonal component, and
- a remainder component.

If we can forecast these components individually, we can combine them to provide forecasts for the original series. There are two basic methods of decomposition into the above components:

an additive one, and a multiplicative one. In the additive decomposition, we assume that they contribute in an additive manner to y_t:

$$y_t = S_t + T_t + R_t \tag{6.1}$$

In the multiplicative decomposition, we assume the following relationship holds:

$$y_t = S_t \times T_t \times R_t \tag{6.2}$$

When we are in the multiplicative case, we sometimes apply the logarithm transform to the data, which returns to an additive model:

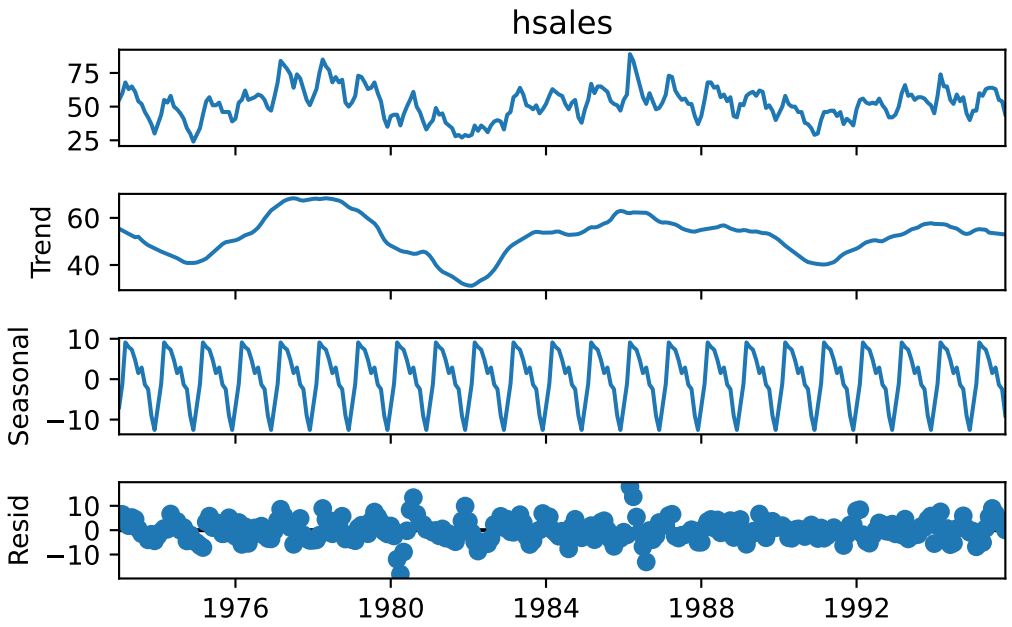
$$\log y_t = \log S_t + \log T_t + \log R_t \tag{6.3}$$

A typical algorithm to decompose time series data would work like this:

1. Estimate the trend component. Let us call it $\hat{T}_t$.
2. Obtain the de-trended data $y_t - \hat{T}_t$ or $y_t/\hat{T}_t$ as appropriate.
3. Estimate the seasonal component. For instance, we could simply average the values in each month. Let us denote this estimate as $\hat{S}_t$.
4. Estimate the remainder component. For instance, in the additive model, it would be $\hat{R}_t = y_t - \hat{T}_t - \hat{S}_t$.

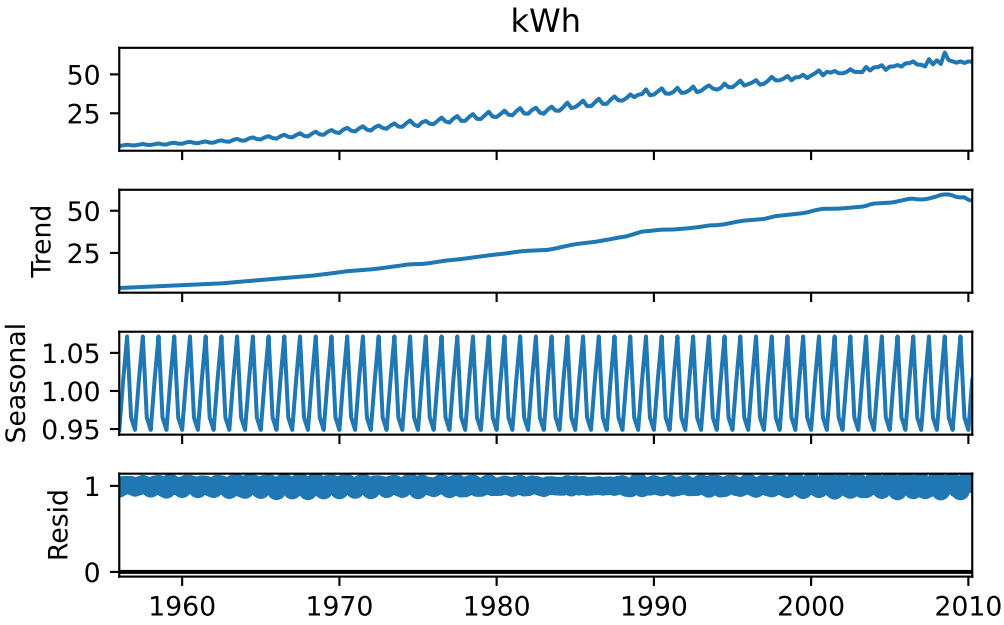
Example 6.5 (Example: Additive Decomposition, Housing Sales)**.** Here is the naive additive decomposition of the housing sales data.

```
hsales_add = seasonal_decompose(hsales.loc[:, 'hsales'],
                                model='additive', extrapolate_trend='freq')
hsales_add.plot();
```



Example 6.6 (Example: Multiplicative Decomposition, Aus Electricity Data)**.** Here is the *multiplicative decomposition for the Australian electric data.

```
qau_mult = seasonal_decompose(qau.loc[:, 'kWh'], model='multiplicative',
                              extrapolate_trend='freq')
qau_mult.plot();
```

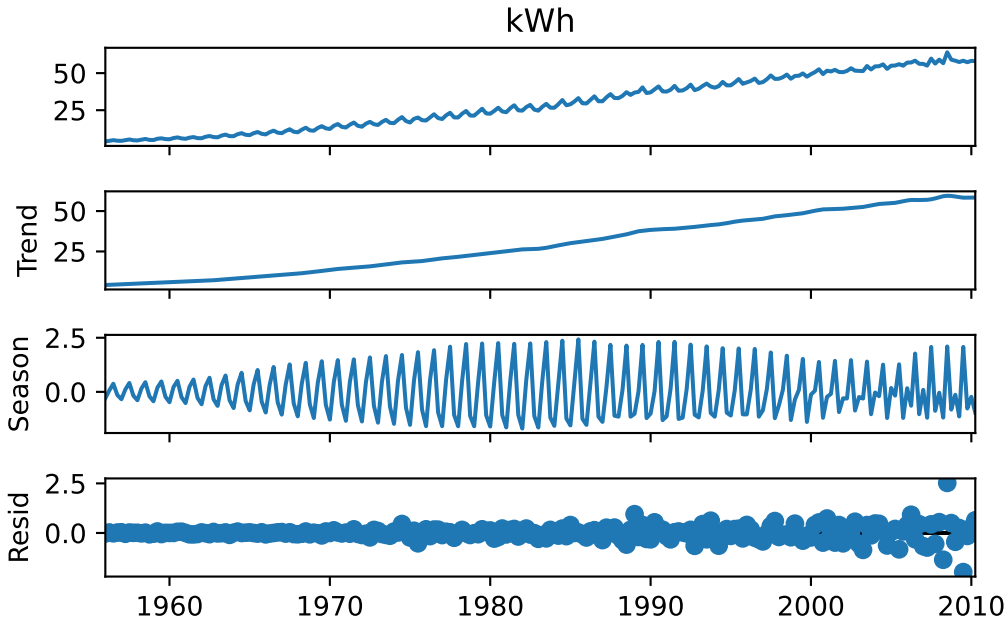


As you may have noticed, there are some issues with the classical seasonal decomposition algorithms. These include:

- an inability to estimate the trend at the ends of the series.
- an inability to account for changing seasonal components.
- it is not robust to outliers.

Example 6.7 (Example: STL decomposition, Aus Electricity)**.** Newer algorithms such as STL decomposition, X11 and others attempt to address these issues. They use locally weighted regression models to obtain the trend. These algorithms iterate over the time series several times, so as to ensure that outliers are not affecting the outcome.

```
qau_stl = STL(qau.kWh).fit()
qau_stl.plot();
```



6.3 Forecasting

6.3.1 Benchmark methods

As in all forecasting methods, it is useful to obtain a baseline forecast before proceeding to more sophisticated techniques. Baseline forecasts are usually obtained from simple, intuitive methods. Here are some such methods:

A. The simple mean forecast:

$$\hat{y}_{T+h|T} = \frac{y_1 + y_2 + y_3 + \cdots + y_T}{T} \tag{6.4}$$

B. The naive forecast:

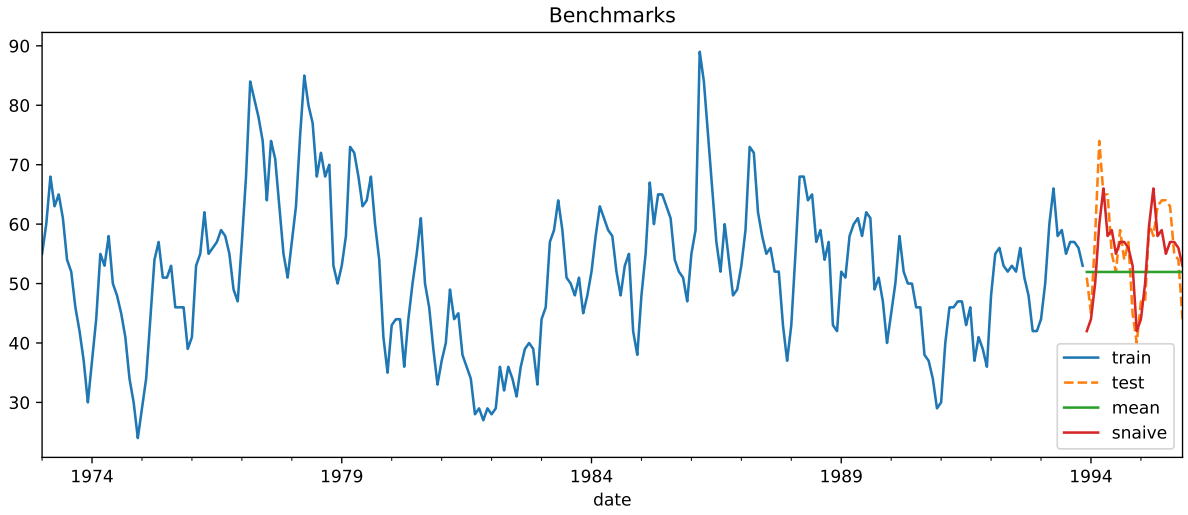
$$\hat{y}_{T+h|T} = y_T \tag{6.5}$$

C. The seasonal naive forecast.

Example 6.8 (Example: Benchmark Forecasts Housing Sales)**.** Suppose we apply some of the above forecasts to the housing sales dataset. We withhold the most recent 2 years of data and foreacst those. Here is a plot depicting the forecasts, and the true values.

```
# Set aside the last two years as the test set.
#hsales = hsales.drop(columns=['year', 'month'])
train_set = hsales.iloc[:-24,]
test_set = hsales.iloc[-24:, ]

# Obtain the forecast from the training set
mean_forecast = ts.meanf(train_set.hsales, 24)
snaive_forecast = ts.snaive(train_set.hsales, 24, 12)
```

```
# Plot the predictions and true values
ax = train_set.hsales.plot(title='Benchmarks', legend=False, figsize=(12,4.5))
test_set.hsales.plot(ax=ax, legend=False, style='--')
mean_forecast.plot(ax=ax, legend=True, style='-')
snaive_forecast.plot(ax=ax, legend=False, style='-')
plt.legend(labels=['train', 'test', 'mean', 'snaive'], loc='lower right');
```



When assessing forecasts, there are a few different metrics that are typically applied. Each of them has it's own set of pros and cons. If we denote the predicted value with $\hat{y}_t$, then these are the formulas for three of the most common error metrics used in time series forecasting

A. RMSE:

$$\sqrt{\frac{1}{h}\sum_{i=1}^{h}(y_{t+i} - \hat{y}_{t+i})^2} \tag{6.6}$$

B. MAE

$$\frac{1}{h}\sum_{i=1}^{h}|y_{t+i} - \hat{y}_{t+i}| \tag{6.7}$$

C. Mean Absolute Scaled Error

$$\frac{1}{h}\sum_{i=1}^{h}\frac{|y_{t+i} - \hat{y}_{t+i}|}{\frac{1}{T-1}\sum_{t=2}^{T}|y_t - y_{t-1}|} \tag{6.8}$$

The RMSE and MAE are scale dependent errors. It is difficult to compare the errors across, or to aggregate errors across different time series with it. Due to the square in the formula, the RMSE is quite sensitive to outliers. The MAE is more robust to outliers. The MASE is a scaled error - it allows us to compare the forecasting performance *across* time series. The other two metrics depend on the scale of the time series and hence do not allow us to make such comparisons.

```
for x in [ts.rmse, ts.mae]:
    print(f"{x.__name__},mean: {x(test_set.hsales.values, mean_forecast.values):.3f}")
    print(f"{x.__name__},snaive: {x(test_set.hsales.values, snaive_forecast.values):.3f}")
    print('---')
```

```
rmse,mean: 9.023
rmse,snaive: 5.906
---
mae,mean: 7.562
mae,snaive: 4.792
---
```

The MASE for the seasonal naive forecast is as follows.

```
ts.mase(test_set.hsales.values,  snaive_forecast.values, train_set.values,
        seasonality=12)
```

```
1.5154940449933834
```

In our simple example, the seasonal naive model outperforms the simple mean forecast according to all the metrics but in general, things are not always this clear-cut.

6.4 ARIMA Models

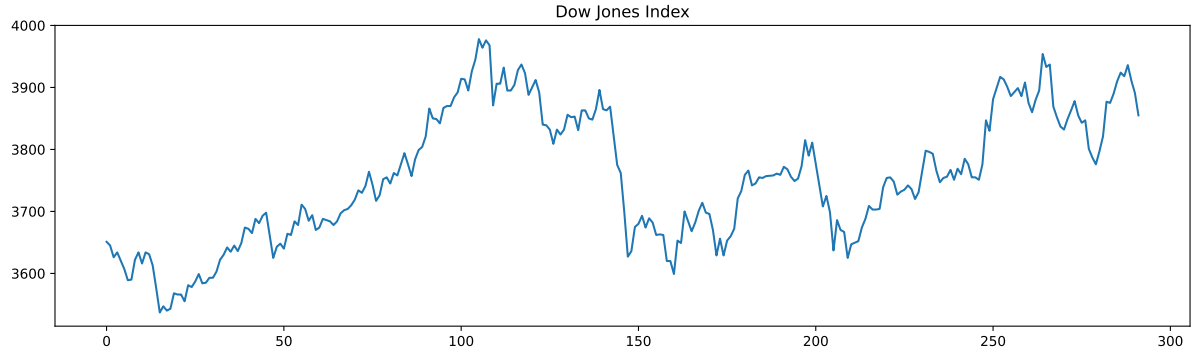
Now let us turn to a huge class of models that have been utilised in time series forecasting since the 1960's. They are known as ARIMA models. ARIMA stands for AutoRegressive Integrated Moving Average models. These models are appropriate for **stationary** processes. Stationarity is a technical term that refers to processes

- that have a constant mean. This means that the process merely fluctuates about a fixed level over time.
- whose covariance function does not change over time. This means that, for a fixed h, the covariance between y_t and y_{t+h} is the same for all t.
- whose variance is constant over time.

How can we tell if a process is stationary or not? The following time series is not stationary. Why?

Example 6.9 (Example: Dow Jones Index)**.** Consider the following time series of the Dow Jones index.

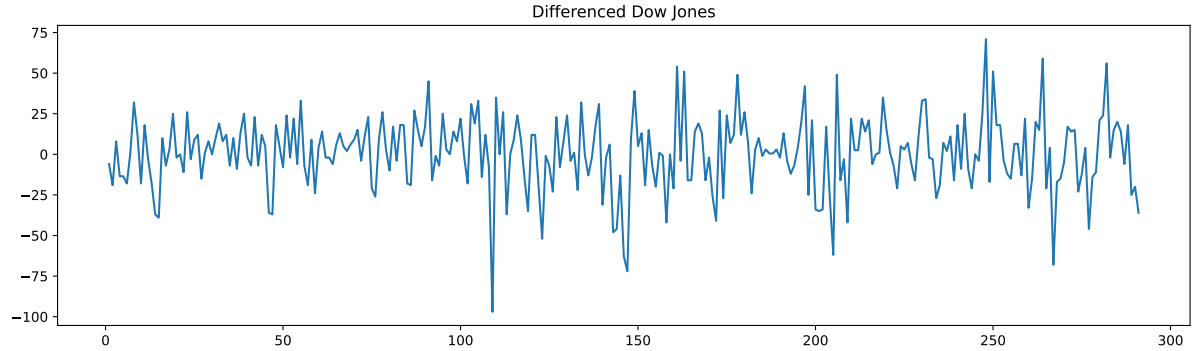
```
dj = pd.read_csv('data/dj.csv')
dj.plot(legend=False, title='Dow Jones Index', figsize=(15,4));
```



However, the following differenced version of the same series is:

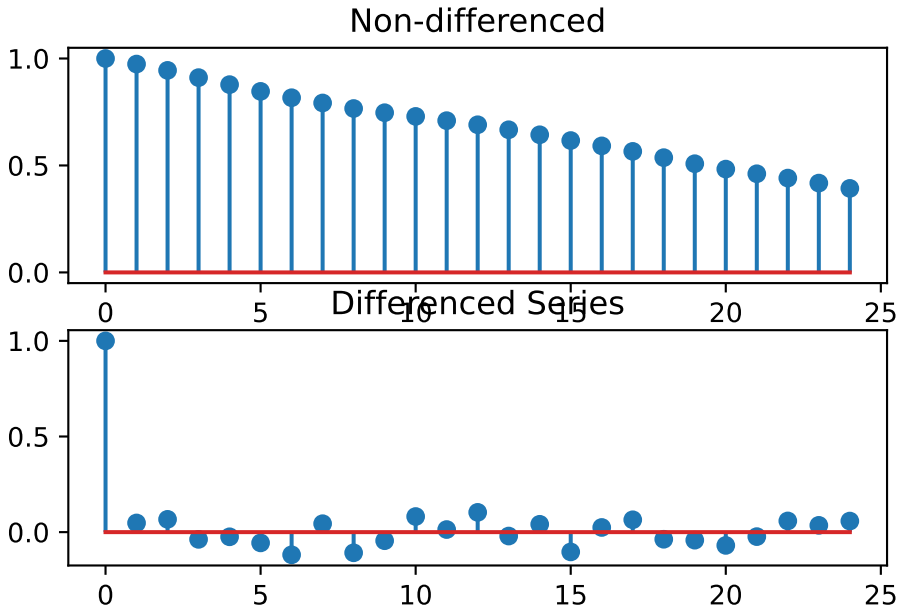
$$\Delta y_t = y_t - y_{t-1} \tag{6.9}$$

```
ddj = diff(dj)
ddj.plot(legend=False, title='Differenced Dow Jones', figsize=(15,4));
```



ARIMA models revolve around the idea that if we have a non-stationary series, we can transform it into a stationary one with a suitable number of differencing. A more general way of studying if a series is stationary is to plot it's AutoCorrelation Function (ACF). The ARIMA method directly models the ACF. That is why it is so important for this class of models. The ACF of a stationary process should "die down" quickly. Here is the ACF of the Dow Jones data, before and after differencing.

```
plt.subplot(211)
plt.stem(acf(dj, fft=False))
plt.title("Non-differenced")
plt.subplot(212)
plt.stem(acf(ddj, fft=False));
plt.title("Differenced Series");
```



Now suppose that, starting from our original series y_t, we difference it a sufficient number of times and obtain a stationary series. Let's call this y'_t. The ARIMA model assumes that

$$y'_t = c + \phi_1 y'_{t-1} + \phi_2 y'_{t-2} + \cdots + \phi_p y'_{t-p} + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q} \tag{6.10}$$

- The e_j correspond to unobserved innovations. They are typically assumed to be independent across time with a common variance.
- The ϕ and θ terms are unknown coefficients to be estimated.
- If y'_t was obtained by performing d successive differencings, then the above ARIMA model is referred to as an ARIMA(p, d, q) model.

In olden days, the p, d and q parameters were picked by the analyst after inspecting the ACF, PACF, and time plots of the differenced series. Today, we can iterate through a large number of them and pick the best according to a well-established criteria (AIC).

Example 6.10 (Example: Auto ARIMA on Aus Electricity)**.** Let us try out the ARIMA auto-fitting algorithm on the qau electricity usage dataset. We shall set aside the last three years of data as the test set. Recall that this is quarterly data. First, we establish the seasonal naive benchmark error for this dataset.

```
train2 = qau.kWh[:-12]
test2 = qau.kWh[-12:]

snaive_f = ts.snaive(train2, 12, 4)
print(f"The RMSE is approximately {ts.rmse(test2.values, snaive_f.values):.3f}.")
```

```
The RMSE is approximately 2.545.
```

Let us see if the automatically fitted ARIMA models can do better than this.

Here is the summary of the final chosen model.

```
#arima_m1 = pm.auto_arima(train2.values, seasonal=True, m=4, test='adf',
#                          trace=False, suppress_warnings=True)
arima_m1.summary()
```

Dep. Variable:	y	**No. Observations:**	206
Model:	SARIMAX(2, 1, 2)x(1, 0, [1], 4)	**Log Likelihood**	-175.320
Date:	Mon, 22 Sep 2025	**AIC**	364.640
Time:	23:31:44	**BIC**	387.901
Sample:	0	**HQIC**	374.049
	- 206		
Covariance Type:	opg		

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
ar.L1	-0.1675	0.613	-0.273	0.785	-1.369	1.034
ar.L2	0.3262	0.418	0.781	0.435	-0.492	1.145
ma.L1	-0.1472	0.604	-0.244	0.807	-1.330	1.036
ma.L2	-0.6210	0.589	-1.054	0.292	-1.776	0.534
ar.S.L4	0.9889	0.009	116.030	0.000	0.972	1.006
ma.S.L4	-0.6195	0.085	-7.310	0.000	-0.786	-0.453
sigma2	0.3110	0.026	12.139	0.000	0.261	0.361

Ljung-Box (L1) (Q):	0.19	**Jarque-Bera (JB):**	21.89
Prob(Q):	0.67	**Prob(JB):**	0.00
Heteroskedasticity (H):	9.53	**Skew:**	-0.09
Prob(H) (two-sided):	0.00	**Kurtosis:**	4.59

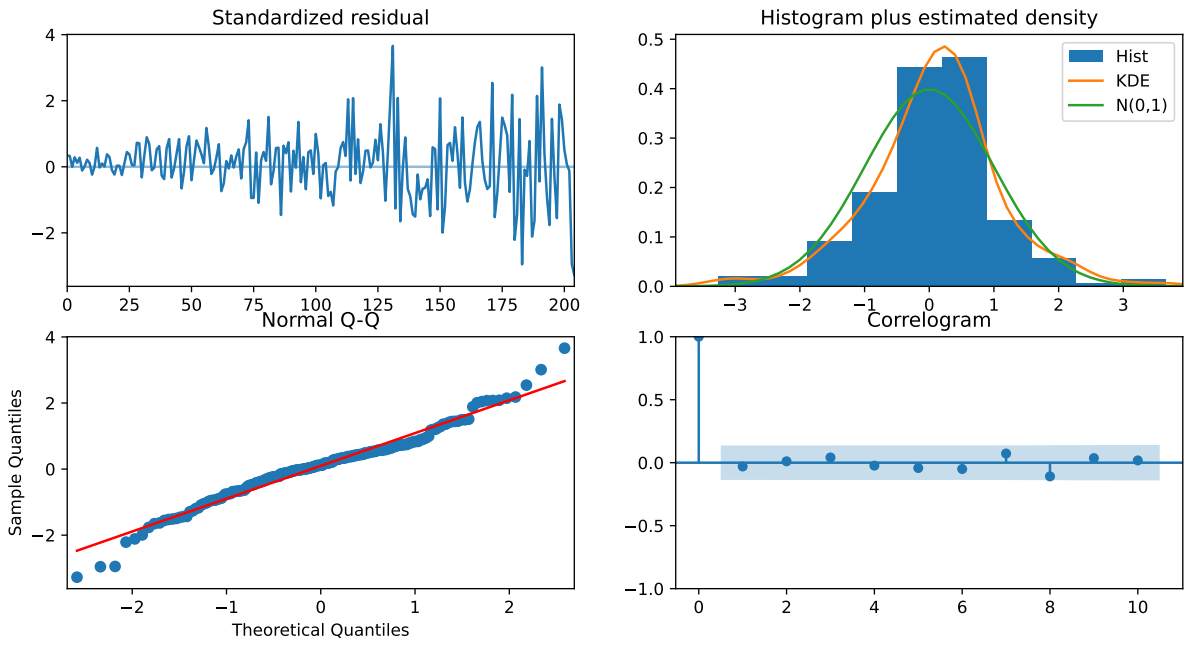
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Once we have found the best-fitting model, we do what we do with any other model in data analytics: we have to inspect the residuals. The residuals should look like trash to us - there should be no clues about the data in them. The standardized residual in the top left should be consistently wide; it isn't. This suggests some sort of transformation of the data before modeling might be appropriate.

The two plots on the off-diagonal are meant for us to assess if the residuals are Normally distributed with mean 0. They do indeed look like it.

The final plot, in the bottom right, displays an ACF with no spikes, indicating that the residuals are uncorrelated. This is precisely what we wished to see.

```
arima_m1.plot_diagnostics(figsize=(12,6));
```



Finally, we assess the error on the test set. The out-of-sample performance is better than the naive methods.

```
print(f"The RMSE on the test set is {ts.rmse(test2.values, arima_m1.predict(n_periods=12)):.3f
print(f"The MASE on the test set is {ts.mase(test2.values, arima_m1.predict(n_periods=12), tra
```

```
The RMSE on the test set is 1.929.
The MASE on the test set is 1.245.
```

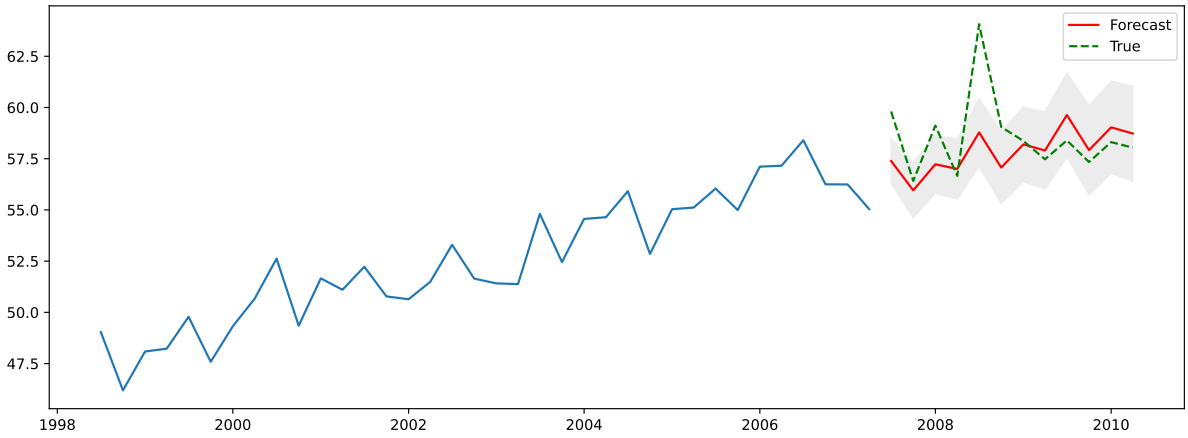
```
/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/utils/de

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/utils/de

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
```

Let us proceed to make a plot of the predictions.

```
n_periods = 12
fc, confint = arima_m1.predict(n_periods=n_periods, return_conf_int=True)

ff = pd.Series(fc, index=test2.index)
lower_series = pd.Series(confint[:, 0], index=test2.index)
upper_series = pd.Series(confint[:, 1], index=test2.index)

plt.figure(figsize=(14, 5))
plt.plot(train2[-36:])
plt.plot(ff, color='red', label='Forecast')
```

```
plt.fill_between(lower_series.index, lower_series, upper_series, color='gray', alpha=.15)
plt.plot(test2, 'g--', label='True')
plt.legend();
```

```
/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/utils/de

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/utils/de

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
```



6.5 ETS

ETS models are a new class of models, that define a time series process according to a set of state space equations:

$$y_t = w' x_{t-1} + e_t \tag{6.11}$$
$$x_t = F x_{t-1} + g e_t \tag{6.12}$$

The unobserved **state** x_t is usually a vector, and it typically contains information about * the level of the process at time t. * the seasonal effect at time t * the trend or growth factor at time t.

All these parameters are updated at every point in time. For instance, the simplest ETS model (A,N,N), is one that assumes that there is only a level parameter, and that it is updated as a weighted average of the most recent levels:

$$l_t = \alpha y_t + (1-\alpha) l_{t-1} \tag{6.13}$$
$$\hat{y}_{t+1} = l_t \tag{6.14}$$

The definition of the model is very flexible, and allows multiplicative growth, linear growth and damped terms in the model. All this, in addition to not requiring the assumption of stationarity!

Here is a full table of the additive models, followed by the multiplicative models:

Example 6.11 (Example: ETS model on Aus Electricity Dataset)**.** Let us try one of the basic models, with just a slope, level and seasonal effect ETS(A,A,A), on the qau dataset.

```
ets_m1 = ExponentialSmoothing(train2.values, seasonal_periods=4, trend='add', seasonal='add')
ets_fit = ets_m1.fit()
```

```
ets_fit.summary()
```

Dep. Variable:	endog	**No. Observations:**	206
Model:	ExponentialSmoothing	**SSE**	64.907
Optimized:	True	**AIC**	-221.915
Trend:	Additive	**BIC**	-195.292
Seasonal:	Additive	**AICC**	-220.787
Seasonal Periods:	4	**Date:**	Mon, 22 Sep 2025
Box-Cox:	False	**Time:**	23:31:45
Box-Cox Coeff.:	None		

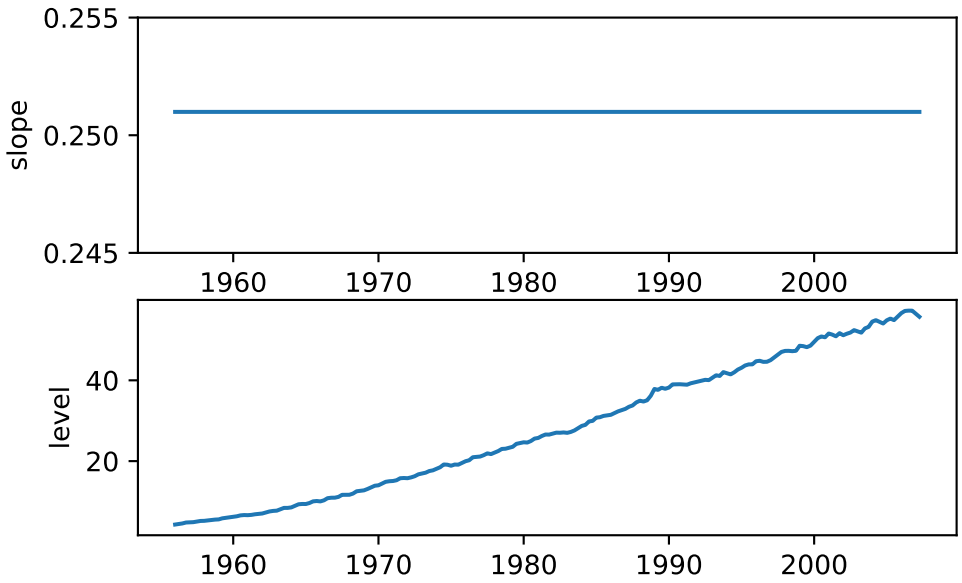
	coeff	code	optimized
smoothing_level	0.6205891	alpha	True
smoothing_trend	1.4395e-11	beta	True
smoothing_seasonal	0.2690339	gamma	True
initial_level	4.0245281	l.0	True
initial_trend	0.2509921	b.0	True
initial_seasons.0	-0.3901302	s.0	True
initial_seasons.1	0.1114788	s.1	True
initial_seasons.2	0.2913054	s.2	True
initial_seasons.3	-0.3722268	s.3	True

```
plt.subplot(211)
plt.plot(train2.index, ets_fit.trend, scaley=False)
plt.ylim(0.245, 0.255)
plt.ylabel('slope')

plt.subplot(212)
plt.plot(train2.index, ets_fit.level)
plt.ylabel('level');
```

Trend	Seasonal		
	N	A	M
N	$\mu_t = \ell_{t-1}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$	$\mu_t = \ell_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/\ell_{t-1}$
A	$\mu_t = \ell_{t-1} + b_{t-1}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$	$\mu_t = \ell_{t-1} + b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = (\ell_{t-1} + b_{t-1})s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + b_{t-1})$
A_d	$\mu_t = \ell_{t-1} + \phi b_{t-1}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$	$\mu_t = \ell_{t-1} + \phi b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = \phi b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + \phi b_{t-1})$
M	$\mu_t = \ell_{t-1}b_{t-1}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t/\ell_{t-1}$	$\mu_t = \ell_{t-1}b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1} + \beta\varepsilon_t/(s_{t-m}\ell_{t-1})$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1}b_{t-1})$
M_d	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/\ell_{t-1}$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1}^{\phi} + \beta\varepsilon_t/(s_{t-m}\ell_{t-1})$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1}b_{t-1}^{\phi})$

Trend	Seasonal		
	N	A	M
N	$\mu_t = \ell_{t-1}$ $\ell_t = \ell_{t-1}(1+\alpha\varepsilon_t)$	$\mu_t = \ell_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \alpha(\ell_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}(1+\alpha\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
A	$\mu_t = \ell_{t-1} + b_{t-1}$ $\ell_t = (\ell_{t-1} + b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$	$\mu_t = \ell_{t-1} + b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = (\ell_{t-1} + b_{t-1})s_{t-m}$ $\ell_t = (\ell_{t-1} + b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
A_d	$\mu_t = \ell_{t-1} + \phi b_{t-1}$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$	$\mu_t = \ell_{t-1} + \phi b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m}$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1+\alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
M	$\mu_t = \ell_{t-1}b_{t-1}$ $\ell_t = \ell_{t-1}b_{t-1}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}(1+\beta\varepsilon_t)$	$\mu_t = \ell_{t-1}b_{t-1} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1} + \alpha(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1} + \beta(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma(\ell_{t-1}b_{t-1} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}(1+\beta\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$
M_d	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}^{\phi}(1+\beta\varepsilon_t)$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi} + s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi} + \alpha(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1}^{\phi} + \beta(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t/\ell_{t-1}$ $s_t = s_{t-m} + \gamma(\ell_{t-1}b_{t-1}^{\phi} + s_{t-m})\varepsilon_t$	$\mu_t = \ell_{t-1}b_{t-1}^{\phi}s_{t-m}$ $\ell_t = \ell_{t-1}b_{t-1}^{\phi}(1+\alpha\varepsilon_t)$ $b_t = b_{t-1}^{\phi}(1+\beta\varepsilon_t)$ $s_t = s_{t-m}(1+\gamma\varepsilon_t)$

(a) Forecasting: Principles and Practice



Finally, let's take a look if the predictions on the test set are competitive with the automatic ARIMA model that was selected earlier. The forecasts are better than the naive ones, but not as good as the ARIMA. However, keep in mind that we are working with a single model; we should iterate over several ETS models and pick the best one in order to compare with the ARIMA model found.

```
ets_fc = ets_fit.forecast(steps=12)
print(f"The RMSE on th tests set is {ts.rmse(test2.values, ets_fc):.3f}.")
```

```
The RMSE on th tests set is 2.237.
```

6.6 Theta Method

The Theta method is a forecasting approach that has proved to be quite effective in competitions. It regularly appears as one of the best-performing models. Instead of decomposing a time series into Trend, Season and Remainder terms, the theta method decomposes a *seasonally adjusted* time series into **short-** and **long-term** components. The full reference for this method is Assimakopoulos and Nikolopoulos (2000).

Suppose we are interested in forecasting a time series $\{y_t\}$. The theta decomposition is based on the concept of amplifying/diminishing the local curvatures of the time series. This is brought about through a coefficient θ; hence the name of the model.

As a preview, we are going to find series $\{x_{\theta,t}\}$ and $\{x_{1-\theta,t}\}$ such that

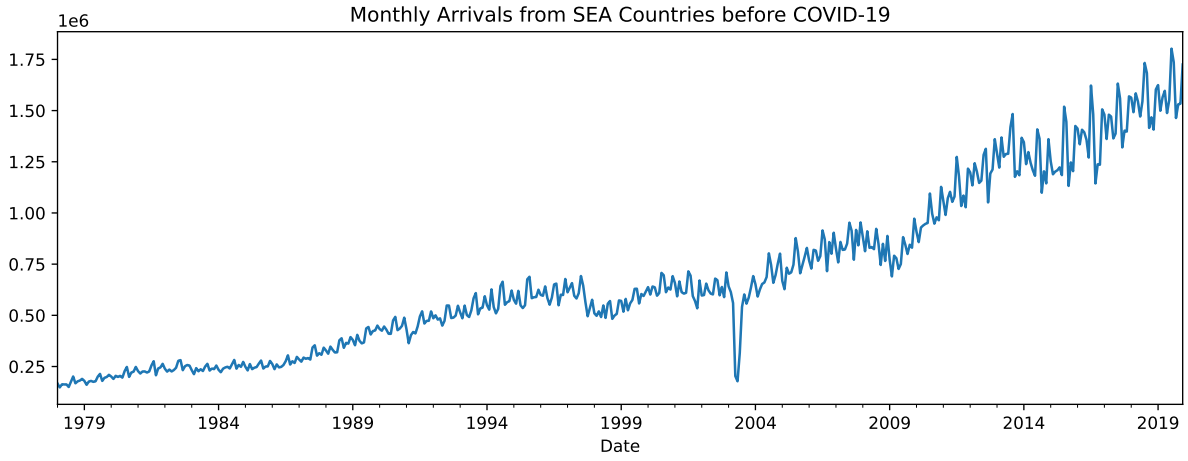
$$y_t = \frac{1}{2}(x_{\theta,t} + x_{1-\theta,t})$$

In order to forecast y_t, we shall forecast $\{x_{\theta,t}\}$ and $\{x_{1-\theta,t}\}$ separately and then combine them. There are numerous possible pairs of $\{x_{\theta,t}\}$ and $\{x_{1-\theta,t}\}$ that we can use. However the most common one is where x_θ corresponds to a straight line OLS fit, and $x_{1-\theta}$ corresponds to a version of the time series with its curvature amplified.

6.6.1 International Tourism Arrivals to Singapore

The Singapore Department of Statistics shares information on tourist arrivals to Singapore at the monthly level. Here is a time series plot of arrivals from South East Asian Countries:

```
tourism = pd.read_excel("data/international_visitor_arrivals_sg.xlsx",
                        parse_dates=[0], date_format="%Y %b",
                        na_values='na')
sea_tourism = tourism.iloc[:, 0:9]
sea_tourism.columns = ['Date', 'SEA', 'Brunei', 'Indonesia' ,'Malaysia',
                       'Myanmar', 'Philippines', 'Thailand', 'Vietnam']
sea_tourism.Date = sea_tourism.Date.str.replace('  ', '')
sea_tourism.Date = pd.to_datetime(sea_tourism.Date, format="%Y %b")
sea_tourism.set_index('Date', inplace=True)
sea_tourism.sort_index(inplace=True)
sea_tourism.index.freq = 'MS'

sea2 = sea_tourism.SEA[sea_tourism.index < datetime.datetime(2020, 1, 1)]
fig = sea2.plot(figsize=(12,4));
fig.set_title('Monthly Arrivals from SEA Countries before COVID-19');
```




```
theta_mod = ThetaModel(sea2)
theta_mod_fit = theta_mod.fit()
theta_mod_fit.summary()
```

Dep. Variable:	SEA
Method:	OLS/SES
Date:	Mon, 22 Sep 2025
Time:	23:31:46
Sample:	01-01-1978
	- 12-01-2019

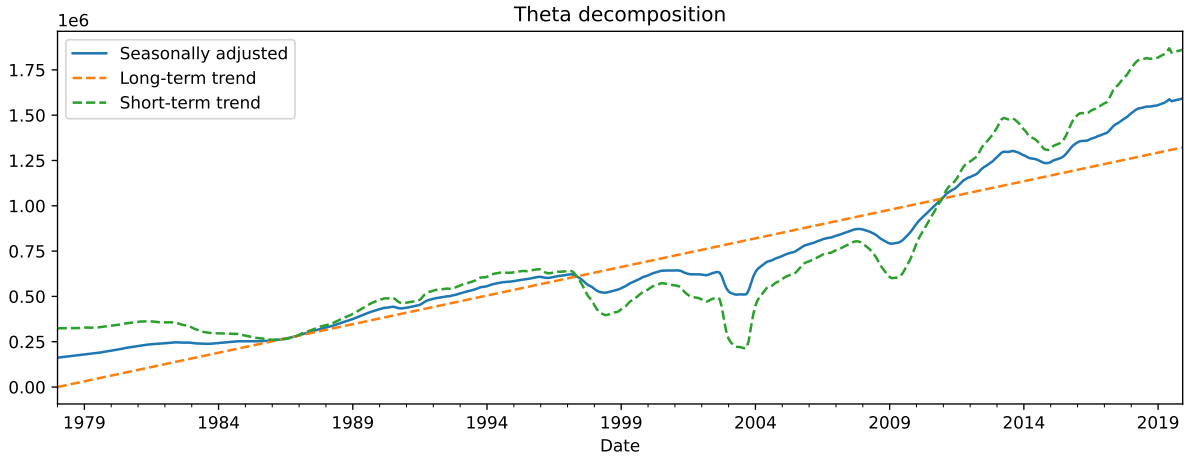
Parameters		
	b0	2628.8595906871597
	alpha	0.8383848338839772

Here is what the two components from the model look like, in comparison to the seasonally adjusted series (based on a multiplicative decomposition).

> **Note**
>
> The calculations in the next cell are based on formulas for the theta decomposition that we won't get into. They are just for illustration of how the decomposition is in terms of long and short term trends, rather than the cross-sectional decompositions we have been dealing with so far.

```
# carry out mutliplicative decomposition in order to obtain seasonally adjusted series
sea2_mult = seasonal_decompose(sea2, model='multiplicative', extrapolate_trend='freq')

# get long-term trend slope and intercept
a0 = (sea2.mean() - theta_mod_fit.params['b0']*(503)/2)/1e6
xvals = np.arange(1, 505)
yvals1 = a0 + theta_mod_fit.params['b0']*(xvals - 1)
yvals1 = pd.Series(data=yvals1, index=sea2.index)

# get short term trend series
a2 = -1*a0
b2 = -1*theta_mod_fit.params['b0']
yvals2 = a2 + b2*(xvals - 1) + 2*(sea2_mult.resid + sea2_mult.trend)
yvals2 = pd.Series(data=yvals2, index=sea2.index)
```

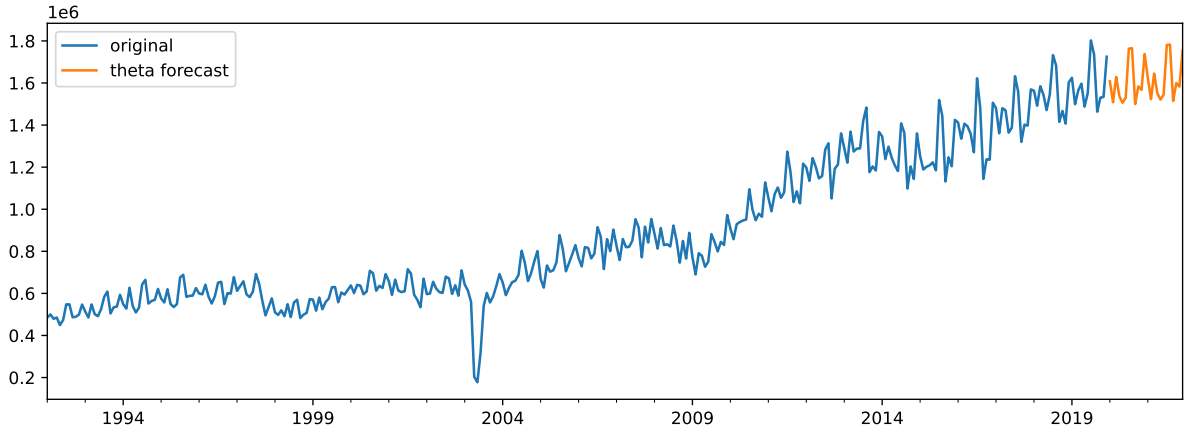
```
fig = (sea2_mult.resid + sea2_mult.trend).plot(label='Seasonally adjusted',
                                               legend=True, figsize=(12,4));
yvals1.plot(legend=True, label='Long-term trend', style="--")
yvals2.plot(legend=True, label='Short-term trend', style="--");
fig.set_title("Theta decomposition");
```



Finally, we visualise the forecasts from this theta decomposition model.

```
theta_forecasts = pd.DataFrame(
    {
        "original": sea2,
        "theta forecast": theta_mod_fit.forecast(24)
    }
)
```

```
theta_forecasts.tail(360).plot(figsize=(12,4));
```



6.7 Forecasting with Seasonal Decomposition

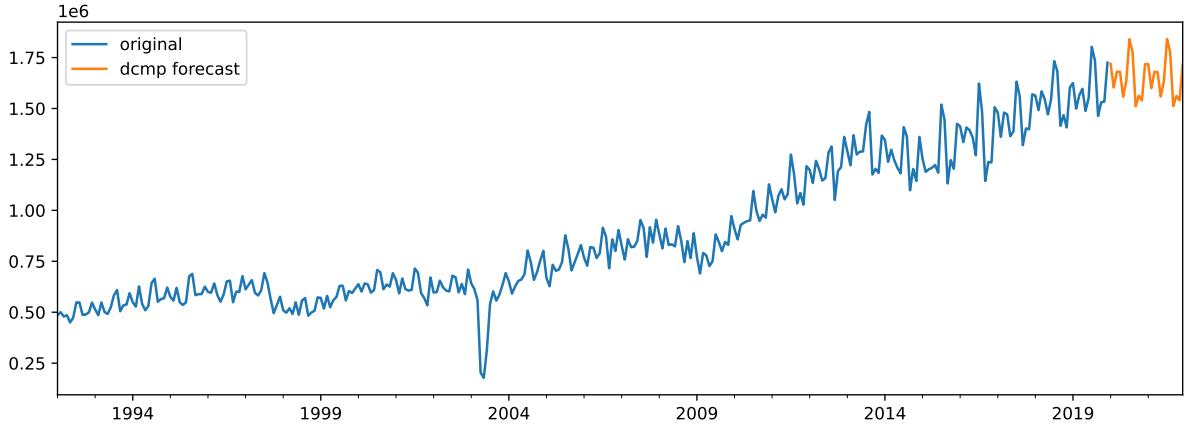
Continuing with the tourism data, we demonstrate how we can utilise a seasonal decomposition (not theta decomposition) model to make forecasts. Recall that a seasonal decomposition breaks the series down into the trend, seasonal and residual components. Seasonal decompositions are typically used to *study* a time series, but they can also be used to make forecasts. One approach is to use either ARIMA or ETS to predict the seasonally adjusted component, and then to use a seasonal naive model to predict the seasonal component. These two forecasts will then be combined to create the final forecast.

```
stlf = STLForecast(sea2, ARIMA, model_kwargs={"order": (3, 1, 2)})
res = stlf.fit( )
```

```
/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/statsmodels/tsa/

Non-stationary starting autoregressive parameters found. Using zeros as starting parameters.
```

```
forecasts2 = pd.DataFrame(
    {
        "original": sea2,
        "dcmp forecast": res.forecast(24)
    }
)
```

```
forecasts2.tail(360).plot(figsize=(12,4));
```



6.8 Miscellaneous Topics

6.8.1 Time Series Clustering

Example 6.12 (Example: Time Series Clustering, US Employment Data)**.** The Forecasting: Principles and Practice contains time series data on employment in various sectors in the US. There are 148 unique series in the dataset.

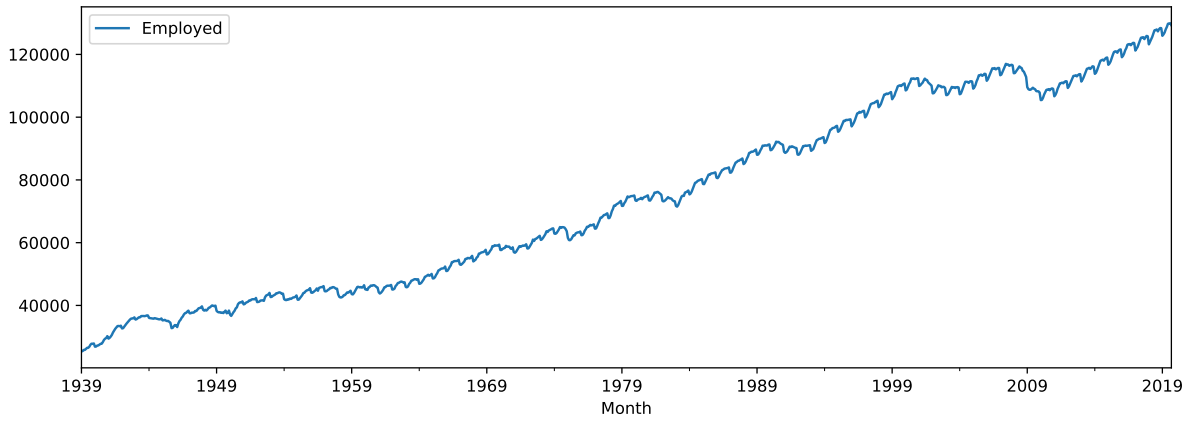
```
us_employment = pd.read_csv("data/us_employment.csv", parse_dates=[0], date_format="%Y %b")

series_ids = us_employment.Series_ID.unique()
print(f"There are {len(series_ids)} unique series in the dataset.")
#us_employment.Month.set_index

us_employment.sample(n=8)
```

```
There are 148 unique series in the dataset.
```

	Month	Series_ID	Title	Employed
109387	2010-08-01	CEU6561000001	Education and Health Services: Educational Ser...	2853.9
109731	1958-07-01	CEU6562000001	Education and Health Services: Health Care and...	NaN
145	1951-02-01	CEU0500000001	Total Private	40449.0
78492	1939-04-01	CEU5051500001	Information: Broadcasting, Except Internet	NaN
31661	1993-06-01	CEU3133600101	Durable Goods: Motor Vehicles and Parts	1075.5
50557	1953-02-01	CEU4244100001	Retail Trade: Motor Vehicle and Parts Dealers	NaN
140499	2019-04-01	CEU9093161101	Government: Local Government Education	8312.0
70635	2011-04-01	CEU4348600001	Transportation and Warehousing: Pipeline Trans...	42.8

Here is a plot of one of the time series in the dataset.

```
us_employment[us_employment.Title == "Total Private"].plot(x='Month',
                                                           y='Employed',
                                                           figsize=(12,4));
```



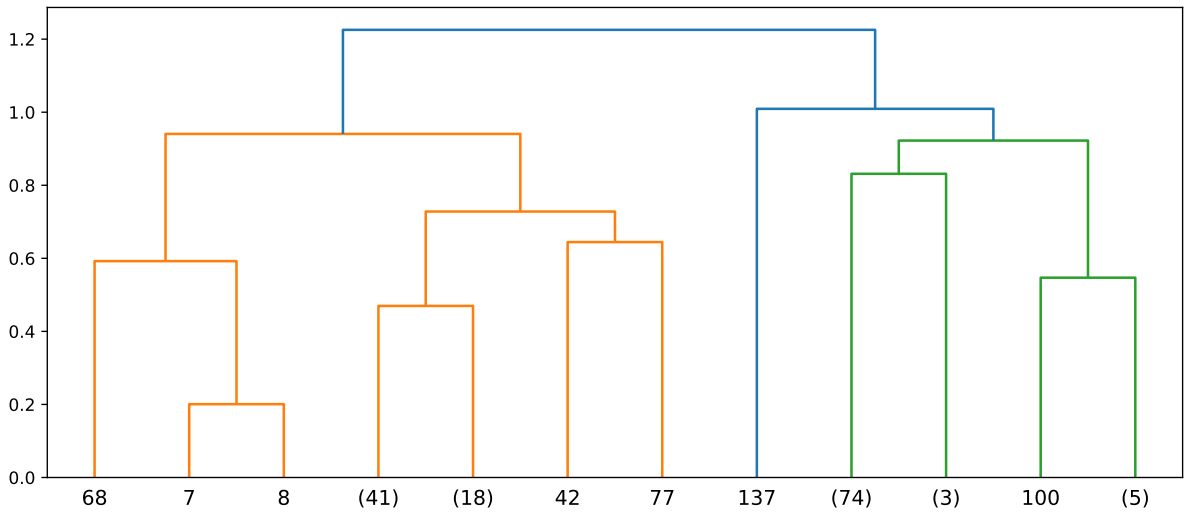
Since we are going to perform clustering on the time series data, our first step is to remove the missing values.

```
us_full = us_employment[(us_employment.Month >= datetime.datetime(2002, 9, 30)) & (us_employme
us2 = us_full.pivot(index='Series_ID', columns='Month', values="Employed")
us2_array = us2.to_numpy()
```

Now we can begin the clustering procedure.

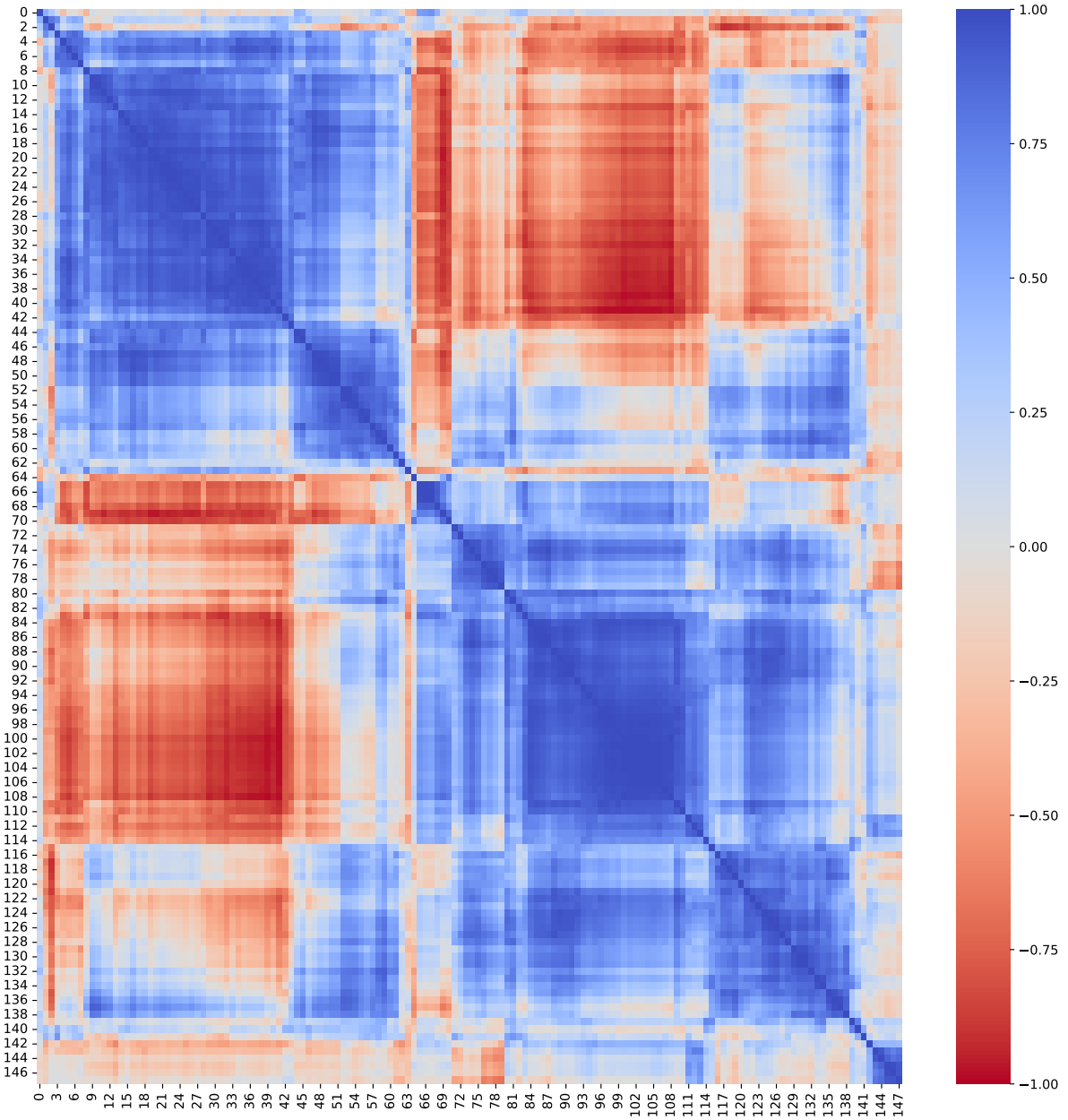
```
# This already converts the correlation into a distance (see the help page)
out = pdist(us2_array, metric='correlation')
#corr_mat = np.corrcoef(X, rowvar=False)
#dist_mat = (1 - corr_mat)/2
```

```
lm1 = hierarchy.linkage(out, method='average', optimal_ordering=True)

plt.figure(figsize=(12,5))
hierarchy.dendrogram(lm1, p=3, truncate_mode='level',color_threshold=True);
```



We reorder the columns and rows in the matrix so that similar time series appear next to one another.

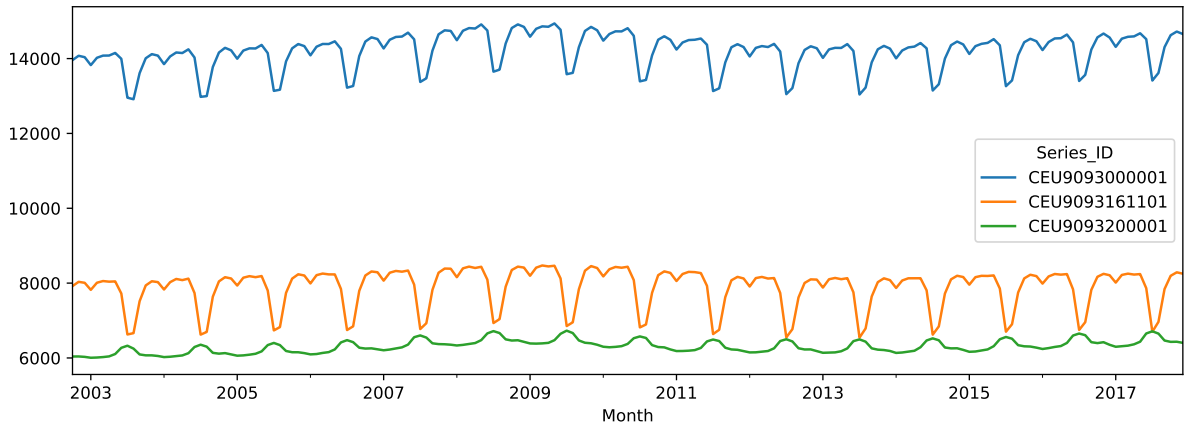
```
X_ord = us2_array[hierarchy.leaves_list(lm1)]
corr_mat_ord = np.corrcoef(X_ord)
#corr_mat_ord.shape
plt.figure(figsize=(15, 15))
sns.heatmap(corr_mat_ord, vmin=-1, vmax=1, cmap='coolwarm_r', center=0);
#, annot=True, cmap='coolwarm', center=0)
```



In the next few plots, we pick out series that are close to one another in blue segments of the matrix above to inspect how similar they are, and in what ways they are similar to one another.

```
us2_series = us2.index.to_list()
```

```
# Similar series set 1
# ss = [us2_series[x] for x in [22, 24, 32, 33, 34]]
ss = [us2_series[x] for x in [143, 144, 145]]
us2.T.loc[:, ss].plot(figsize=(12,4));

# Similar series set 2
#ss = us2_series[65:68]
#us2.T.loc[:, ss].plot(logy=False, figsize=(12,4)); # try with and without log

# Simular series set 3:

#ss = [us2_series[x] for x in [92, 94, 96, 97, 98]]
#us2.T.loc[:, ss].plot(logy=False, figsize=(12,4));
```



6.9 Summary

We have briefly covered the following time series concepts: Exploratory data analysis, assessing point forecasts and a few commonly used models. To finish up the section, we also applied clustering to time series. With today's code, the model-fitting routines are easier to run. As always, the onus is on us as analysts to inspect the residuals from the models to understand the time series we are working with.

6.10 References

6.10.1 Stats models pages

1. Main page
2. Forecasting with statsmodels
3. Theta model

6.10.2 Forecasting principles and practice

Although the code for this textboook uses R, the concepts are very well explained. It is written by one of the foremost experts in time series forecasting methods. The reference is Hyndman and Athanasopoulos (2018).

- Forecasting: Principles and Practice

7 Simulation

7.1 Random Variables

In statistics, we use random variables to describe the probabilistic behaviour of phenomenon. Random variables are real numbers that represent the phenomena we observe. For instance, we might let X represent a coin toss, with $X = 1$ representing Heads and $X = 0$ representing Tails.

Random variables come with a rule (or function) that prescribes the probabilities with which it takes on particular values. For instance, if we had a fair coin, then the rule would be that

$$P(X = 1) = P(X = 0) = \frac{1}{2}$$

On the other hand, a biased coin might follow the rule

$$P(X = 1) = \frac{2}{3}, \quad P(X = 0) = \frac{1}{3}$$

This rule tells us which events are more likely, and which are less likely.

```
from math import pi
from IPython.display import IFrame,Video

import mesa

import seaborn as sns
import matplotlib.pyplot as plt

import nltk
from nltk import word_tokenize

from numpy.random import default_rng
import numpy as np
import pandas as pd
from scipy import stats
from scipy.stats import binom, bernoulli, norm, expon, uniform
```

7.1.1 Discrete vs. Continuous Random Variables.

Discrete random variables take on only a countable number of values. Examples are:

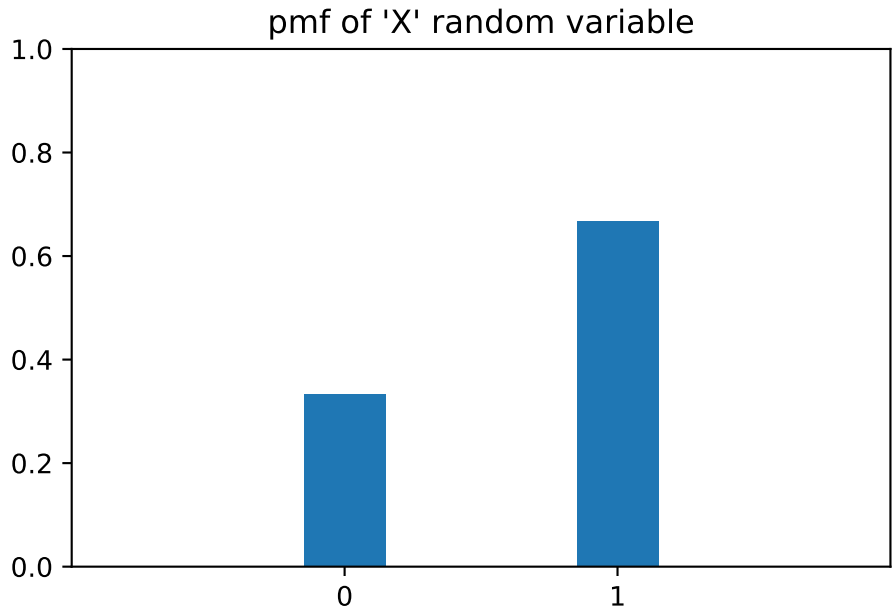
- A coin toss ($\{0, 1\}$)

- The number of taxis passing by a particular junction between 12noon and 1pm. ($\{0, 1, 2, \dots, \}$)
- The number of coin tosses until we observe Heads. ($\{1, 2, 3, \dots, \}$)

Discrete random variables are defined by their probability mass function (pmf), which is just a table or a function describing $P(X = i)$ for all possible i values. Once we know the pmf of a random variable, we know everything about it - the mean, variance, quantiles, maximum values, etc.

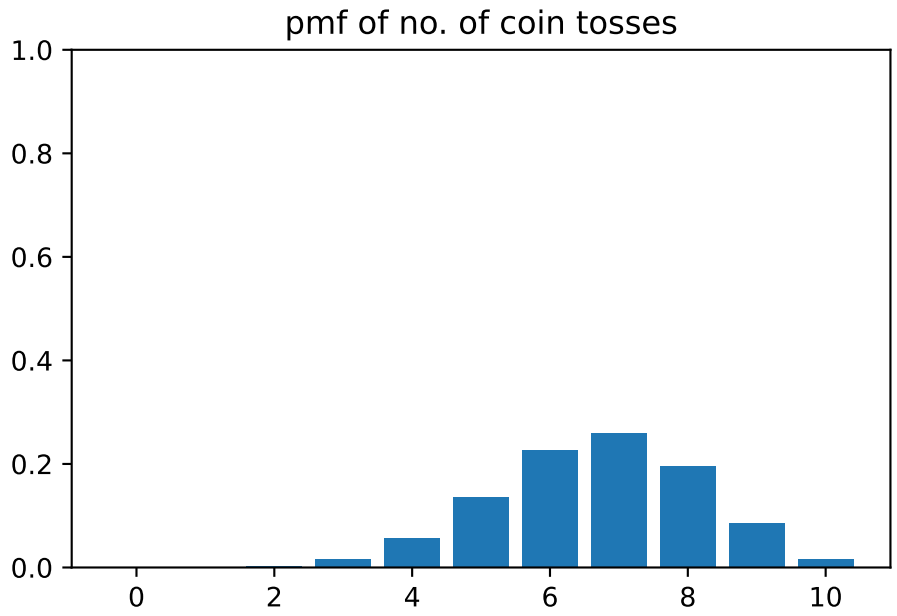
We can visualise a pmf using a bar-chart. Here is the pmf for the above biased coin.

```
plt.bar([0,1], [1/3, 2/3], tick_label=['0', '1'], width=0.3);
plt.xlim(-1,2);
plt.ylim(0,1);
plt.title('pmf of \'X\' random variable' );
```



Here is the pmf for a random variable representing the total number of Heads after 10 tosses of that same coin. Suppose we call that new random variable Y.

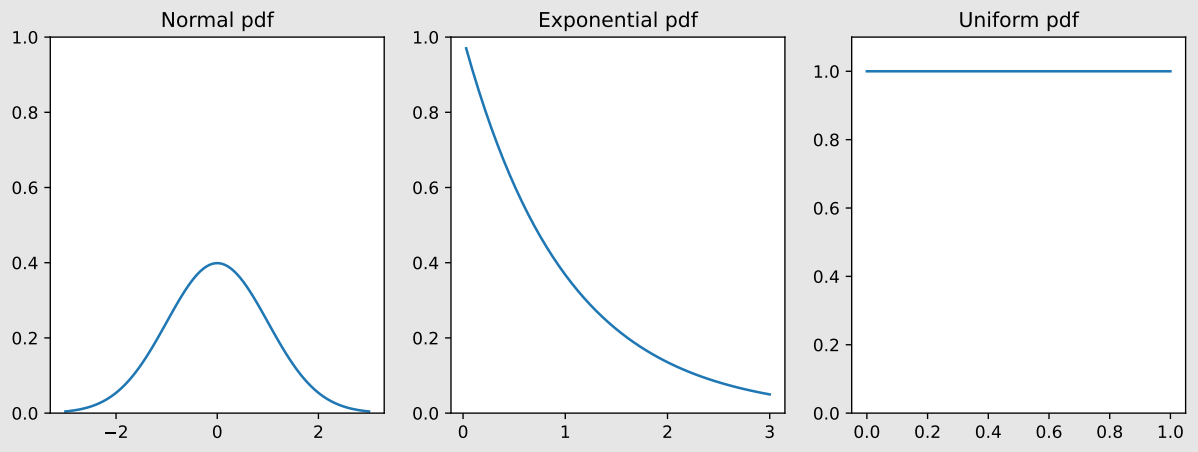
```
probs = binom.pmf(np.arange(0, 11), n=10, p=2/3)
plt.bar(np.arange(0, 11), probs);plt.ylim(0,1);
plt.title('pmf of no. of coin tosses');
```



If we wish to find the probability of events, for instance, $P(Y \leq 4)$, we sum up the heights of the bars.

Continuous random variables are defined by what is known as a probability density function. Continuous random variables can take on an uncountable number of values, for instance, all real numbers in the interval $[0, 1]$.

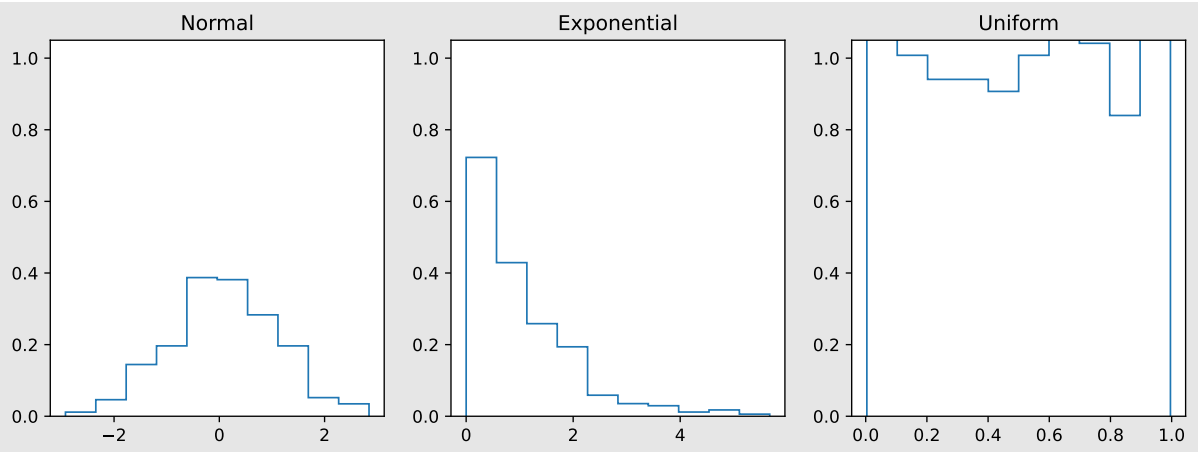
```
x = np.linspace(-3, 3, num=100)
y = norm.pdf(x)
plt.figure(figsize=(12,4), facecolor='0.9')
plt.subplot(131)
plt.plot(x,y)
plt.ylim(0,1)
plt.title('Normal pdf')

y = expon.pdf(x[x>0])
plt.subplot(132)
plt.plot(x[x>0],y)
plt.ylim(0,1)
plt.title('Exponential pdf');

x = np.linspace(0, 1, num=100)
y = uniform.pdf(x)
plt.subplot(133)
plt.plot(x,y)
plt.ylim(0,1.1)
plt.title('Uniform pdf');
```



7.1.2 Generating Random Variates

With a computer, we can generate random variables from almost any distribution. There are built-in routines to generate from the 'named' distributions. For instance,

```
rng = default_rng(5003)

yvals = np.zeros((3,300))
yvals[0,:] = rng.normal(size=300)
yvals[1,:] = rng.exponential(size=300)
yvals[2, :] = rng.uniform(size=300)
pdfs = ['Normal', 'Exponential', 'Uniform']
plt.figure(figsize=(12,4), facecolor='0.9')

for i in np.arange(0,3):
    plt.subplot(1, 3, i+1)
    plt.hist(yvals[i,:], density=True, histtype='step')
    plt.title(pdfs[i])
    plt.ylim(0, 1.05)
```



Notice how the (normalised) histograms look a lot like the corresponding pdfs.

7.2 General Principles in Simulation Studies

7.2.1 Introduction

The objective of any simulation study is to estimate an expectation $E(X)$. Simulation studies involve the use of a computer to generate independent copies of the random variable of interest X. Here are a couple of examples where simulation studies would be applicable.

Through the estimation of $E(X)$, simulation studies allow us to:

1. Consider different planning scenarios and estimate the throughput from them.
2. Study the emergent behaviour of complex situations.

Example 7.1 (Example: Insurance Claims)**.** Before the financial year begins, an insurance company has to decide how much cash to keep, in order to pay out the claims for that year. Suppose that claims are independent of each other and are distributed as $Exp(1/200)$ dollars. This means that the probability density function (pdf) of the random variable is $f_X(x) = \frac{1}{200}\exp(-x/200),\ x > 0$

Also suppose that the number of claims in a year is a Poisson random variable with mean 8.2.

An actuary has been asked to determine the size of the reserve fund that should be set up, and he recommends \$12,000. We might consider answering the following question using simulation:

- What is the probability that the total claims will exceed the reserve fund?

If we let Y be the random variable representing the total sum of claims, we are interested in estimating $P(Y > 12000)$. Since probabilities are expectations, we can use simulation to estimate this value.

Example 7.2 (Example: Sandwich Shop Closing Time)**.** Here is a slightly more sophisticated example.

Suppose that you run a sandwich shop, which is open from 9am till 5pm. Your philosophy has always been to serve every customer who has entered before 5pm, even if that requires you to stay back until they have been served. You would like to estimate the mean amount of overtime you have to work.

If you are willing to assume that the inter-arrival times of customers is $Exp(3)$ hours, then it is possible to simulate this process to estimate the mean time that you would have to remain open, beyond 5pm.

7.2.2 Steps in a Simulation Study

The two examples above are known as Discrete Event Simulations. It is one type of simulation study. Another type of simulations are Agent-Based Models. No matter the type of simulation, the basic steps in a simulation study are:

1. Identify the random variable of interest and write a program to simulate it.
2. Generate an iid sample $X_1, X_2, \ldots, X_n$ using this program.
3. Estimate $E(X)$ using $\bar{X}$.

Before proceeding, let us refresh our knowledge of the properties of the sample mean.

7.2.3 Theory

There are two important theorems that simulation studies rely on. The first is the Strong Law of Large Numbers (SLLN).

Strong Law of Large Numbers

If $X_1, X_2, \ldots, X_n$ are independent and identically distributed with $E(X) < \infty$, then

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \to E(X) \quad \text{with probability 1.}$$

In the simulation context, it means that as we generate more and more samples (i.e. increase n), our sample mean $\bar{X}$ converges to the desired value $E(X)$, *no matter what the distribution of X is.*

The second theorem that aids us is the Central Limit Theorem (CLT).

Central Limit Theorem

Let $X_1, X_2, \ldots, X_n$ be i.i.d., and suppose that

- $-\infty < E(X_1) = \mu < \infty$.
- $Var(X_1) = \sigma^2 < \infty$.

Then

$$\frac{\sqrt{n}(\bar{X} - \mu)}{\sigma} \Rightarrow N(0, 1)$$

where $\Rightarrow$ denotes convergence in distribution.

This is sometimes informally interpreted to mean that when n is large, $\bar{X}$ is approximately Normal with mean μ and variance σ^2/n. In the simulation context, we can use this theorem to obtain a confidence interval for the expectation that we are estimating.

Also take note of the following properties of the sample mean and variance:

Sample Estimates

It can be shown that both the sample mean and sample standard deviation are unbiased estimators.

$$E(\bar{X}) = E(X), \quad E(s^2) = \sigma^2$$

where $s^2 = \frac{\sum(X_i - \bar{X})^2}{n-1}$. :::

To obtain a $(1-\alpha)100$ confidence interval for μ, we use the following formula, from the CLT:

$$\bar{X} \pm z_{1-\alpha/2}\frac{s}{\sqrt{n}}$$

When our goal is to estimate a probability p, we have to introduce a corresponding indicator variable X such that

$$X = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

In this case, the formula for the CI becomes

$$\bar{X} \pm z_{1-\alpha/2}\sqrt{\frac{\bar{X}(1-\bar{X})}{n}}$$

7.3 Object-Oriented Programming in Python

Python has been developed as both a functional and object-oriented programming language. Much of the code we will soon use for the mesa package involves creation of an instance, and then accessing the attributes (data or methods) of that instance.

The syntax for *defining* a class is:

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

A class definition typically consists of function definitions and attributes. A special function within a class definition, `__init__()`, can be used to set the initial state of *instances* of a class. Think of a class definition as a template, and instances as copies made using that template.

Consider the following class Circle.

```
class Circle:
    """ A simple class definition

    c0 = Circle()
    c0.radius

    """
    def __init__(self, radius = 1.0):
        self.radius = radius

    def area(self):
        """ Compute area of circle"""
        return pi*(self.radius**2)
```

We can create multiple circles using this template. Note the use of the keyword `self` to refer to attributes of the instance.

```
c1 = Circle(3.2)
```

```
c2 = Circle(4.0)
c2.area()
```

```
50.26548245743669
```

```
print(f"""
The radius of c1 is {c1.radius} and the radius of c2 is {c2.radius}.
The area of c1 is {c1.area():.3f} and the area of c2 is {c2.area():.3f}.
""")
```

```
The radius of c1 is 3.2 and the radius of c2 is 4.0.
The area of c1 is 32.170 and the area of c2 is 50.265.
```

7.4 Introduction to Agent Based Models

Agent-based models (ABM) consist of individual elements (agents) interacting with one another. One of the most common uses of an agent-based model is to understand how the behaviour of the system, as a whole, emerges from simple rules for the agents' and their interactions. A very famous example is from the simulation software used to create battle scenes in The Lord of The Rings movies from the early 2000s. The following video was from an article on the MASSIVE software.

```
#IFrame(src='https://house-fastly-signed-us-east-1-prod.brightcovecdn.com/media/v1/pmp4/static
#       width=960, height=480)
Video("../data/lotr.mp4", width=960, height=480)
```

Today, ABM are used in scenarios where the overall behaviour of a system is to complicated to predict by modeling the stages of interaction. Examples of these scenarios are:

1. COVID-19 modeling,
2. Modeling of driver behaviour on roads for the purpose of understanding traffic,
3. Modeling of swarms, e.g. locusts, fish, etc, and
4. Modeling of effects of networks in marketing and in organisations.

The essential characteristics of an ABM are the following:

1. Agents at various scales,
2. Agents make decisions, based on defined behaviours
3. Agents interact with one another in time and possibly space.

Remember that our role as analysts is to imbue the agents with the appropriate traits (behaviours), interactions and probabilities. Otherwise, our model is not going to yield useful results.

7.4.1 Introduction to Mesa

Mesa is a Python package for running simulations using Agent-Based Models (ABM). Before we proceed, here are a few overarching concepts about this framework:

1. *Agents* are defined using classes. Within the class definition for an agent, we need to define:
    - how the agent is initialised,
    - what the agent does at each time-step.
2. The *model* is defined as a class as well. The model constitutes agents (defined through the class(es) above) and a *scheduler* that determines the sequence in which agents act, and the space in which they act.

Mesa also comes equipped with data collection tools to extract information from agents and/or the model at each step of the scheduler. Both the agent and the model classes should define a `step()` method. In ABM, a *step* is the smallest unit of time. It is also sometimes referred to as a "tick".

7.5 Boltzmann Model Overview

This is a simple model from social sciences, to demonstrate the use of mesa for agent-based modeling. In this agent-based economy, the assumptions are:

1. There are a certain number of agents.
2. All agents begin with 1 unit of money.
3. At every step of the model, an agent gives 1 unit of money (if they have it), to another, randomly chosen, agent.

The reference for this paper is Dragulescu and Yakovenko (2002). Before we begin the model, think about what you expect to see after some time.

7.6 Agent and Model Classes (v1)

The code below contains our initial definition for the agent class. The initialisation simply consists of setting the wealth for the agent. The `step` method defines that the agent will pick an agent (possibly itself) and transfer 1 unit of money to that agent.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def step(self):
```

```
        # Verify agent has some wealth
        if self.wealth > 0:
            other_agent = self.random.choice(self.model.agents)
            other_agent.wealth += 1
            self.wealth -= 1
```

The following flowchart depicts the flow of events for each agent. It is useful to sketch such a flowchart when implementing agent-based models, since it allows for easier debugging and information sharing with colleagues.

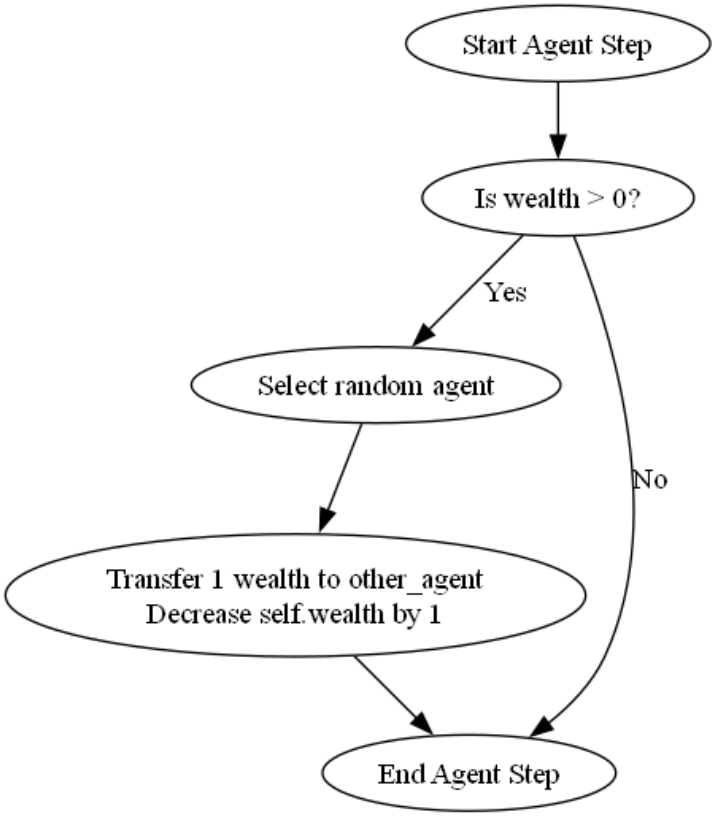


Figure 7.1: Agent trajectory

The next class defines the model itself. The initialisation of a model typically consists of populating it with various agents. However, it is also important to define the scheduler. In this case, the scheduler is based on `RandomActivation`. This means that *at every step*, the agents are shuffled, and then each of their `step` methods are executed in that order. The order changes at every time step. It is necessary to avoid running all agents' steps at the same time to avoid clashes.

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N):
        super().__init__()
        self.num_agents = N
        # Create scheduler and assign it to the model
```

```
        # self.schedule = mesa.time.RandomActivation(self)
        # self.agents.shuffle_do("step")

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(self)
            # Add the agent to the scheduler
            #self.schedule.add(a)

    def step(self):
        """Advance the model by one step."""

        # The model's step will go here for now this will call the step method of each agent a
        # self.schedule.step()
        self.agents.shuffle_do("step")
```

The next cell creates a model with 10 agents, and runs the model for 100 steps. At the end of this run, we compute the proportion of customers with wealth 0. This is a simple initial metric that we define on the model. Once again, just as with the parameters of the model, the metrics we extract determine how well we can use the model.

Remember that, to begin with, all agents had equal wealth. If this proportion is high, it indicates a severe imbalance in wealth distribution after 100 time steps.

```
all_wealth = []
model = MoneyModel(10)

# allows replicability of results; useful for debugging
model.reset_randomizer(seed=5003)

for i in range(100):
    model.step()

# Extract the results
# for agent in model.schedule.agents:
for agent in model.agents:
    all_wealth.append(agent.wealth)
```

```
prop = (np.array(all_wealth) == 0).mean()
print(f"The proportion of agents with zero wealth is {prop:.3f}.")
```

```
The proportion of agents with zero wealth is 0.300.
```

7.6.1 Data Collection

However, remember that when we perform simulations, we need to repeat the run multiple times, take the average, and then form a confidence interval. To prepare the mesa model to perform data collection, we write a function that, given a model, will compute the proportion of zero-wealth agents.

```
def compute_zero_prop(mesa_model):
    all_wealth = []
    # Extract the results
    for agent in mesa_model.agents:
        all_wealth.append(agent.wealth)
    prop = (np.array(all_wealth) == 0).mean()
    return prop

compute_zero_prop(model)
```

```
0.3
```

Next, we update the model class to define a data collector that calls the `compute_zero_prop()` function that we just defined. With Mesa, the data collector can be based on the model, or it could be a function that operates on individual agents.

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N):
        super().__init__()
        self.num_agents = N
        # Create scheduler and assign it to the model
        # self.schedule = mesa.time.RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(self)
            # Add the agent to the scheduler
            # self.schedule.add(a)

        # initialise the data collector, telling it to use the compute_zero_prop() function
        # on the model.
        self.datacollector = mesa.DataCollector(
            model_reporters={"zero_prop": compute_zero_prop}
        )

    def step(self):
        # self.schedule.step()
        self.agents.shuffle_do("step")
        # Collect data at every step
        self.datacollector.collect(self)
```

Once again, we reset the seed and re-run the model. We can then retrieve the zero-wealth proprortion at *every step of the model*.

```
model = MoneyModel(10)
model.reset_randomizer(seed=5003)
```

```
for i in range(100):
    model.step()
```

```
model.datacollector.get_model_vars_dataframe()
```

	zero_prop
0	0.4
1	0.5
2	0.6
3	0.6
4	0.5
...	...
95	0.1
96	0.5
97	0.5
98	0.3
99	0.3

Take note that this is considered *a single simulation*. Any simulation model requires time to reach an equilibrium phase. In this case, we are interested in measuring the average proportion of zeros after 100 time steps. To do so, we need to run several simulations, compute the zero-wealth proportion at each step and take the average.

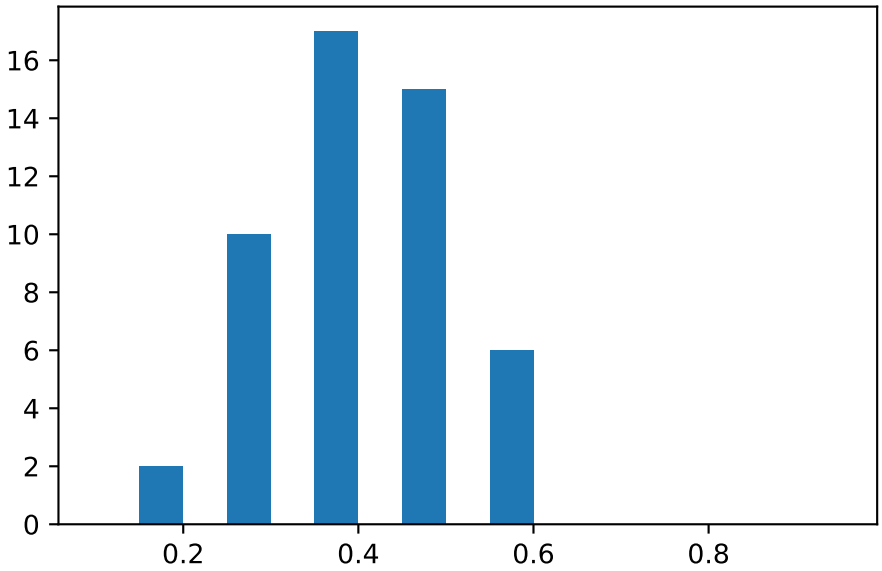
7.7 Multiple Iterations

The mesa package has features to help us to run multiple simulations and then to collate the results. The primary method is `.batch_run()`. Now we shall run the model for 50 iterations. Remember that each iteration consists of a fresh set of 100 time steps.

```
params = {"N": 10}
#params = {"N": [10, 50, 100]}

results = mesa.batch_run(
    MoneyModel,
    parameters=params,
    iterations=50,
    max_steps=100,
    number_processes=1,
    data_collection_period=-1,
    display_progress=True,
)
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

```
results_df = pd.DataFrame(results)

results_df.zero_prop.hist(grid=False, bins=np.arange(0.10, 1.00, 0.05));
# results_df.zero_prop.groupby(results_df.N).describe()
```




```
stats.ttest_1samp(results_df.zero_prop, 0.0).confidence_interval()
#results_df.zero_prop.groupby(results_df.N).apply(lambda x:
#                                                 stats.ttest_1samp(x, 0.0).confidenceinterva
```

```
ConfidenceInterval(low=0.39627070344688253, high=0.45572929655311745)
```

We can see that the proportion of agents with zero income is approximately 0.42. This is considerably higher than the 0.0 that the population began with.

7.8 Agent and Model Classes (v2)

7.8.1 Assessment of Income Inequality

The Gini coefficient is a common measure of the income inequality of a population. The Gini coefficient can be defined using a Lorenz curve for a population:

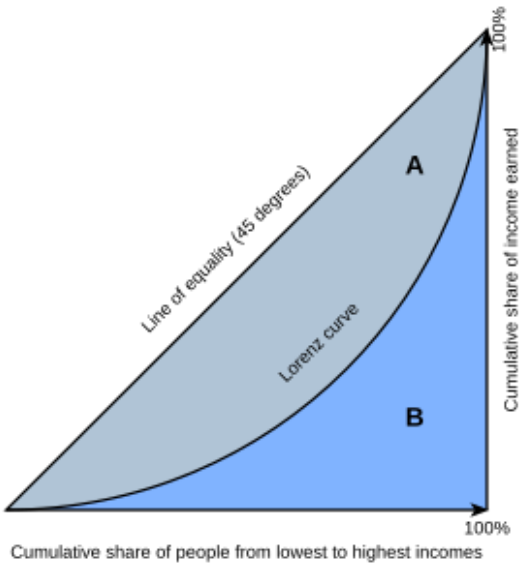


Figure 7.2: Lorenz curve

The x- and y-axes both run from 0 to 100%. The x-axis depicts the cumulative proportion of a population (increasing from left to right), ranked by income. The y-axis depicts the cumulative proportion of total income earned by the population. If all members earned an equal amount, the Lorenz curve would be a straight line, from bottom left to top right. Gini coefficient measures deviation from this equality as:

$$\frac{A}{A+B}$$

The Gini coefficient is a measure between 0 and 1. A value of 0 indicates income equality, while a value of 1 indicates that only 1 member of the population earns all the income (extreme inequality). Here is an interactive visualisation from Our World in Data on Gini coefficient values around the world, across time.

The wikipedia page on Gini coefficient contains a formula for computing Gini coefficient, given the income of all agents. After implementing it, we include it in the model class, along with our earlier metric.

```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B

class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N):
        super().__init__()
        self.num_agents = N
```

```
        # Create scheduler and assign it to the model
        # self.schedule = mesa.time.RandomActivation(self)

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(self)
            # Add the agent to the scheduler
            # self.schedule.add(a)

        # initialise the data collector, telling it to use the gini() and compute_zero_prop()
        # on the model.
        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini, "zero_prop": compute_zero_prop}
        )

    def step(self):
        # self.schedule.step()
        self.agents.shuffle_do("step")
        # Collect data at every step
        self.datacollector.collect(self)
```

7.8.2 Data Collection

Suppose we are interested to study how the Gini coefficient of our little population evolves over time. We shall use the `batch_run` to run 50 iterations, and then create a summary visualisation of the evolution of Gini coefficient for each iteration.

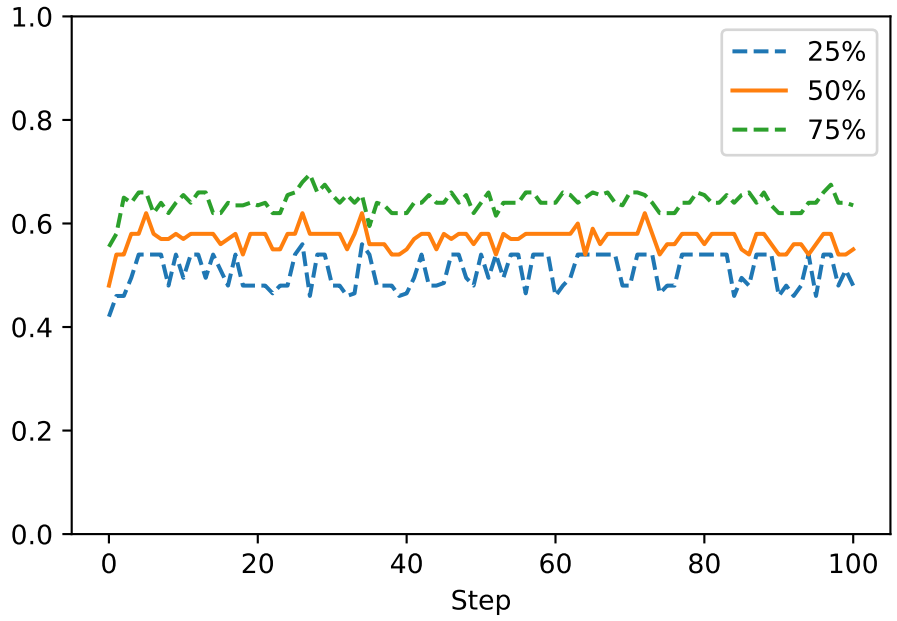
```
params = {"N": 10}

results = mesa.batch_run(
    MoneyModel,
    parameters=params,
    iterations=50,
    max_steps=100,
    number_processes=1,
    data_collection_period=1,
    display_progress=True,
)
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

We would like to inspect how the inequality varies over the time steps. Hence, at each time step, we obtain the lower and upper quantiles along with the median and plot them.

```
results_df = pd.DataFrame(results)
grp_by_step = results_df.Gini.groupby(results_df.Step).describe()
#grp_by_step.head()
```

```
grp_by_step[['25%', '50%', '75%']].plot(style=['--', '-', '--'])
plt.ylim([0, 1]);
```



We can see that the Gini index quickly achieves one of the highest inequality values that we observe in the world, and remains there.

7.9 Agent and Model Classes (v3)

7.9.1 Adding a spatial component

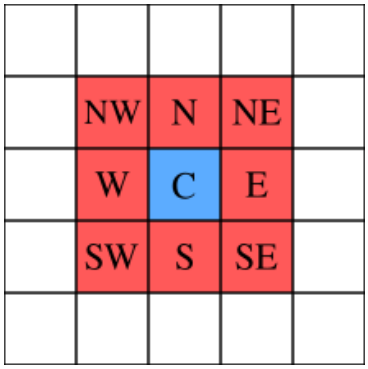
Next we introduce another critical component of agent-based modeling - the spatial component. We retain the Gini coefficient computation, but we are going to modify the model behaviour such that agents only exchange money if the other agent is nearby. The `MultiGrid` defines a rectangular grid, indexed by `[x, y]`, where `[0, 0]` is assumed to be at bottom-left and `[width - 1, height - 1]` is the top-right. If a grid is toroidal, the top and bottom, and left and right, edges wrap to each other.

In the loop to create agents, the agents are added to a random x- and y- cell of the grid.

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        # self.schedule = mesa.time.RandomActivation(self)

        # Create agents
```

```
        for i in range(self.num_agents):
            a = MoneyAgent(self)
            # self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        # Data collectors for the model
        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini},
            agent_reporters={"Wealth": "wealth"}
        )

    def step(self):
        self.datacollector.collect(self)
        self.agents.shuffle_do("step")
        # self.schedule.step()
```

Now we turn to the agent class. The new version defines methods to `move()` and to `give_money()`. Within the `move` method, we define that the agent could move to a neighbouring cell. With mesa, it is possible to define a couple of types of neighbourhoods: Moore and Von Neumann. In this case, we use the Moore neighbourghood.



In the `give_money` method, the logic proceeds as follows:

1. Retrieve the agents in neighbouring cells.
2. Remove the agent itself from the list of neighbours, so that an agent will not give money to itself.
3. Choose one of the neighbouring agents at random and give money to it.

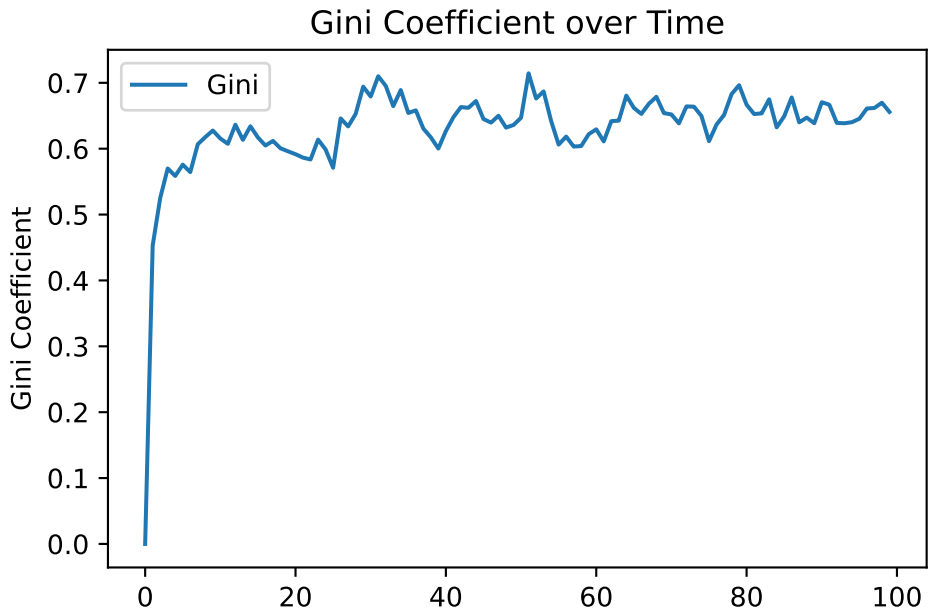
```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        super().__init__(model)
        self.wealth = 1
```

```
    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        cellmates.pop(
            cellmates.index(self)
        )  # Ensure agent is not giving money to itself
        if len(cellmates) >= 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    # First, the agent moves. Then, it might give money away to another agent in the neighbour
    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
```

7.9.2 Data Collection

The next code cell runs a single iteration with 100 agents in a 10x10 grid.

```
model = MoneyModel(100, 10, 10)
for i in range(100):
    model.step()

gini = model.datacollector.get_model_vars_dataframe()
# Plot the Gini coefficient over time
g = sns.lineplot(data=gini)
g.set(title="Gini Coefficient over Time", ylabel="Gini Coefficient");
```




```
params = {"width": 10, "height": 10, "N": range(5, 20, 5)}

results = mesa.batch_run(
    MoneyModel,
    parameters=params,
    iterations=50,
    max_steps=100,
    number_processes=1,
    data_collection_period=1,
    display_progress=True,
)
```

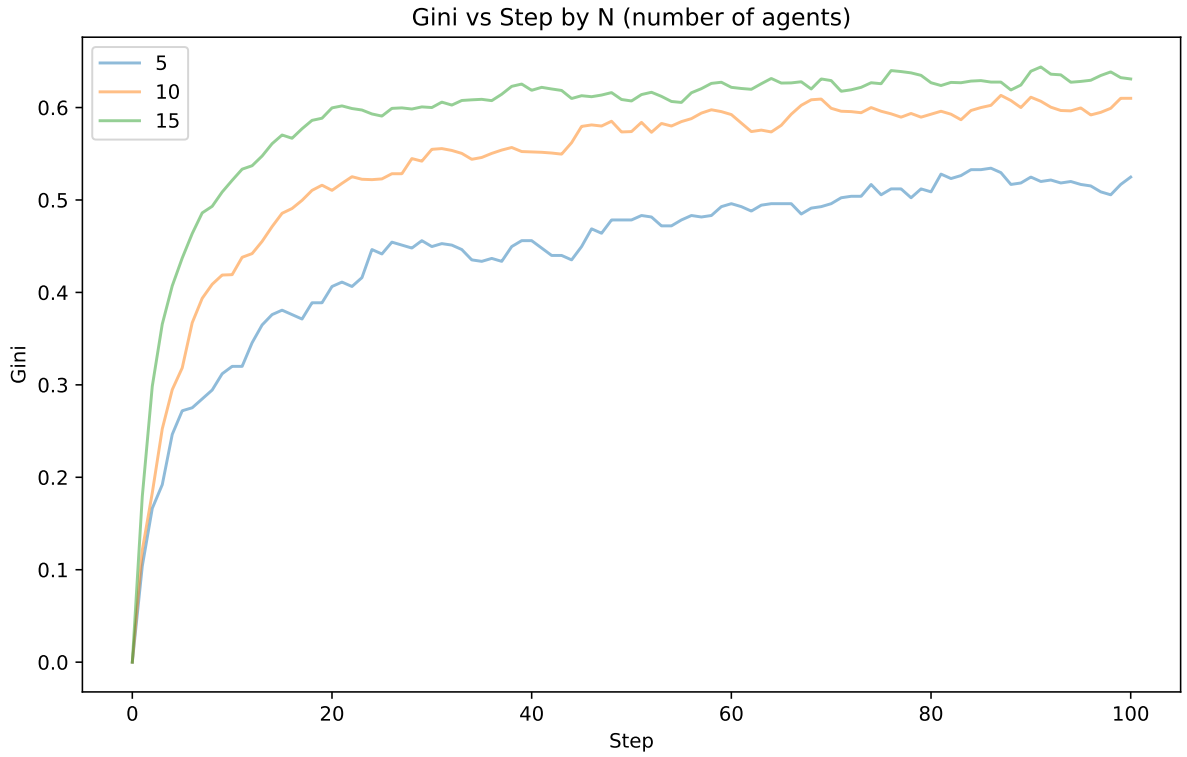
```
  0%|          | 0/150 [00:00<?, ?it/s]
```

For a visualisation of the results, we shall compute the mean Gini coefficient at each time step, grouped by `N`.

```
results_df = pd.DataFrame(results)
grp_by_step = results_df.Gini.groupby([results_df.Step, results_df.N]).describe()
grp_by_step.reset_index(inplace=True)

#grp_by_step.head()
```

```
grouped = grp_by_step.groupby('N')
plt.figure(figsize=(10, 6))

for group in grouped:
    plt.plot(group[1]['Step'], group[1]['mean'], label=group[0], alpha=0.5)

plt.xlabel('Step')
```

```
plt.ylabel('Gini')
plt.title('Gini vs Step by N (number of agents)');
plt.legend();
```



As we can see, the value at which the Gini coefficient converges to differs for varying `N`. For smaller `N`, the final Gini value seems to be lower. For all the settings, the convergence does appear to happen at approximately the same time step.

7.10 Visualisations

As with most ABM software, it is possible to visualise the steps. This is not to be used for conveying results to management; it is more for debugging, and for study for the purpose of understanding the behaviour and for generating hypothesis.

```
def agent_portrayal(agent):
    #return {
    #    "color": "tab:blue",
    #    "size": 50,
    #}
    size = 10
    color = "tab:red"
    if agent.wealth > 0:
        size = 50
        color = "tab:blue"
    return {"size": size, "color": color}
```

```
model_params = {
    "N": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}
```

```
#from mesa.experimental import JupyterViz

#page = JupyterViz(
#    MoneyModel,
#    model_params,
#    measures=["Gini"],
#    name="Money Model",
#    agent_portrayal=agent_portrayal,
#)
# This is required to render the visualization in the Jupyter notebook
#page
```

Remember that simulations are a risk-free approach to trying out different scenarios. They are especially useful when it is difficult to guage what interactions the output depends on. In this case, here are some modifications you can try to understand the Boltzmann Wealth Distribution model better.

- Vary the number of agents, to understand the Gini coefficient after 100 steps.
- When there are N agents, what is the proportion of agents that own 80% of the total wealth after a certain number of steps?
- How can we measure social mobility in this population?
- What if we change the starting income distribution?
- What if we want to know when Gini coefficient first crosses a threshold?
- What if we want to use Palma Ratio instead?

7.11 Simpler Simulation Models

In this section, we provide two examples of "simpler" simulation scenarios. These are not agent-based models, but they use probability distributions to model language, and then to represent the scenario where we intend to perform a 2-sample t-test.

7.11.1 N-gram models

Here is an example of how we can simulate from distributions to generate random (nonsensical) sentences that are "similar" to actual sentences from a text.

```
#nltk.download('genesis')

kjv = nltk.corpus.genesis.words('english-kjv.txt')
bg = nltk.bigrams(kjv)
cfd = nltk.ConditionalFreqDist(bg)
```

The object `cfd` contains information on the distribution of words that come after. For instance, these are the words that come after "she":

```
cfd['she']
```

```
FreqDist({'said': 22, 'was': 14, 'bare': 14, 'called': 13, 'conceived': 11, 'had': 11, 'saw':
```

The object `kjv` contains all the words and punctuations from the King James Bible. Here are the first 20 words:

```
' '.join(kjv[:20])
```

```
'In the beginning God created the heaven and the earth . And the earth was without form , and
```

Next, we extract all bigrams and create a dictionary of frequency distributions. Each frequency distribution tabulates the occurences of the *next* word.

Suppose we are interested in what words/tokens come after the word 'And' in the bible:

```
cfd['And'].keys()
```

```
dict_keys(['the', 'God', 'let', 'to', 'on', 'every', 'out', 'a', 'Adam', 'they', 'he', 'when',
```

Thus 'the' is the most likely word after 'And'. The second most likely pairing would be 'And the', and so on. We can convert this into a pmf, just by normalising to ensure that it sums to 1. Then we are ready to generate a pairing once we know that the first word is 'And'. With the second word, we turn to its frequency distribution and simulate the third word, and so on.

```
rng = default_rng(1361)

def generate_model(cfdist, word, num=15):
    for i in range(num):
        print(word, end=' ')

        choices = np.array(list(cfdist[word].keys()))
        pp = np.array(list(cfdist[word].values()))
        pp = pp / np.sum(pp)
        word = rng.choice(choices, size=1, p = pp)[0]
```

```
generate_model(cfd, 'God', 10)
```

```
God amongst the daughter of his hands , to Jacob
```

7.11.2 Power Analysis

Recall that this is the general form of a 2-sample t-test:

$$H_0: \quad \mu = \mu_0 \tag{7.1}$$
$$H_1: \quad \mu \neq \mu_0 \tag{7.2}$$

Instead of using the p-value alone to assess the strength of evidence against the null hypothesis, an alternative method is to specify a significance level and then compare the p-value to it.

This returns the possible four outcomes of a test: * H_0 was true, but we rejected it (Type I error). * H_0 was true, and we did not reject it. * H_0 was false, and we rejected it. * H_0 was false, but we did not reject it (Type II error).

We want the Type I and II errors to be small. 1 minus the Type II error is known as the power of a test. Typically, we have to specify a particular value of the parameter in order to compute the power of a test. We cannot simply leave it as $\mu \neq \mu_0$. These ideas can be used to compute a sample size for an experiment that we wish to conduct. Let's think about this for a minute.

Suppose we wish to perform some simple A/B testing: we have two groups, and we wish to be able to detect a difference. What sample size do I need?

The question is actually a little more complex than that. Let's think about what we need to consider: * How does the true difference affect my sample size? * How does the variability within each group affect my sample size? * How does the significance level affect the sample size I need? Recall that the significance level determines my Type I error.

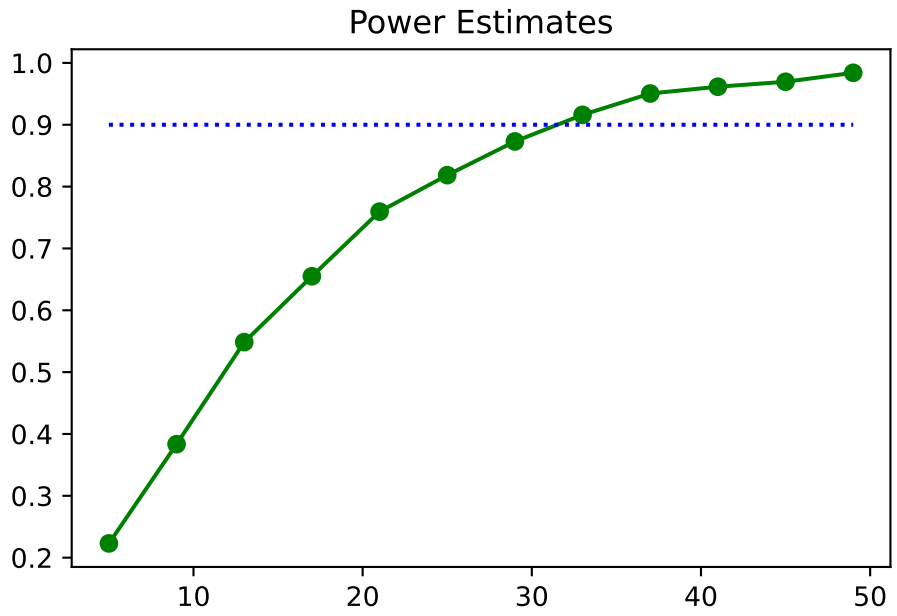
Suppose now, that we fix the significance level to be 5%, with power of at least 0.9 and we are interested in detecting a difference between the means of the value of 1, when the standard deviation of observed values is 1.2. What sample size do I need?

```
def generate_one_sample(alpha, delta_m, sd1, n):
    X = np.random.randn(n)*sd1
    Y = np.random.randn(n)*sd1 + delta_m

    ts2 = stats.ttest_ind(X, Y)
    if ts2.pvalue < alpha:
        return 1 # 1 means reject H0
    else:
        return 0
```

```
n_vals = np.arange(5, 50, step=4)

power_est = []
for n_ in n_vals:
    x = [generate_one_sample(0.05, 1, 1.2, n_) for ii in np.arange(0, 2000)]
    power_est.append(np.mean(x))
    print("Done with sample size " + str(n_))
```

```
Done with sample size 5
Done with sample size 9
Done with sample size 13
Done with sample size 17
Done with sample size 21
Done with sample size 25
Done with sample size 29
Done with sample size 33
Done with sample size 37
Done with sample size 41
Done with sample size 45
Done with sample size 49
```

```
ax = plt.plot(n_vals, power_est, 'go-')
plt.hlines(0.9, n_vals[0], n_vals[-1], colors='b', linestyles='dotted')
plt.title('Power Estimates');
```



7.12 Summary: Simulation-Based Modeling

In simulation-based modeling, we have several things to consider.

1. What input distributions should I use?
2. How complicated does my simulation need to be?
3. How do I know it is giving the right results?
4. How many simulations should I run?

Here are some general guidelines to begin with your simulation model:

1. Start simple.
2. Use your own data to decide what distributions should be used.

3. Try with different distributions to see how sensitive your results are to those choices.
4. Add more and more realistic layers to our simulation as you proceed.
5. Remember that simulation is used in estimating mean values. You will be able to estimate standard deviations around your estimate too.

7.13 References

7.13.1 Mesa Links

1. Mesa documentation: The Boltzmann tutorial in our lecture was modified from this page.
2. Mesa examples on github

7.13.2 Other ABM Software

1. Netlogo: Netlogo is an open-source software for ABM. There are a host of examples that you can take a look. However NetLogo uses it's own programming language.
    - https://ccl.northwestern.edu/netlogo/models/index.cgi : Several of the examples can be run directly on the web. The mesa examples repository has several mesa versions of these NetLogo models.
2. AnyLogic: AnyLogic is a commercial software from a Russian company that combines ABM with DES. It is very powerful and allows for the creation of impressive animations.

7.13.3 Reference Papers and Websites

1. Statistical Mechanics of Money, Income, and Wealth: A Short Survey: This paper contains more information about more complex versions of the Boltzmann model.
2. Explanation of Boids model
3. Hotelling's Law
4. Segregation in polygons

7.13.4 Other Simulation Software

1. Simpy documentation Simpy is a Python package for Discrete Event Simulations (DES). It is appropriate when there are arrivals to a set of queues.
2. Arena ARENA is a commercial DES software. There is a free license for academic use.

7.13.5 Other Links

1. Python documentation on classes

8 Supervised Learning

8.1 Introduction

In supervised learning, our goal is to develop a model that can predict a quantity of interest from a set of features. In this process,

- Algorithms learn from a training set of labelled examples.
- This training set is meant to be representative of the set of all possible inputs.
- Example algorithms include logistic regression, support vector machines, decision trees and random forests. Regression models can also be used for supervised learning.

Here are some examples:

1. We wish to predict if a student will graduate from university or not, based on his/her 'A' level results.
2. We wish to predict tomorrow's stock price based on today's price.

8.2 Classification versus Regression

If the answer to the question (supervised learning problem) we are facing is either YES or NO, then what we have is a **classification** problem. Here are some examples:

- Given the results of a clinical test, does this patient suffer from diabetes?
- Given an MRI, is there a tumor?

On the other hand, if we are trying to predict a real-valued quantity, then we are faced with a **regression** problem.

- Given the details about an apartment, what will the rental be?
- Given historical transactions of a customer, how much is he likely to spend on his next purchase?

8.3 Supervised Learning Workflow

The overall workflow in supervised learning is as follows:

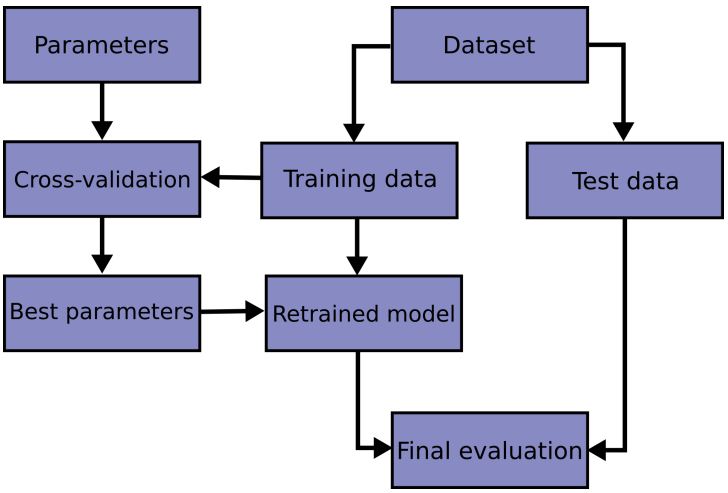


Figure 8.1: Figure from sklearn documentation

Here are the specific details:

1. Split up a dataset into training and test datasets, typically along a 80/20 or 75/25 split. **Do not touch the test data again until the end.**
2. Preprocess/clean the training data and store the parameters for later use on the test data.
   - Example preprocessings are scaling, one-hot encoding, PC decomposition, etc.
3. Decide on what models you wish to try. Each model has parameters to be fit (from the data), and **hyperparameters** to be chosen by you.
   - Example models are k-nearest neighbours (KNN) and random forests.
     - A hyperparameter for KNN is the number of neighbours to use.
     - A hyperparameter for random forests is the number of trees.
   - Hyperparameters usually control the **complexity** of a model. If a model is too complex, it will over-fit to the training data but fare poorly on the test data.
4. Use **cross-validation** or a set-aside validation set to decide on the hyperparameters for your chosen estimator. To fit the parameters for a particular hyperparameter configuration, we typically minimise a loss function or error metric.
5. Once you are satisfied with your choice(s) or model, evaluate the selected model on the test set to obtain an estimate of your generalisation error.

8.4 Scikit-learn

Scikit-learn is a library in Python which has several useful functions used in machine learning. The library has many algorithms for classification, regression, clustering and other machine learning methods. It uses other libraries like NumPy and matplotlib which are also used in this course. The website for scikit-learn is an excellent source of examples and tips on using the functions within this package (see the references).

All objects in scikit-learn have common access points. The three main interfaces are:

1. Estimator interface - `fit()` method.
   - This function allows us to build and fit models.

- Any object that can estimate some parameters based on a dataset is an *estimator*.
- Estimation is performed by the `fit()` method. This method takes in two datasets as arguments (the input data, and the corresponding output/labels).

2. Predictor interface - `predict()` method.

   - This function allows us to make predictions.
   - Estimators capable of making predictions when given a dataset are called *predictors*.
   - A predictor has a `predict()` method. It takes in a dataset of new instances and returns a dataset of corresponding predictions.

3. Transformer interface - `transform()` method.

   - This function is for converting data.
   - Estimators which can also transform a dataset are called *transformers*.
   - Transformations are carried out by the `transform()` method.
   - This method takes in the dataset to transform as a parameter and returns the transformed dataset.
   - We will not have too much time to spend on the transformer interface in this course.

8.4.1 Input Data Structure

For supervised learning problems in scikit-learn, the input data has to be structured in NumPy-like arrays.

The **feature matrix X**, of shape n by d contains features: * n rows: the number of samples * d columns: the number of features or distinct traits used to describe each item in a quantitative manner

Each row in the feature matrix is referred to as a sample, example or an instance.

$$\text{feature matrix:} \quad \mathbf{X}_{n\times d} = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,d} \\ \dots & \dots & \dots & \dots \\ x_{n,1} & x_{n,2} & \dots & x_{n,d} \end{bmatrix}$$

A **label vector y** stores the target values. This vector stores the true output value for each corresponding sample (row) in matrix X.

$$\text{label vector:} \quad \mathbf{y} = \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix}^T$$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from itables import show

from lime import lime_tabular

from sklearn import tree
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import cross_val_score, cross_validate, ShuffleSplit, learning_cu
```

```
from sklearn.inspection import permutation_importance
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.inspection import PartialDependenceDisplay, partial_dependence
```

8.5 Measures of Performance

8.5.1 For Classification

Before we head into creating classifiers which will help us predict heart failure, let's understand what determines the usefulness of a classifier. For now, we focus on the case where the outcome is binary (only two possible values for y_i).

Accuracy

A basic measure of performance would be the *accuracy* of predictions.

$$\text{accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

When more detailed analysis is needed, partial performance metrics can be presented in a *confusion matrix*. A confusion matrix is a contingency table that arises from cross-classification of predictions and the actual outcomes.

	Positive prediction	**Negative prediction**
Positive truth	TP	FN
Negative truth	FP	TN

In the confusion matrix, there are 4 possible cases: * True positives (TP) * Classifier predicts sample as positive and it really is so. * False positives (FP) * Classifier predicts sample as positive but in truth, it is negative. **Incorrect prediction** * True negatives (TN) * Classifier predicts sample as negative and it really is so. * False negatives (FN) * Classifier predicts sample as negative but in truth, it is positive. **Incorrect prediction**

Precision and Recall

With the confusion matrix, more performance metrics can be defined besides the accuracy of a classifier.

The **recall** of a classifier is the proportion of TP correctly identified:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The **precision** of a classifier is the proportion of predicted positives that are truly positive:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Depending on the context of the problem, it may be more important to have better recall than precision. In the above, we have defined recall and precision for the *positive* category outcome. There are analogous definitions for the *negative* outcome.

Note that recall is also sometimes referred to as the True Positive Rate (TPR), while (1 − precision) is also referred to as the False Positive Rate (FPR).

F1 score

The harmonic mean of two numbers x_1 and x_2 is

$$\left(\frac{1/x_1 + 1/x_2}{2}\right)^{-1}$$

We can combine precision and recall into one score using their harmonic mean:

$$F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{receall}}$$

Roughly, the F1 score is a summary of how good the classifier is in terms of both precision and recall. The F1 score is preferable to the simple arithmetic mean of precision and recall, because it ensures that both are high; the F1 score will be significantly lower than the mean if one of precision or recall is very low.

8.5.2 For Regression

Root Mean Squared Error

For regression problems, the typical measure of accuracy is RMSE. Let * y_i be the observed quantity (that we wish to predict), for $i = 1, \ldots, n$, and * $\hat{y}_i$ be the predicted quantity for observation i.

The RMSE is defined to be:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Mean Absolute Error

Using the RMSE amplifies the effect of outliers because of the square-term in the equation. Hence, in order to be resistant to outliers, one alternative is to the mean absolute error.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

8.6 Classification

In this section, we demonstrate how we can use `scikit-learn` to perform supervised learning on a classification problem.

Example 8.1 (Example: Heart Failure)**.** This dataset, from the UCI machine learning repository contains records on 299 patients who had heart failure. The data was collected during the follow-up period; each patient had 13 clinical features recorded. The primary variable of interest (y) was whether they died or not. Here are more details about each column:

- `age`: age of the patient (years)
- `anaemia`: decrease of red blood cells or hemoglobin (boolean)
- `creatinine phosphokinase` (CPK): level of the CPK enzyme in the blood (mcg/L)
- `diabetes`: if the patient has diabetes (boolean)
- `ejection_fraction`: percentage of blood leaving the heart at each contraction (percentage)
- `high_blood_pressure`: if the patient has hypertension (boolean)
- `platelets`: platelets in the blood (kiloplatelets/mL)
- `sex`: woman or man (binary)
- `serum_creatinine`: level of serum creatinine in the blood (mg/dL)
- `serum_sodium`: level of serum sodium in the blood (mEq/L)
- `smoking`: if the patient smokes or not (boolean)
- `time`: follow-up period (days)
- `DEATH_EVENT`: if the patient died during the follow-up period (boolean). This is the categorical outcome that we wish to predict.

```
hf = pd.read_csv("data/heart+failure+clinical+records/"+
                 "heart_failure_clinical_records_dataset.csv")
print(hf.head())
```

```
    age  anaemia  creatinine_phosphokinase  diabetes  ejection_fraction  \
0  75.0        0                       582         0                 20
1  55.0        0                      7861         0                 38
2  65.0        0                       146         0                 20
3  50.0        1                       111         0                 20
4  65.0        1                       160         1                 20

   high_blood_pressure  platelets  serum_creatinine  serum_sodium  sex  \
0                    1  265000.00               1.9           130    1
1                    0  263358.03               1.1           136    1
2                    0  162000.00               1.3           129    1
3                    0  210000.00               1.9           137    1
4                    0  327000.00               2.7           116    0

   smoking  time  DEATH_EVENT
0        0     4            1
1        0     6            1
2        1     7            1
3        0     7            1
4        0     8            1
```

8.6.1 Decision Tree

To begin our journey into supervised learning, we shall fit a decision tree to this dataset. A decision tree consists of a hierarchal set of rules, that when followed, will return a prediction for an individual observation. Each is a (typically binary) split of one of the features in the observation. One of the main advantages of decision tree classifiers is that they are easy to interpret. However, some disadvantages are that: They tend to overfit to a dataset, and they have high variability. This latter point means that a small change in the training data *could* lead to vastly different predictions. Although in this example we focus on classification, a decision tree can also be used for regression.

In this first example, we shall only split the data into a test and training set. We shall fit the tree using the training set, and then apply the model to the test set.

```
y = hf.DEATH_EVENT
X = hf.iloc[:, 0:12]

clf = tree.DecisionTreeClassifier(max_depth=4)
```

The `max_depth` of a decision is the maximum number of splits down each branch of a tree. If it is not specified, it is possible that the splits continue until the terminal nodes are homogeneous. This could result in overfitting of the tree to this particular dataset.

```
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.25,
                                                 random_state=41, stratify=y)
```

The `train_test_split` divides the data into a training and test set. When doing so, the function will attempt to ensure that the proportion of classes in both the training and test sets are roughly equal to the proportion in the overall dataset.

```
print(f"The proportion of 1's in the overall data is {y.mean():.3f}.")
print(f"The proportion of 1's in the training data is {y_train.mean():.3f}.")
print(f"The proportion of 1's in the test data is {y_test.mean():.3f}.")
```

```
The proportion of 1's in the overall data is 0.321.
The proportion of 1's in the training data is 0.321.
The proportion of 1's in the test data is 0.320.
```

```
clf.fit(X_train, y_train,)
```

criterion	'gini'
splitter	'best'
max_depth	4
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	None
random_state	None

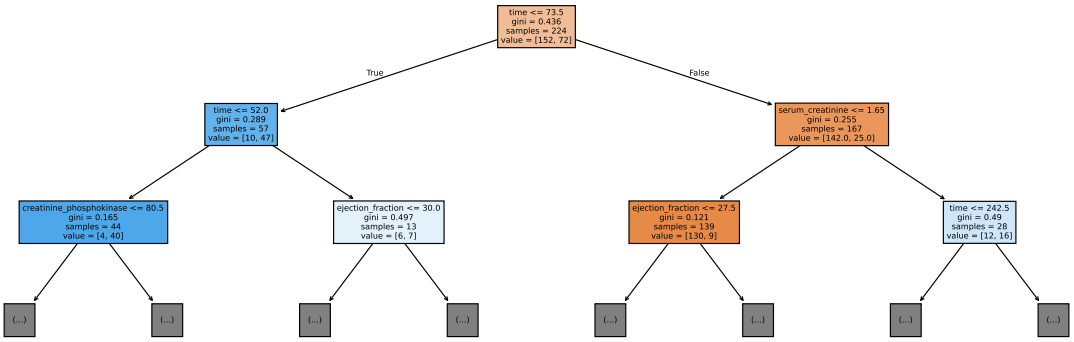
max_leaf_nodes	None
min_impurity_decrease	0.0
class_weight	None
ccp_alpha	0.0
monotonic_cst	None

Notice that when we call the `.fit()` method, there is no output object returned; it is just that the parameters in the `clf` object are updated.

```
clf.predict_proba(X_train.sample(random_state=3005))
```

```
array([[0.66666667, 0.33333333]])
```

```
plt.figure(figsize =(18, 6))
tree.plot_tree(clf,feature_names=X.columns, filled=True, max_depth=2);
```



The figure above visualises the rules in the first two layers (depth 2) of the tree. First off, notice that the tree is upside down, with the root on top; the terminal nodes (or boxes) at the bottom are referred to as the leaves. To understand the tree, consider using it to obtain a new prediction, with the following two features:

- `time`=70, `ejection_fraction` = 40.5

Since the value for the `time` variable is less than or equal to 73.5, we go down the left branch. Next, since `time` is more than 52.0, we go down the right branch. Finally, since `ejection_fraction` is more than 30, we would go down the right branch, and so on.

The information in the node summarises the splitting rule at that node. `samples` refers to the number of observations from the training set that would reach that node. The `value` vector indicates the count of each output class that has reached that node. For instance, for the blue node at the second level on the extreme left, out of the 44 observations that reached it, 4 were class 0 and 40 were class 1. The Gini impurity index is what is used to decide the split (not the same Gini as the Gini coefficient for income inequality!). For our binary classification problem, the formula for Gini impurity at each node reduces to:

$$p_0(1-p_0)$$

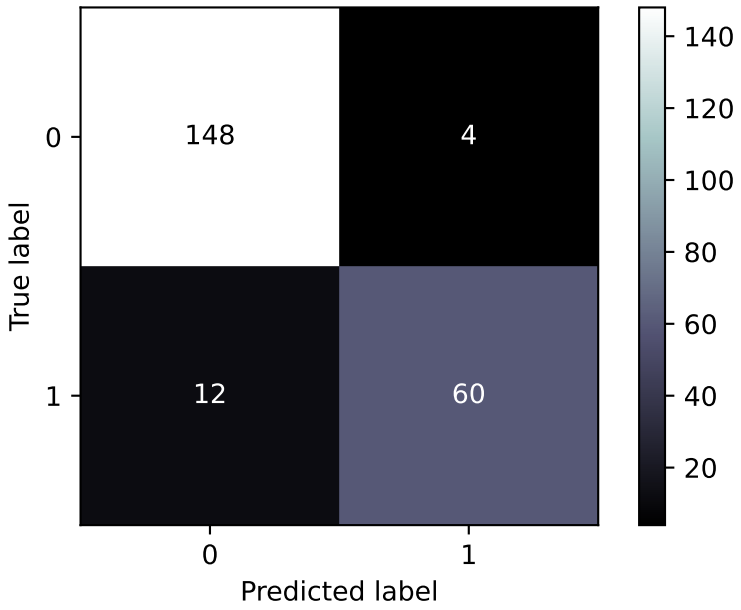
where p_0 is the proportion of class 0 at that node. If a node consists all 0's or all 1's, then the impurity index would be 0. The fitting algorithm tries all features, and all split points for each feature, to obtain the feature and split point that yields the largest drop in impurity at that node.

Before proceeding with the calculation of error metrics, notice the non-linearity of the splits. Both the initial split and the subsequent split on the left are on `time`. This indicates a non-linear relationship with time, and is not a property of classical models such as out-of-the-box linear regression.

Example 8.2 (Example: Heart Failure Classification Scores)**.** Performance of classification algorithms are usually presented in the form of a *confusion matrix*. First we display the results for the training set.

```
y_pred_train = clf.predict(X_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_pred_train,
                                        labels=clf.classes_, cmap='bone');
print(f"""
##: For training set:
----
The precision (for cat. 1) is {precision_score(y_train, y_pred_train):.3f}
The recall (for cat. 1) is {recall_score(y_train, y_pred_train):.3f}
The accuracy (for cat. 1) is {accuracy_score(y_train, y_pred_train):.3f}
The f1-score (for cat. 1) is {f1_score(y_train, y_pred_train):.3f}
""")
```

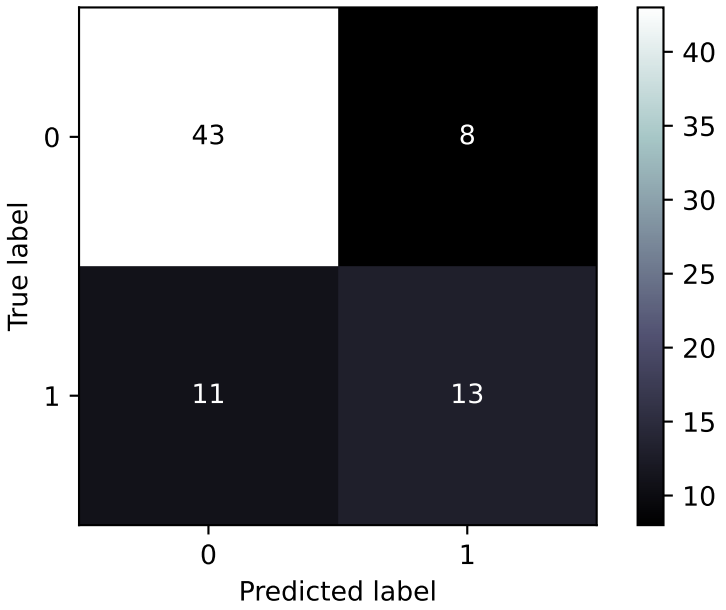
```
##: For training set:
----
The precision (for cat. 1) is 0.938
The recall (for cat. 1) is 0.833
The accuracy (for cat. 1) is 0.929
The f1-score (for cat. 1) is 0.882
```



Next, we display the results for the test set.

```
y_pred_test = clf.predict(X_test)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_test,
                                        labels=clf.classes_, cmap='bone');
print(f"""
##: For test set:
----
The accuracy (for cat. 1) is {accuracy_score(y_test, y_pred_test):.3f}
The precision (for cat. 1) is {precision_score(y_test, y_pred_test):.3f}
The recall (for cat. 1) is {recall_score(y_test, y_pred_test):.3f}
The f1-score (for cat. 1) is {f1_score(y_test, y_pred_test):.3f}
""")
```

```
##: For test set:
----
The accuracy (for cat. 1) is 0.747
The precision (for cat. 1) is 0.619
The recall (for cat. 1) is 0.542
The f1-score (for cat. 1) is 0.578
```



It is normal to observe poorer performance on the test set than on the training set. However, such a large difference in the scores usually indicates that there has been overfitting by the model; it is too tuned to the data in the training set. In supervised learning, we aim for a classifier that performs almost as well on the test data as it does on the training data. That would indicate that the model, with this particular hyperparameter configuration, will perform well on new data that has yet to be seen.

8.6.2 Variable importance

As we shall discuss in a later section of this topic, it is important to be able to identify which features are important to the model. This allows non-technical folk in your team (who could be end-user domain experts, or upper management) to have faith in your models. If the features that are identified to be important align with what the domain experts know or intuit, that provides more buy-in for your analyses and recommendations.

We shall demonstrate two types of feature importance:

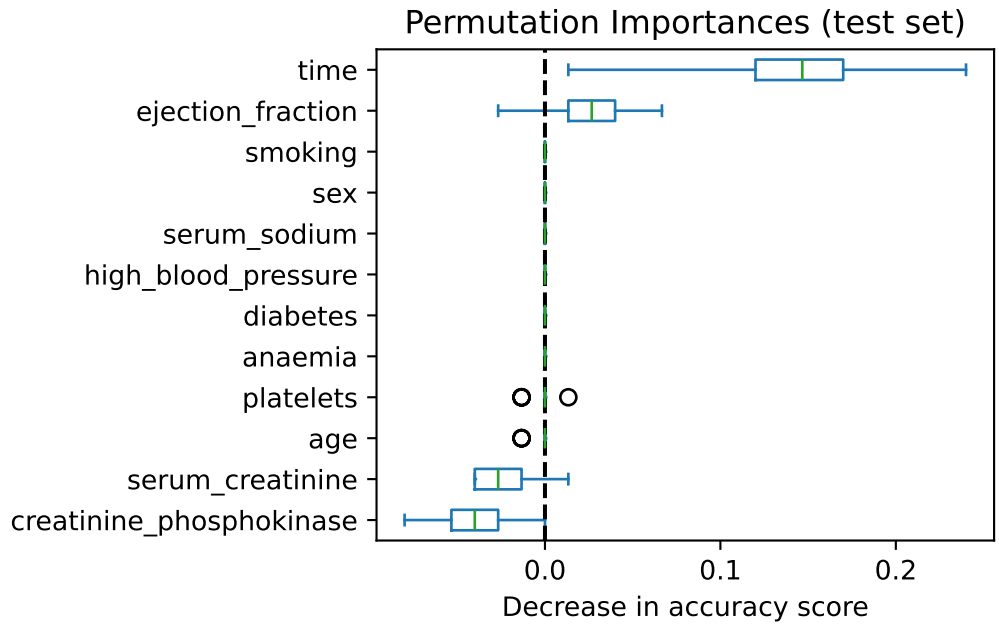
1. *Permutation based feature importance.* It works by permuting one of the features at a time, and measuring the drop in accuracy on the test set. The intuition is that, if a variable is highly important, randomising it will cause a sharp drop in predictive value.
2. *Partial dependence plots (PDP).* A PDP can be generated for each feature in the model, or a pair of features. Each PDP is generated by:

    1. Vary the feature over a range of values
    2. For each feature value in that range, average the predictions over all values of *other* variables occuring in the training data.

There are certain caveats with using the above measures. One is that these measures are indicative of how important a feature is to a particular model, not to predictive value. Hence an unimportant feature for a poor model (in terms of accuracy) could in fact be an important feature for a good model! Second, in the permutation approach, the importance of a variable may not show up if certain features are highly correlated (dropping one of them may have no effect because the other feature is still present in the model).

Permutation Based

```
result = permutation_importance(
    clf, X_test, y_test, n_repeats=30, random_state=42,
    # clf, X_test, y_test, n_repeats=10, random_state=42, n_jobs=2

)

sorted_importances_idx = result.importances_mean.argsort()
importances = pd.DataFrame(
    result.importances[sorted_importances_idx].T,
    columns=X.columns[sorted_importances_idx],
)
ax = importances.plot.box(vert=False, whis=10)
ax.set_title("Permutation Importances (test set)")
ax.axvline(x=0, color="k", linestyle="--")
ax.set_xlabel("Decrease in accuracy score")
ax.figure.tight_layout()
```



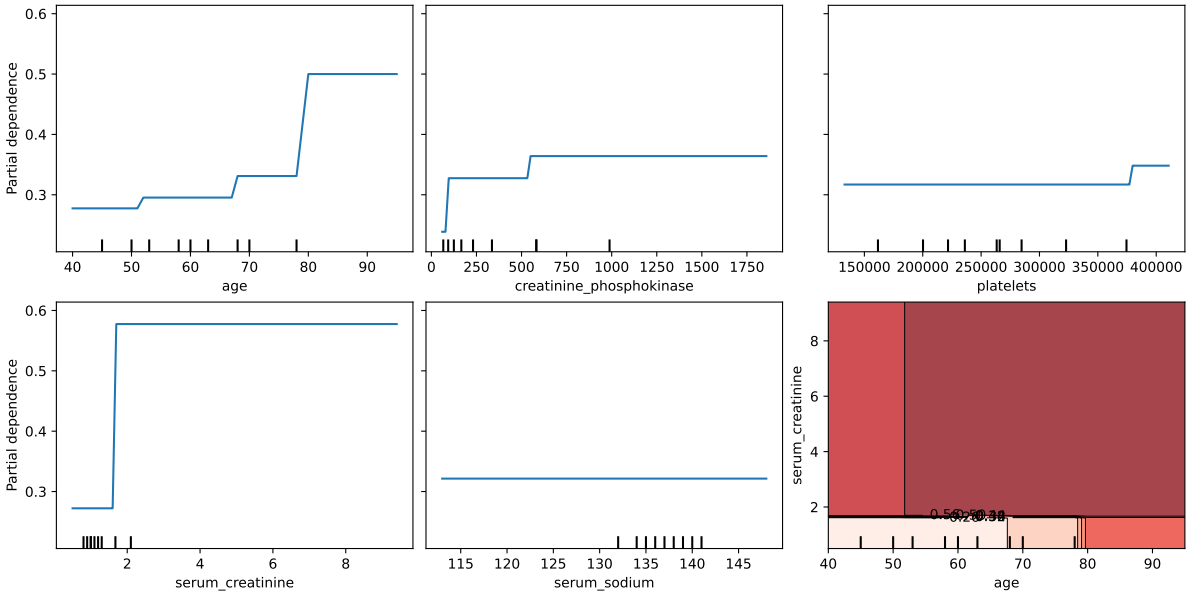
In the plot above, we can see that `time`, `ejection_fraction` and `serum_sodium` are the top 3 in terms of importance when assessing generalisability to test set.

Partial Dependence Plots

```
_, ax = plt.subplots(ncols=3, nrows=2, figsize=(12, 6), constrained_layout=True)
features_info = {
    "features": ["age", "creatinine_phosphokinase", "platelets",
                 "serum_creatinine", "serum_sodium", ("age", "serum_creatinine"),
                ],
```

```
    "kind": "average",
}
display = PartialDependenceDisplay.from_estimator(
    clf,
    X_train,
    **features_info,
    ax=ax,
    contour_kw = {'cmap': "Reds"}
)
```

```
/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/inspecti

The column 2 contains integer data. Partial dependence plots are not supported for integer dat

/home/viknesh/NUS/coursesTaught/ind5003-book/env/lib/python3.10/site-packages/sklearn/inspecti

The column 8 contains integer data. Partial dependence plots are not supported for integer dat
```



PDP are meant to visualise the relationship that has been learned in the training set; hence it is not usually created with the test set. From the plots, we can interpret that when other variables are already in the model, age returns a visible bump in the probability of death. `serum_creatinine` is associated with a similar increase, but at low levels. The final plot, at the lower right, darker reds indicate higher probability of death.

8.6.3 Random Forest

A random forest model is an example of an ensemble model. It aims to fix the weakness of decision trees by introducing a little noise in the fitting process. A random forest is in fact a collection of decision trees, with some added modifications. First of all, each tree in a random forest is fit using a bootstrapped version of the training data. Second, at each split, not all features are considered - only a sample of all available features is considered. When

a classification prediction is eventually made, it is made by averaging the predictions from all the individual trees. Through the introduction of these perturbations, the eventual model has lower variance (less susceptible to changes in the data), thus generalising to new data better.

Clearly an important property of the random forest is the number of trees to be fitted. This is known as a hyperparameter. More trees leads to a more complex model, which could overfit to the data. scikit-learn provides convenient grid search utilities for identifying the optimal number of trees through. This process, known as hyperparameter tuning, uses cross-validation on the training dataset.

Cross-validation begins with a split of the training data into k blocks (typically $k = 5$). Then, for each block i,

1. Set aside block i, and use the *remaining* data to fit the model.
2. Use the fitted model to predict on block i, thus obtaining one estimate of generalised error.

The process yields k estimates of error - one for each block that was set aside. This can be used to compare between hyperparamter settings. In this case below, we try to identify the optimal `max_depth` parameter for the trees. Too large a depth would overfit to the data, and we wish to avoid that.

```
p_range = range(1, 11, 1)
#list(p_range)
cv_search = GridSearchCV(RandomForestClassifier(n_estimators=20, random_state=21),
                         return_train_score=True,
                         param_grid ={'max_depth': p_range},
                         scoring = 'accuracy', cv= 5, verbose=1)
cv_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

estimator	RandomForestC...ndom_state=21)
param_grid	{'max_depth': range(1, 11)}
scoring	'accuracy'
n_jobs	None
refit	True
cv	5
verbose	1
pre_dispatch	'2*n_jobs'
error_score	nan
return_train_score	True

n_estimators	20
criterion	'gini'
max_depth	4
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0

max_features	'sqrt'
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False
n_jobs	None
random_state	21
verbose	0
warm_start	False
class_weight	None
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

After calling the `.fit()` method, the optimal parameters are available for persual:

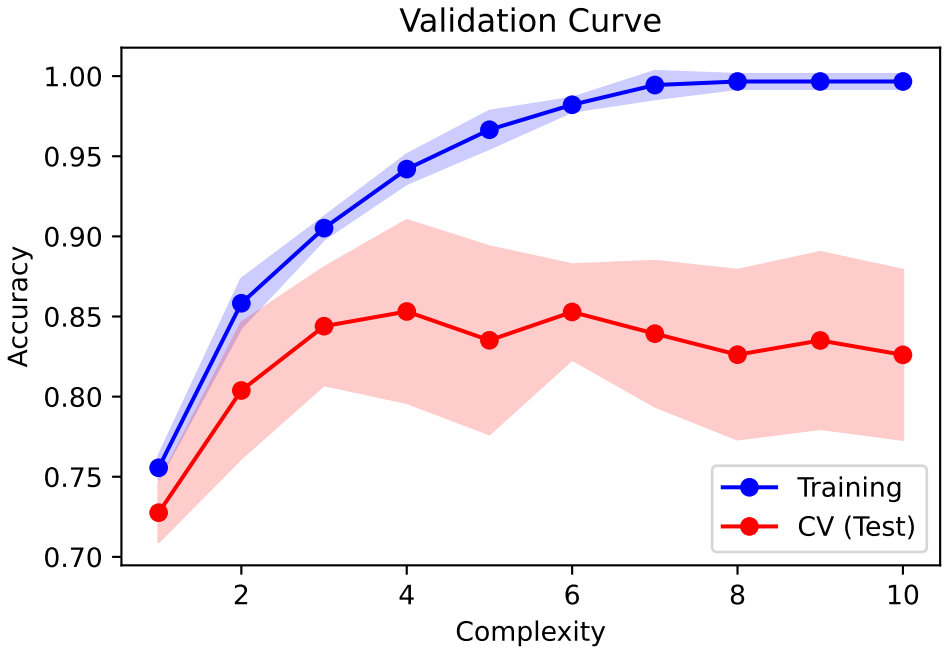
```
cv_search.best_estimator_
```

n_estimators	20
criterion	'gini'
max_depth	4
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	'sqrt'
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False
n_jobs	None
random_state	21
verbose	0
warm_start	False
class_weight	None
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

It is also possible, at this point, to extract the training and test scores for each cross-validation set, and to then plot the validation curve.

```
train_means = cv_search.cv_results_['mean_train_score']
train_sd = cv_search.cv_results_['std_train_score']

test_means = cv_search.cv_results_['mean_test_score']
test_sd = cv_search.cv_results_['std_test_score']
plt.plot(p_range, train_means, 'o-', label='Training', color='blue')
```

```
plt.fill_between(p_range, train_means-train_sd, train_means+train_sd,
                 color='blue', alpha=0.2)

plt.plot(p_range, test_means, 'o-', label='CV (Test)', color='red')
plt.fill_between(p_range, test_means-test_sd, test_means+test_sd,
                 color='red', alpha=0.2)

plt.legend(loc='lower right');plt.ylabel('Accuracy');
plt.xlabel('Complexity');plt.title('Validation Curve');
```



From the above chart, we can see that as the max. depth increases, both the training and test errors increase sharply, and then plateau. It even appears that the test error appears to come down. As we discussed, it is normal for the test error to be lower than the training set. Observe that with too high a complexity, accuracy in the training set approaches 1. From the curve, a good choice for the `max_depth` parameter is 3:

- The test error for this value is close to the training error.
- Any `max_depth` larger than this would lead to overfitting (fitting the noise patterns in the data to the model).
- The convention is to take the smallest complexity that is not more than 1 standard error away from the best performing model.

Example 8.3 (Exercise: Heart Failure Classification Scores II)

```
rf = RandomForestClassifier(n_estimators=20, max_depth=3, random_state=40)
rf.fit(X_train, y_train)
```

n_estimators	20
criterion	'gini'

max_depth	3
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	'sqrt'
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False
n_jobs	None
random_state	40
verbose	0
warm_start	False
class_weight	None
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

```
y_pred_train = rf.predict(X_train)
print(f"""##: For training set:
----
The precision (for cat. 1) is {precision_score(y_train, y_pred_train):.3f}
The recall (for cat. 1) is {recall_score(y_train, y_pred_train):.3f}
The accuracy (for cat. 1) is {accuracy_score(y_train, y_pred_train):.3f}
The f1-score (for cat. 1) is {f1_score(y_train, y_pred_train):.3f}
""")
```

```
##: For training set:
----
The precision (for cat. 1) is 0.950
The recall (for cat. 1) is 0.792
The accuracy (for cat. 1) is 0.920
The f1-score (for cat. 1) is 0.864
```

```
y_pred_test = rf.predict(X_test)
print(f"""##: For test set:
----")
The precision (for cat. 1) is {precision_score(y_test, y_pred_test):.3f}
The recall (for cat. 1) is {recall_score(y_test, y_pred_test):.3f}
The accuracy (for cat. 1) is {accuracy_score(y_test, y_pred_test):.3f}
The f1-score (for cat. 1) is {f1_score(y_test, y_pred_test):.3f}
""")
```

```
##: For test set:
----")
The precision (for cat. 1) is 0.762
The recall (for cat. 1) is 0.667
The accuracy (for cat. 1) is 0.827
The f1-score (for cat. 1) is 0.711
```

Compared with the earlier single decision tree, we have obtained an improved test accuracy (from 0.760 to 0.827).

8.7 Regression

Just to demonstrate the use of random forests for regression, we return to the Taiwan Data. In our scoring, instead of using RMSE or MAE, we shall use the R^2.

First, we read the Taiwan data into Python and transform each column to have mean 0 and sd 1.

```
re2 = pd.read_csv("data/taiwan_dataset.csv")

X_re = re2.loc[:, ['trans_date', 'house_age', 'dist_MRT',
                   'num_stores', 'Xs', 'Ys']]
re_scaler = StandardScaler().fit(X_re)
X_re_scaled = re_scaler.transform(X_re)
y_re = re2.price

Xre_train,Xre_test, yre_train,yre_test = train_test_split(X_re_scaled,
                                                          y_re, test_size=0.2,
                                                          random_state=41)
```

8.7.1 Random Forest Regressor

Next, we set up a grid search to set up a random forest regressor.

```
p_range = range(1, 11, 1)
rf_search = GridSearchCV(RandomForestRegressor(n_estimators=10, random_state=43),
                         {'max_depth': p_range},
                         scoring='r2', cv=5, verbose=1, return_train_score=True)
rf_search.fit(Xre_train, yre_train,)
rf_search.best_estimator_
```

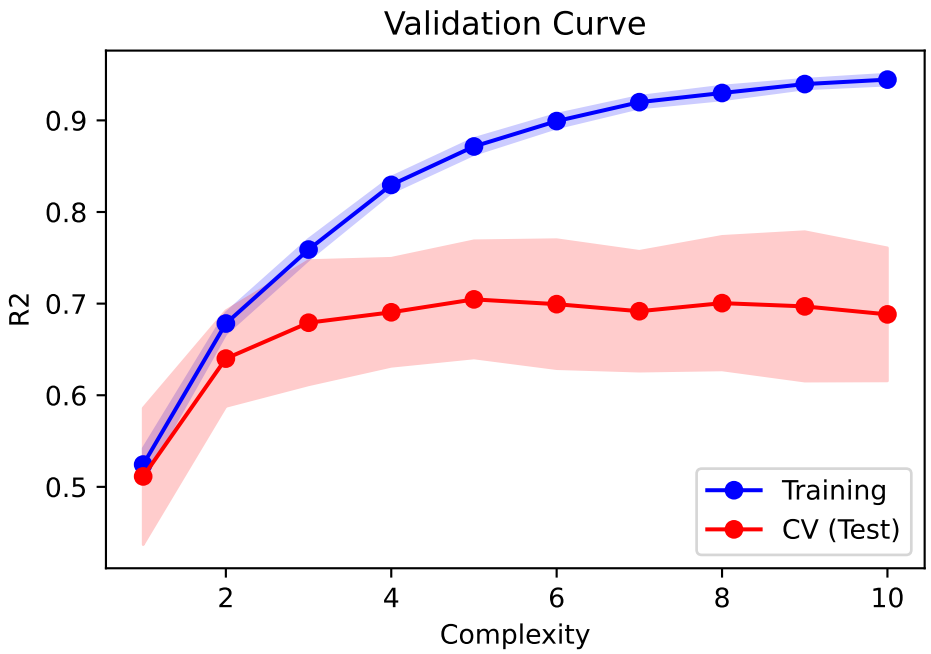
```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

n_estimators	10
criterion	'squared_error'
max_depth	5
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	1.0
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False

n_jobs	None
random_state	43
verbose	0
warm_start	False
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

Here is the validation curve, that will allow us to decide on the optimal number of trees.

```
train_means = rf_search.cv_results_['mean_train_score']
train_sd = rf_search.cv_results_['std_train_score']

test_means = rf_search.cv_results_['mean_test_score']
test_sd = rf_search.cv_results_['std_test_score']

plt.plot(p_range, train_means, 'o-', label='Training', color='blue')
plt.fill_between(p_range, train_means-train_sd, train_means+train_sd,
                 color='blue', alpha=0.2)

plt.plot(p_range, test_means, 'o-', label='CV (Test)', color='red')
plt.fill_between(p_range, test_means-test_sd, test_means+test_sd,
                 color='red', alpha=0.2)

plt.legend(loc='lower right');plt.ylabel('R2');
plt.xlabel('Complexity');plt.title('Validation Curve');
```



Example 8.4 (Example: Taiwan Data Regression)**.** We shall fit a model with 10 trees, and max_depth 2, and assess the test error.

```
rf1 = RandomForestRegressor(n_estimators=10, max_depth = 2, random_state=89)
rf1.fit(Xre_train, yre_train)
```

n_estimators	10
criterion	'squared_error'
max_depth	2
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	1.0
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False
n_jobs	None
random_state	89
verbose	0
warm_start	False
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

```
yrf_pred = rf1.predict(Xre_test)
```

```
r2_score(yre_test, yrf_pred)
```

```
0.5487486857968205
```

8.8 Interpretability of Models

8.8.1 LIME

While the earlier methods indicated how we can understand the importance of features to the overall model, we have better tools to understand how features play a role in particular predictions. One such approach is known as LIME (Local Interpretable Model-Agnostic Explanations). A local model is one that explains why the model is making a particular prediction for a particular combination of feature values.

It works by training a *new model* that is inherently explainable, like a decision tree or a linear regression model, using instances that are weighted according to their proximity to the instance of interest. The new model's prediction for this instance should be as close as possible to the prediction using the original model. To be precise, the steps are:

1. Select your instance of interest for which you want to have an explanation of its black box prediction.
2. Perturb your dataset and get the black box predictions for these new points.
3. Weight the new samples according to their proximity to the instance of interest.

4. Train a weighted, interpretable model on the dataset with the variations.

The following diagram visualises the process when making predictions (light-blue vs. gray) using two features (x_1 and x_2). The instance of interest is the yellow point.

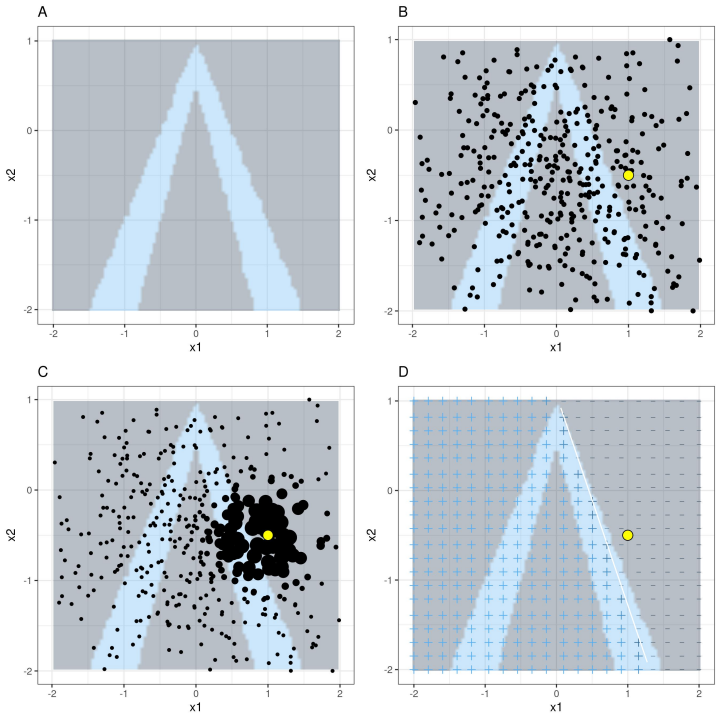


Figure 8.2: LIME

Example 8.5 (Example: Taiwan Data LIME)**.** Now we return to our dataset on real estate transations and attempt to understand the prediction made for a particular instance:

```
re_scaler.inverse_transform(Xre_test[4, :].reshape(1, -1)).round()
#X_re.mean(axis=0).round(3)
#Xre_test[4, :].round(4)
```

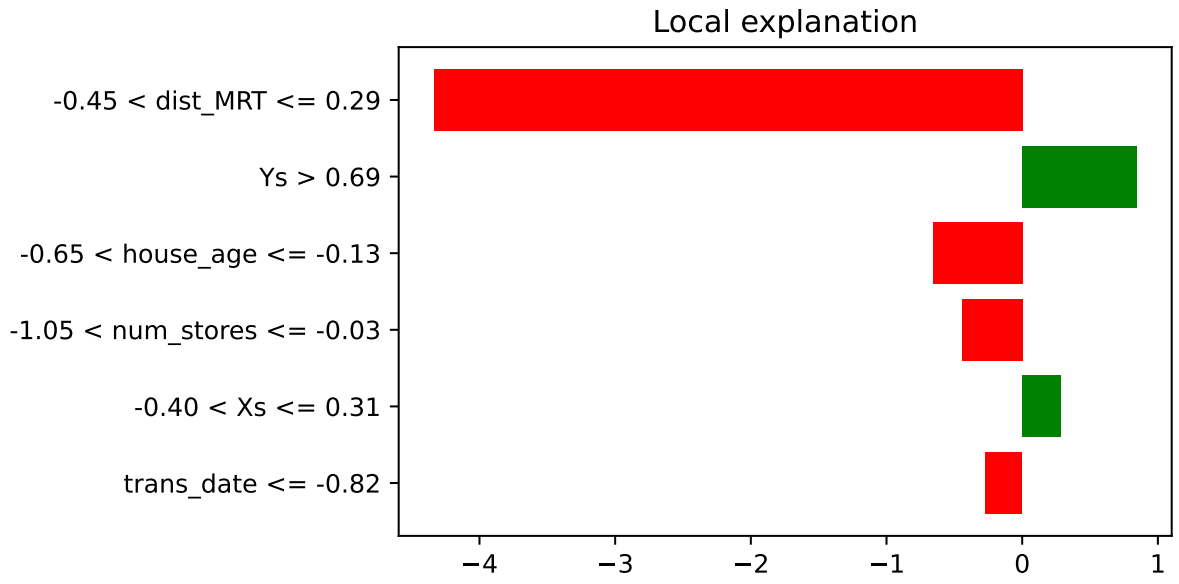
```
array([[2013.,   12., 1144.,    4.,    0.,    3.]])
```

First we instantiate an explainer object, and then provide it with the instance we wish to explain.

```
explainer = lime_tabular.LimeTabularExplainer(
    training_data=np.array(Xre_train),
    feature_names=X_re.columns,
    mode='regression',
)
```

```
exp = explainer.explain_instance(
    data_row=Xre_test[4, :],
    predict_fn=rf1.predict,
)
```

```
exp.as_pyplot_figure();
```




```
exp.show_in_notebook(show_table=True)
```

```
<IPython.core.display.HTML object>
```

Here is how we can interpret the plots above. In terms of importance to the prediction *of this particular instance*, `dist_MRT` is the highest. The local model that has been fit has been constrained to make as close a prediction as possible to the original model. However, it may fall short. In this case, the original predicted value was 29.58. However, the value predicted by the local approximation was:

```
print(f"The value predicted by the local approximation was: {exp.local_pred[0]:.3f}")
```

```
The value predicted by the local approximation was: 33.593
```

The numbers in the bar chart essentially decompose this prediction from the local model:

```
s = 0.0
for x,y in exp.local_exp[1]:
    s += y

s + exp.intercept[0]
```

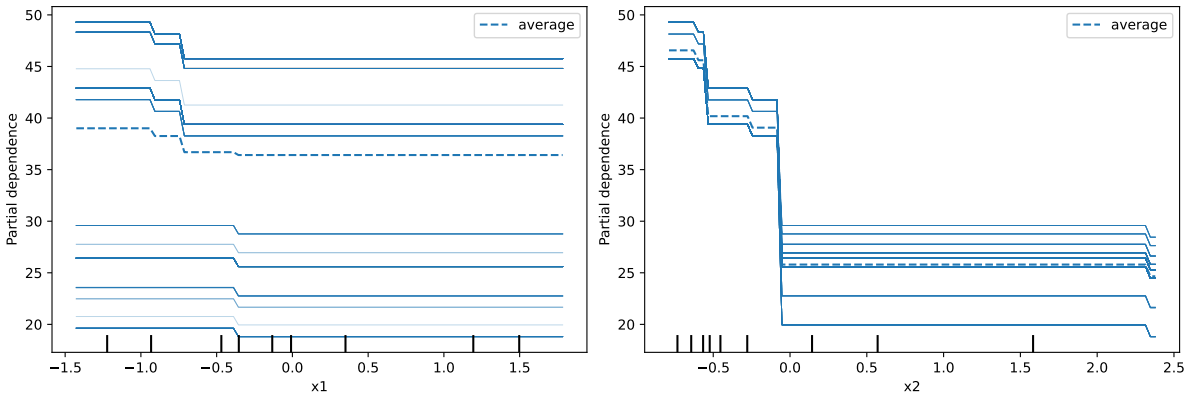
```
33.59339556024793
```

This means that, from the values, we can see that the `dist_MRT`, which was close to the average distance, had a negative impact on the price. The longitude had a positive impact on the price.

8.8.2 ICE plots

ICE plots are conditional versions of PDP. Instead of *averaging* over all instances in the training set, a single line is generated for each instance corresponding to features other than the one under study. This allows us to inspect if the relationship between the response and the feature is consistent across the data.

```
_, ax = plt.subplots(ncols=2, nrows=1, figsize=(12, 4), constrained_layout=True)
features_info = {
    "features": [1,2], # no names in the array; 1 and 2 correspond to house_age and dist_MRT
    "kind": "both",
}
display = PartialDependenceDisplay.from_estimator(
    rf1,
    Xre_train,
    **features_info,
    ax=ax
)
```



8.9 Summary

We have only touched on a random forests and decision trees. However, there are numerous other (non-deep learning) machine learning models. Examples are Support Vector Machines, Nearest Neighbours, and Linear Models. It would be good to skim through these models in the sklearn documentation to be aware of their existence. Instead of concentrating on learning a variety of shallow learning models, our topic has focused on the workflow when using these models, and how we should assess them. Nonetheless, it will be good if you can read up on your own on some of these models. The textbook ISL below is very good for learning about these models.

When working on your project, if you intend to perform supervised learning, please ensure that you also interpret the models, and do not leave them as a black box.

8.10 References

One of the best textbook references for this topic is Hastie, Tibshirani, and Friedman (2009). For more information on interpretable machine learning, refer to Molnar (2020). It is very comprehensive, with much more details on LIME and Shaply values.

8.10.1 Website and video references

1. Decision trees, clearly explained: This is from a popular YouTube channel that explains stats and data science concepts.
2. sklearn documentation: Documentation on available supervised learning models in sklearn.

8.10.2 Documentation references

1. Interpretable Machine Learning: A comprehensive textbook on interpreting machine learning models. See this book for more information about LIME and SHAPLY values.
2. LIME: Full documentation on LIME
    - Video introduction to LIME

9 Computer Vision

9.1 Introduction

Data science has a number of applications in computer vision. These have increased in number and accuray in the past 5 years or so due to the advancements in deep learning (both theory and computational feasibility). In this topic, we shall experiment with some of the applications, utilising existing deep learning models.

Here is a short list of applications of computer vision techniques:

1. Optical Character Recognition: Reading handwritten documents, car license plate numbers from images.

2. Surveillance and traffic monitoring: Monitoring cars on a highway, tracking humans in security cameras for suspicious activity.
3. Machine inspection: Automatically detecting damage on components manufactured, such as silicon wafers, etc.

Here is an old page with a more comprehensive list of applications.

In computer vision, the goal is to get a computer to perceive the world as *we* see it. This is not easy even for us to do - we can get fooled by optical illusions such as the ones below.

In general though, for us, it is possible to do things like pick out faces that we recognise from a photograph of a crowd. But how can we get a computer to do it?

```
import cv2
import sys
import numpy as np
import os

import matplotlib.pyplot as plt
from IPython.display import YouTubeVideo, display, HTML, Image, Video

opencv_path = "opencv/samples/data/"
opencvxtra_path = "opencv_extra/testdata/dnn/"
```

9.2 Image Processing

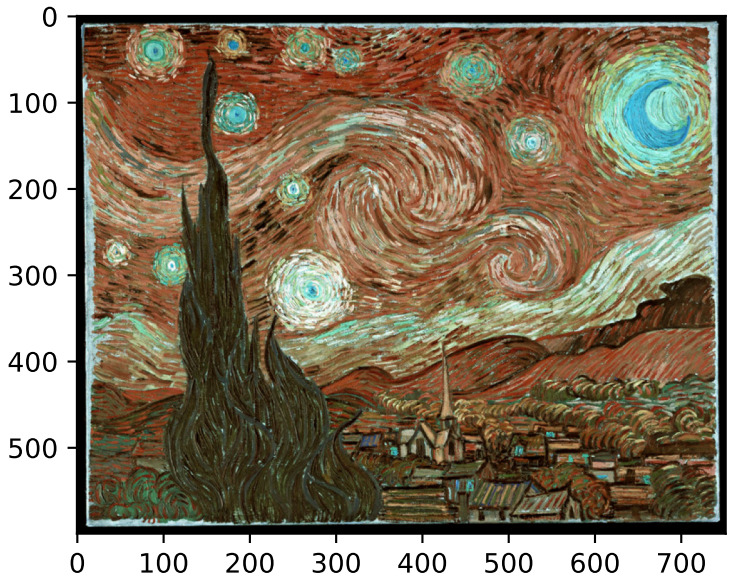
9.2.1 Reading Images

The `opencv` package contains routines for reading images into Python. Images are typically represented using the RGB colorspace. Each image is a (W x H x 3) numpy array. Each layer

corresponds to one of these channels. However, opencv uses the order BGR. When you use `plt.imshow` to plot an image that has been read in using opencv, take note of this, as it might not appear "correct".

```
Image("data/starry_night.jpg", width=240)

from matplotlib import colormaps
#list(colormaps)
```

```
starry_night = cv2.imread('data/starry_night.jpg')

# Reversed (not correct)
plt.imshow(starry_night);
#plt.imshow(starry_night[:, :, 0], cmap='Blues');
```



The following code should open up the starry night image in a new window on your computer.

```
# plotting with cv2.imshow returns the correct colours.
cv2.namedWindow('dst_rt', cv2.WINDOW_NORMAL)
cv2.resizeWindow('dst_rt', starry_night.shape[1], starry_night.shape[0])

# hit "q" to close window (do not hit the "X" button)
cv2.imshow('dst_rt', starry_night)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

9.3 Working with Masks

When working in a computer vision project, we often need to extract parts of the image to work with, or to modify. This is typically done using masks. Masks are black-and-white images that identify the foreground (white) and the background (black).

Let us work with the following image of Lionel Messi, and see how we can use masks and colour selection to outline the ball.

```
Image('data/messi5.jpg')
```

Our first task is to read the data into Python, and use a colour picker to identify the colour of the ball. The following website provides a useful tool for us to upload an image, and isolate the HSV (Hue-Saturation-Value) representaion of the colour we wish to pick out:

https://redketchup.io/color-picker

Using this website, we learn that the RGB representation we want is RGB=(244,251,95). We the convert this to HSV using a colour conversion function.

```
messi = cv2.imread('data/messi5.jpg')
hsv = cv2.cvtColor(messi, cv2.COLOR_BGR2HSV)
```

```
# 244, 251, 95
cv2.cvtColor(np.uint8([[[95, 251, 244 ]]]), cv2.COLOR_BGR2HSV)
```

```
array([[[ 31, 158, 251]]], dtype=uint8)
```

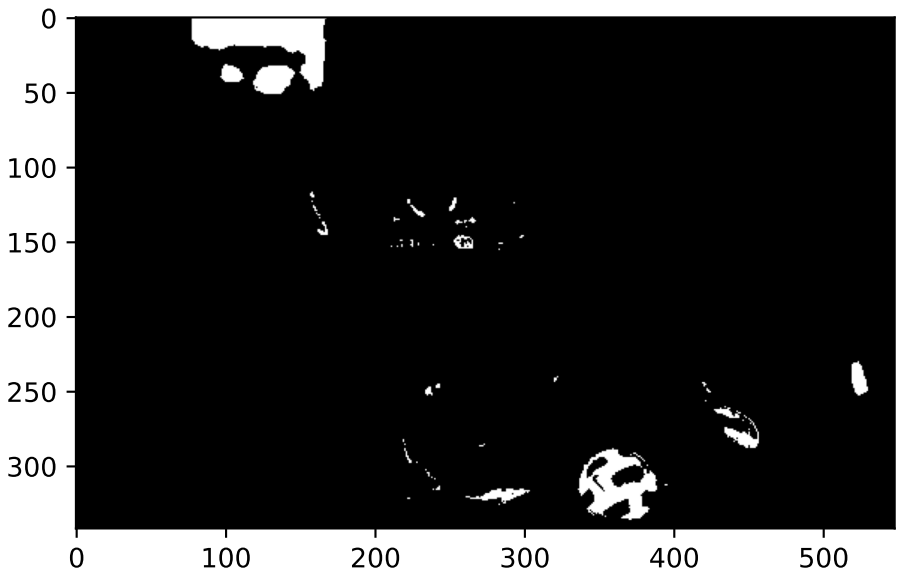
```
# define range of yellow color in HSV
lower_yellow = np.array([21,  100, 100])
upper_yellow = np.array([41, 255, 255])
```

```
# Threshold the HSV image to get only blue colors
mask1 = cv2.inRange(hsv, lower_yellow, upper_yellow)
```

Compare the black-and-white image below with the original one. The white regions below correspond to yellow coloured sections in the original image. The white pixels take the value 255, while the black pixels are 0. We are going to modify the numpy array corresponding to this mask to correspond to only the ball.

Take note of where the origin is: at the top-left corner. The x-values increase to the right from there; y-values increase downward from there.

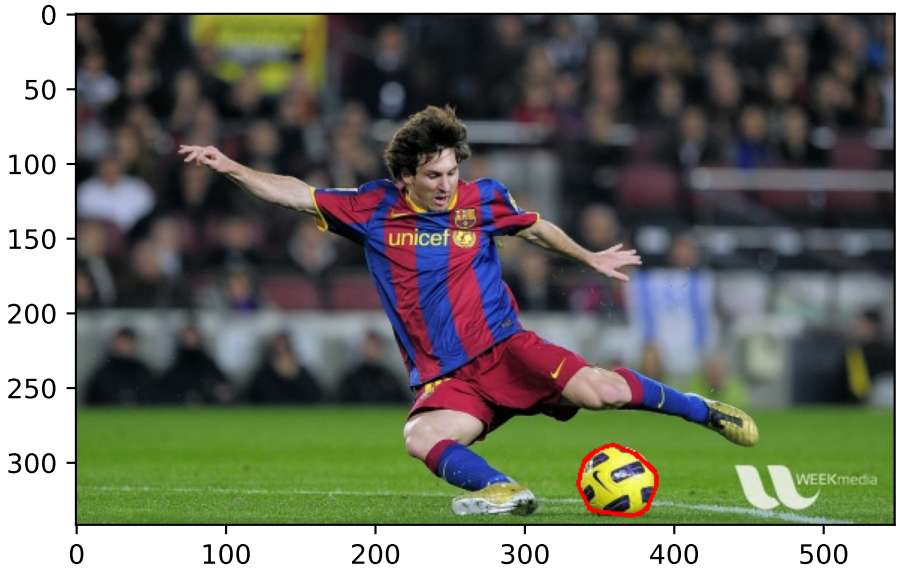
```
plt.imshow(mask1, cmap='gray');
```




```
mask1[:, :330] = 0
mask1[:, 405:] = 0
```

The next section of code identifies contours around the ball region, computes the convex hull around these points and then draws this on the original image.

```
#im2, contours= cv2.findContours(mask1, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

contours, _ = cv2.findContours(mask1, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

largest_contour = max(contours, key=cv2.contourArea)
hull = cv2.convexHull(largest_contour)
cv2.drawContours(messi, [hull], -1, (0, 0, 255), thickness=2)

plt.imshow(cv2.cvtColor(messi, cv2.COLOR_BGR2RGB));
```

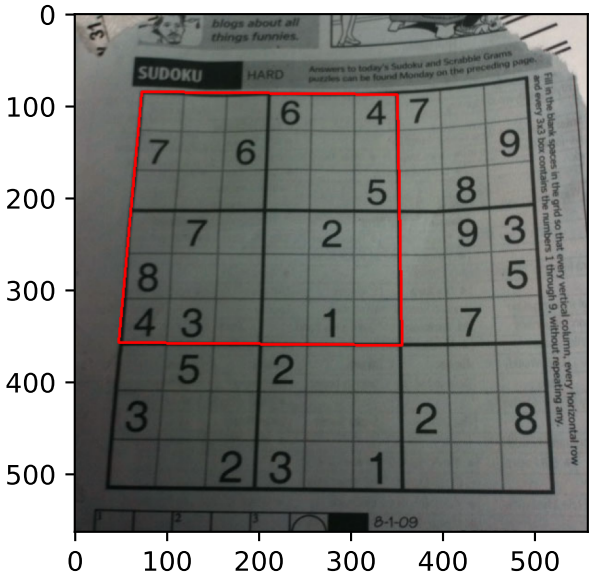


9.4 Modifying Perspective of Images

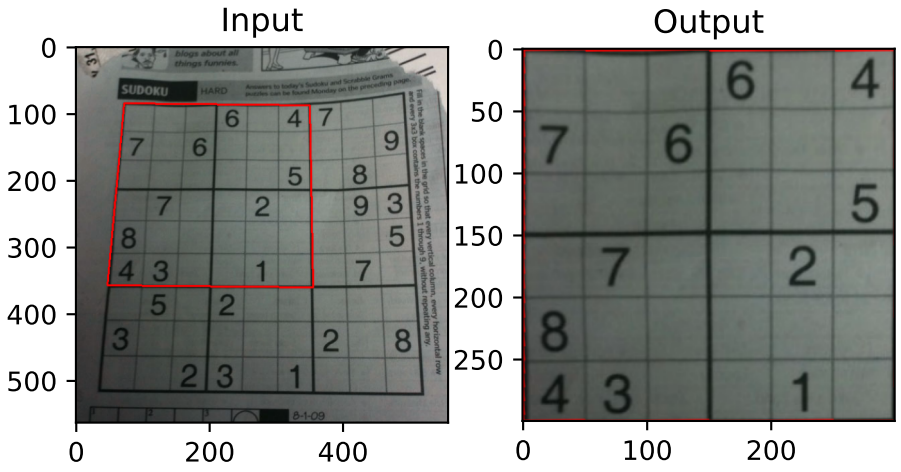
There are situations where we need to transform the perspective of an image, in order to identify objects better, or to perform OCR better. To do so, we need to provide a map of four points from the original image (in the same plane), and the corresponding four points in the transformed image. Here are a couple of examples.

Example 9.1 (Example 1: Sudoku)**.** Here is an example of transforming the perspective.

```
sudoku = cv2.imread('data/sudoku.png')
rows,cols,ch = sudoku.shape

pts1 = np.float32([[73,85], [350, 88], [355, 360], [48, 357]])

pts = np.array(pts1, np.int32)
img2 = cv2.polylines(sudoku, [pts], True, (255, 0, 0), 2 )
plt.imshow(img2);
```



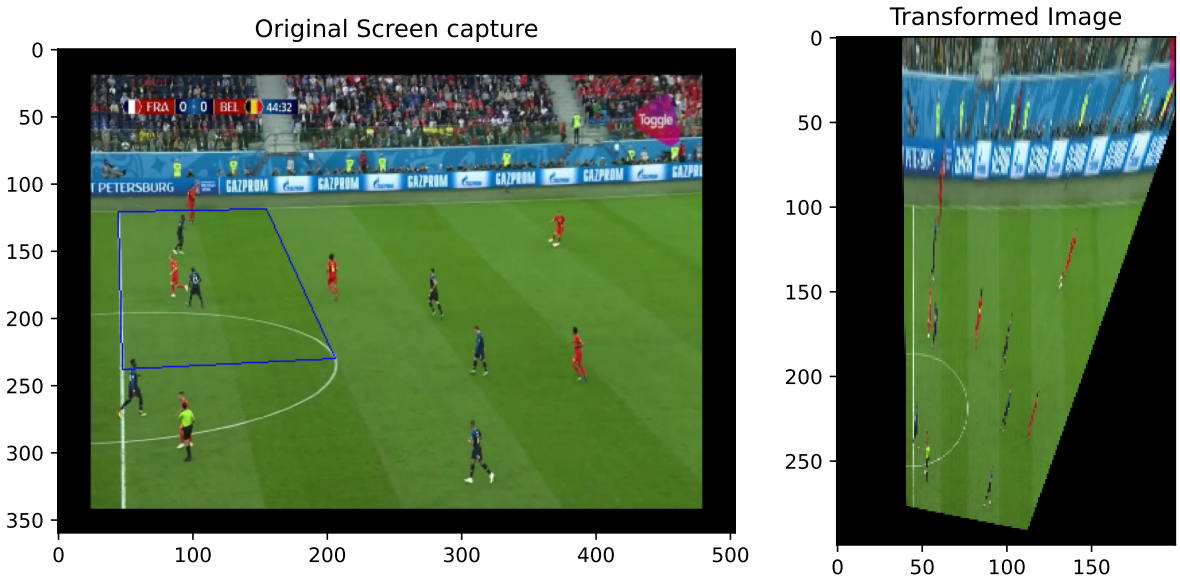

```
pts2 = np.float32([[0,0],[300,0],[300,300],[0,300]])
M = cv2.getPerspectiveTransform(pts1,pts2)
dst = cv2.warpPerspective(sudoku ,M,(300,300))

plt.subplot(121),plt.imshow(sudoku),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output');
```



Example 9.2 (Example 2 - Football).


```
football1 = cv2.imread('data/football1.png')
pts1 = np.float32([[45,121],[48,238],[206,230], [155,118]])
pts2 = np.float32([[45,100], [45,220], [77,220], [77,100]])

M = cv2.getPerspectiveTransform(pts1,pts2)
dst = cv2.warpPerspective(football1, M, (504, 360))

pts = np.array([[45,121],[48,238],[206,230], [155,119]], np.int32)
pts = pts.reshape((-1,1,2))
img2 = cv2.polylines(football1, [pts], True, (255, 0, 0), 1 )
```

```
fig = plt.figure(figsize=(10, 5))
gs = fig.add_gridspec(1, 2, width_ratios=[2, 1])

# First subplot
ax1 = fig.add_subplot(gs[0])
ax1.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
ax1.set_title('Original Screen capture')

# Second subplot
ax2 = fig.add_subplot(gs[1])
ax2.imshow(cv2.cvtColor(dst[:300, :200], cv2.COLOR_BGR2RGB))
ax2.set_title('Transformed Image');
```



9.5 Computer Vision Tasks

Just as we observed there are numerous NLP tasks, the field of computer vision has made great strides in several tasks. Among them are:

1. Object detection
2. Object classification
3. Object tracking
4. Face detection
5. Pose estimation
6. QR/Bar code detection
7. Text detection/extraction

Almost all the models that perform well in the above tasks are deep learning models. In the next few sections, we are going to practice running/configuring some of the above models. Although these models are already trained, it is a little tricky to find the resources and parameters to get them up and running.

There are three repositories of interest:

1. `opencv`: This contains example images and videos, and configuration files for several models.
2. `opencv_extra`: This contains more configuration details for the models, along with routines to download model weights.
3. `opencv_zoo`: This contains a smaller range of models, along with the model weights.

For our course, we have pre-downloaded some models for us to play around with, `models`. However, the general approach to start working on one of these models is as follows:

1. Download the model weights using `opencv_extra/testdata/dnn/download_models.py`
2. Figure out what other configuration files/parameters are needed for this model. Sometimes information is in `opencv/samples/dnn/models.yml`. At other times, you may need to figure it out from the github repository for the particular model.

3. Test it out using one of the sample scripts from `opencv/samples/dnn/*.py`
4. Once you have that working, inspect the script to obtain details on how to call the model programmatically, and proceed from there.

9.6 References

9.6.1 Opencv documentation

1. Python tutorials
2. Changing colour spaces
3. Background subtraction
4. Opencv bootcamp: This is a very useful course on opencv techniques. It will also provide you several more notebooks with template code for object tracking, etc.

9.6.2 Books

A very comprehensive book on vision techniques is by Szeliski (2022). An online version can be found here: Computer Vision: Applications and Algorithms

9.6.3 Github repositories

1. opencv
2. opencv-extra
3. opencv model zoo

Academic References

Agresti, Alan. 2012. *Categorical Data Analysis*. Vol. 792. John Wiley & Sons.

Assimakopoulos, Vassilis, and Konstantinos Nikolopoulos. 2000. "The Theta Model: A Decomposition Approach to Forecasting." *International Journal of Forecasting* 16 (4): 521–30.

Clark, Kevin, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. "What Does Bert Look at? An Analysis of Bert's Attention." *arXiv Preprint arXiv:1906.04341*.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–86.

Dragulescu, Adrian A, and Victor M Yakovenko. 2002. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." *arXiv Preprint Cond-Mat/0211175*.

Draper, NR. 1998. *Applied Regression Analysis*. McGraw-Hill. Inc.

Han, Xiaochuang, Byron C Wallace, and Yulia Tsvetkov. 2020. "Explaining Black Box Predictions and Unveiling Data Artifacts Through Influence Functions." *arXiv Preprint arXiv:2005.06676*.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. "An Introduction to Statistical Learning."

Hyndman, Rob J, and George Athanasopoulos. 2018. *Forecasting: Principles and Practice*. OTexts.

Jurafsky, Daniel, and James H. Martin. 2025. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models*. 3rd ed. https://web.stanford.edu/~jurafsky/slp3/.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. "Distributed Representations of Words and Phrases and Their Compositionality." *Advances in Neural Information Processing Systems* 26.

Molnar, Christoph. 2020. *Interpretable Machine Learning*. Lulu. com.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning. 2014. "Glove: Global Vectors for Word Representation." In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–43.

Sun, Xiaofei, Diyi Yang, Xiaoya Li, Tianwei Zhang, Yuxian Meng, Han Qiu, Guoyin Wang, Eduard Hovy, and Jiwei Li. 2021. "Interpreting Deep Learning Models in Natural Language Processing: A Review." *arXiv Preprint arXiv:2110.10470*.

Szeliski, Richard. 2022. *Computer Vision: Algorithms and Applications*. Springer Nature.

Index