



Classical Machine Learning

Practical Workbook

Dr Rob Collins

Classical Machine Learning

Practical Workbook

Dr Rob Collins

Oxpex Press
Oxford, UK

© 2023 Oxpex Press

All rights are reserved.

Oxpex Press would like to thank the peer reviewers and students who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

Printed and bound in the United Kingdom

Publisher: Oxpex Press.

Publication date: 22nd December 2023

ISBN: 978-1-7385058-0-7

Author: Rob Collins

Email: robert.collins@kellogg.ox.ac.uk

Address: Oxford, UK

Website: <https://www.oxpex.com>

Please direct all enquiries to the author.

Brief Contents

Introduction	i
1. Feature Engineering	6
2. Implementing Gradient Descent	67
3. End-to-end Supervised Learning.....	93
4. Polynomial Regression	113
5. Matrices and Linear Algebra.....	131
6. Clustering	151
7. Trees and Ensemble Methods.....	179
8. Dimensionality Reduction.....	202
9. Reinforcement Learning.....	221
10. Classical Natural Language Processing.....	244
11. Gaussian Mixture Models.....	265
12. Tutor Demonstration 1 : Naïve Approaches to Model Fitting.....	286
13. Student Activity: Naïve Approaches to Model Fitting	295
14. Tutor Demonstration 2 : Non-parametric Models	305
15. Tutor Demonstration 3 : Introduction to Dimensionality Reduction.....	314

Introduction

This laboratory workbook has been written to support the practical workshops for the 'Classical Machine Learning' (CML) module of the MSc in Software Engineering at Oxford University. CML is a one-week, full-time module and is one of many options that students may elect to complete as part of their MSc. This module is set at the introductory level and is intended as a general introduction to Classical Machine Learning for Software Engineers rather than being a detailed, advanced course for Data Scientists. The course addresses the general need for Software Engineers to be aware of the tools, methods and key ideas of Classical Machine Learning.

The use of the term 'Classical' is important here: This course module does not address Neural Networks, Deep Learning or Large Language Models. Those subjects are dealt with by other modules within the Oxford Software Engineering MSc portfolio. This course addresses a broad-range of tools and techniques that should be useful to Software Engineers. Perhaps most importantly it is intended to enable generalist Software Engineers to communicate using a common vocabulary and understanding with specialist Data Scientists.

Somewhat unusually, this book has been published in two forms:

1. For use within laboratory workshops themselves and from which the majority of source code has been omitted. The required code is to be provided by the student!
2. For use subsequent to workshops and providing all of the required source code for each practical workshop. This is the 'solution' text for the tasks identified in item (1)

The text provides much of the information required by students to enable them to discover their own solution to each problem. It develops ideas systematically and provides some code where either (1) It would provide little learning value for a student to develop it or (2) It is too complex, obscure or arduous to develop given the current treatment of the subject thus far.

In the practical workshops for this course students are encouraged to work through each chapter either individually, in pairs or in small groups. At the end of each workshop the course tutor provides a walk-through of the solution and shares the full, un-expurgated version of the book to students study. Thus, students have an opportunity to practice on their own terms whilst also being provided with solutions to the problems.

There is an expectation that during the workshop students will access internet based resources to help them solve each workshop problem. As a matter of preference this is intended to mean the on-line manuals for popular Python Libraries including sklearn, pandas, numpy, matplotlib and others. Students may decide to access other online resources including blog posting and discussion forum and, most recently the Large Language Models such as chatGPT. This is an inevitable reflection of current professional practice. In general students are discouraged from whole-scale 'plundering' of solutions that are available on-line: there is very little learning

advantage in this. Students are encouraged to build note-books that include not only solution code, but such notes, rationale and explanation as they need to help them understand the material and later to act as reference material.

The graphics that appear at the start of each chapter are a matter of whimsy and entertainment. The images connote an alternate history in which Machine Learning has been implemented over a very long history, either through mechanical means or by the curious occurrence of advanced computing devices in ancient history. It goes without saying that none of the images have any relationship to the real history of this discipline.

This is a draft version of this book and the author would be pleased to receive feedback, corrections and comments via robert.collins@kellogg.ox.ac.uk

Chapter 1 - Feature Engineering

Workbook for Student Practical Classes

Dr Rob Collins 2023

2024-07-12

Table of contents

1 Feature Engineering	1 - 5
1.1 Introduction	1 - 5
1.2 Useful References	1 - 5
1.2.1 Books	1 - 5
1.2.2 Python libraries used in this section	1 - 6
1.3 Instructions for Students	1 - 6
1.4 Load the required libraries	1 - 7
1.5 Importing CSV data into a pandas dataframe	1 - 8
1.6 Overall description of the types of data	1 - 8
1.7 Displaying Dataframes and a sub-set of columns from dataframes	1 - 9
1.8 Visual review of category data stored in features of type ‘object’	1 - 11
1.9 Using seaborn ‘countplot’ to create a chart showing a count of each value in an category data-set	1 - 11
1.10 Re-review the number of non-null data items in the dataframe	1 - 14
1.11 Using missingno to identify missing data	1 - 19
1.12 Is there a correlation in missing data?	1 - 19
1.13 Is bulk deletion of rows with missing data a good strategy?	1 - 21
1.14 Overview of data distribution and potential relationships between features .	1 - 21
1.15 Visual review of the distribution (shape) of data	1 - 23
1.16 Reviewing outliers using box-plots	1 - 24
1.17 Violin Plots - Another way of displaying the distribution of data and outliers	1 - 30
1.18 Outliers : To Remove or not to remove?	1 - 30
1.19 Changing outliers to NaN based on the number of standard deviations from the mean	1 - 30
1.20 Imputing missing values	1 - 37
1.20.1 A simple method based on either the mean or median	1 - 37
1.20.2 A more sophisticated approach to Imputation	1 - 40
1.21 Imputing categorical variables	1 - 42
1.22 Transforming the data to a standard scale	1 - 44
1.22.1 We need to record the values used for the transformation	1 - 48

1.23	Translating Categorical Data into a Form that can be used in Models	1 - 50
1.23.1	Transforming Ordinal data	1 - 50
1.23.2	Transforming Categorical Data to ‘1-hot-encoded’ data	1 - 51
1.24	Other data visualizations that may provide insight during early exploratory data analysis	1 - 52
1.25	Other Feature engineering tools (out of scope for this course)	1 - 61
1.25.1	QQ Plot	1 - 61

List of Figures

1	Medieval stone mason calculating the exact proportion of features for a freeze according to secret rules of proportion (C. 1100)	1 - 4
2	Bar-chart showing the number of items of each type within the ‘ag’ feature .	1 - 12
3	Bar-chart showing the number of items of each type within the ‘bg’ feature .	1 - 13
4	Seaborn version of Bar-chart for ‘ag’ feature	1 - 14
5	Seaborn version of Bar-chart for ‘bg’ feature	1 - 15
6	‘ag’ feature after removal of values	1 - 15
7	‘bg’ feature after removal of values	1 - 16
8	missingno ‘matrix’ function showing missing data in the df dataframe . . .	1 - 19
9	missingno ‘bar’ function showing missing data in the df dataframe	1 - 20
10	missingno ‘heatmap’ function showing missing data in the df dataframe . .	1 - 20
11	missingno ‘bar’ chart showing impact of dropping rows containing missing values	1 - 21
12	High-level overview of all data using the ‘scatter_matrix’ function	1 - 22
13	Visualisation of a sub-set of the dataframe using ‘scatter_matrix’ function .	1 - 23
14	Histogram showing detailed view of data distribution for V1	1 - 24
15	Histogram showing detailed view of data distribution for V2	1 - 25
16	Histogram showing detailed view of data distribution for V3	1 - 25
17	Box-plot showing relative distribution of features and outliers	1 - 26
18	Box-plots showing relative distribution of features and outliers	1 - 27
19	Violin plots: A different view of data distributions and outliers	1 - 31
20	Histograms of data after removal of outliers	1 - 33
21	Histograms of data after removal of outliers	1 - 33
22	Histograms of data after removal of outliers	1 - 34
23	Histograms of data after removal of outliers	1 - 34
24	Histograms of data after removal of outliers	1 - 35
25	Results of imputing missing values of Th	1 - 38
26	Results of imputing missing values for cost variables	1 - 39
27	Final results of imputing missing values for all numerical variables . .	1 - 39
28	msno matrix of dataframe after imputing with IterativeImputer	1 - 42
29	Final results of imputing all missing values	1 - 43
30	Box-plot of all variables showing significant scale differences	1 - 46
31	Box-plot of all variables after standardization	1 - 47
32	kde plot of the data	1 - 53

33	kde plots of selected features using a common x axis scale	1 - 59
34	2-Dimensional kde (surface contour) plot of selected features	1 - 60
35	Q-Q plot to help show deviation from a Gaussian distribution	1 - 61



Figure 1: Medieval stone mason calculating the exact proportion of features for a freeze according to secret rules of proportion (C. 1100)

1 Feature Engineering

1.1 Introduction

In this workshop we will focus on ‘Feature Engineering’ - that is the ‘cleaning’, organising and preparation of data ahead of analysis and modelling. Whilst many student tutorials are based around ‘clean’, pre-prepared data this is, in fact, an unreasonable scenario. Most real-world data suffers from a variety of problems that need to be dealt with ahead of analysis and modelling. Problems include:

- Missing values (null data)
- ‘Outliers’ - data that may be identified as invalid as it falls so far outside of the range of other data
- Category (string) data that needs to be translated into numerical values to enable statistical processing
- Multiple features that span over very different ranges - leading to issues in modelling
- Generation of new features based on original source data to represent known features of the real-world problem

It is perhaps surprising that a significant fraction of the overall effort for a Machine learning / Data-Science project is often associated with this activity. It may not be the most exciting aspect of ML/DS, but it can be technically challenging and is extremely important.

1.2 Useful References

The following references are useful support for this chapter:

1.2.1 Books

Galli, Soledad (2022) “**Python feature engineering cookbook**”

https://solo.bodleian.ox.ac.uk/permalink/44OXF_INST/35n82s/alma991022130657707026

Ozdemir, Sinan (2022) “**Feature engineering bookcamp**”

https://solo.bodleian.ox.ac.uk/permalink/44OXF_INST/35n82s/alma991022083570907026

Zheng, Alice,Casari, Amanda, (2018) “**Feature engineering for machine learning : principles and techniques for data scientists**”

https://solo.bodleian.ox.ac.uk/permalink/44OXF_INST/35n82s/alma991022176399807026

1.2.2 Python libraries used in this section

1. **pandas** - Python Data Analytics Library
 - <https://pandas.pydata.org/>
2. **numpy** - Scientific computing with Python
 - <https://numpy.org/>
3. **matplotlib** - Visualisation with Python
 - <https://matplotlib.org/>
4. **seaborn** - Statistical data visualisation
 - <https://seaborn.pydata.org/>
5. **missingno** - Visualising missing data in dataframes
 - <https://github.com/ResidentMario/missingno>
6. **scikit-learn** - Machine Learning Library for Python
 - <https://scikit-learn.org/stable/>

1.3 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class

8. After class, enter and run any remaining code blocks that you have not been able to complete in class

1.4 Load the required libraries

For this practical session we will need the following libraries:

- ‘pandas’: Conventionally given the reference name ‘pd’
- ‘scatter_matrix’ : from pandas.plotting
- ‘numpy’ : Conventionally given the reference name ‘np’
- ‘seaborn’ : Conventionally given the reference name ‘sns’
- ‘matplotlib.pyplot’ : Conventionally given the reference name ‘plt’
- ‘matplotlib’ :
- ‘figure’ : from matplotlib.pyplot
- ‘missingno’ : Conventionally given the reference name ‘msno’
- ‘enable_iterative_imputer’ : from sklearn.experimental
- ‘IterativeImputer’ : from sklearn.impute
- ‘SimpleImputer’ : from sklearn.impute import
- ‘StandardScaler’ : from sklearn.preprocessing

‘pandas’ is used frequently in Python-based Machine Learning projects as it provides easy manipulation of tabulated data. The library includes functions to load and save data from a variety of file types; the ability to filter, insert, delete, modify and display rows and columns (and indeed, sub-sets of rows and columns). The library also provides some easy-to-use charting functions suited for tabular data.

‘scatter_matrix’ is one of several pandas methods for plotting data. Specifically, this method creates a grid of graphs showing data distributions on a feature-by-feature basis.

‘numpy’ is a hugely powerful and fast numerical processing library. It contains a huge range of numerical algorithms to process scalars, arrays and multi-dimensional arrays.

‘seaborn’ is a graphical library providing easy access to some attractive charts and graphics.

The ‘matplotlib’ library functions are used ubiquitously in Python-based Machine Learning, scientific and statistical projects to plot graphs and charts.

‘missingno’ is a group of tools designed to help identify missing data and any patterns in missing data.

We will use the ‘sklearn’ library extensively during this course as it includes a wide range of Machine Learning algorithms that are accessible by a highly unified interface. In this case we will be using the ‘IterativeImputer’ and ‘SimpleImputer’ methods that enable missing values to be imputed (inferred and filled in) in data-frames. We will also use the sklearn ‘StandardScaler’ method which scales all data to a common range.

Create a cell to import all of these libraries.

 Your code here

Pandas tables display more neatly if you set an appropriate precision for display. In this case, it will be enough to show only 2 digits after the decimal point .. so add the following code to achieve that.

 Tutor provided code

```
pd.set_option('precision', 2)
```

1.5 Importing CSV data into a pandas dataframe

Load the data from the file ‘machine_params_and_cost_v2.csv’ into a dataframe called ‘df’ using the pandas ‘read_csv()’ function.

 Your code here

1.6 Overall description of the types of data

Obtain some overall descriptive data about the data using the pandas functions ‘info’ and ‘describe’.

Display the overall ‘shape of the dataframe using the ’shape’ attribute from Pandas

```
df.shape = (2000, 31)
```

Then use the pandas ‘info()’ function to learn some basic information about the features (columns) in the dataframe

 Your code here

```
df.info() =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   ag              2000 non-null    object 
 1   bg              2000 non-null    object 
```

```

2   T1                  1960 non-null    float64
3   V1                  1965 non-null    float64
4   A1                  1964 non-null    float64
5   P1                  1967 non-null    float64
6   B1                  1969 non-null    float64
7   T2                  1963 non-null    float64
8   V2                  1965 non-null    float64
9   A2                  1950 non-null    float64
10  P2                  1955 non-null    float64
11  B2                  1968 non-null    float64
12  T3                  1960 non-null    float64
13  V3                  1967 non-null    float64
14  A3                  1957 non-null    float64
15  P3                  1963 non-null    float64
16  B3                  1960 non-null    float64
17  T4                  1959 non-null    float64
18  V4                  1972 non-null    float64
19  A4                  1957 non-null    float64
20  P4                  1977 non-null    float64
21  B4                  1964 non-null    float64
22  Contamination_Defect 2000 non-null    int64
23  Crystalisation_Defect 2000 non-null    int64
24  Ion_Diffusion_Defect 2000 non-null    int64
25  Burnishing_Defect    2000 non-null    int64
26  m1_cost              1966 non-null    float64
27  m2_cost              1963 non-null    float64
28  m3_cost              1959 non-null    float64
29  m4_cost              1957 non-null    float64
30  ID                   2000 non-null    int64
dtypes: float64(24), int64(5), object(2)
memory usage: 484.5+ KB

```

We appear to have a mixture of types of data: ‘object’, ‘float64’ and ‘int64’.

It is also clear that many of the features have missing data (the count of ‘non-null’ items is less than the total number of items). That is a problem we will need to deal with shortly. For now, let’s just display the data to get a feel for it..

1.7 Displaying Dataframes and a sub-set of columns from dataframes

Add a cell to display the contents of your Pandas dataframe. When using Jupyter notebooks that is as easy as simply using the name of the dataframe (or any variable) as the last line of the cell.

When doing this as an exercise then I suggest that you do exactly this .. after all, jupyter provides a slide-bar that enables you to view the full width of a dataframe.

In my case, because I want to format this to fit onto a printed page I am going to use several cells to display smaller groups of columns at a time:

 Tutor provided code

either simply use

`df`

to display a whole dataframe. Or use the ‘loc’ method to access a sub-set of features:

`df.loc[0:5,:'ag':'B1']`

to display a sub-set of columns based on the name of the features (column headings). Finally you could use the ‘iloc’ method to access numbered ranges of rows and columns:

`df.iloc[0:5,2:5]`

	ag	bg	T1	V1	A1	P1	B1
0	Low	Argon	93.99	445746.58	299.44	30.67	61.76
1	Low	Argon	57.16	389648.47	275.28	49.96	132.62
2	Medium	Argon	81.26	396609.84	235.95	15.26	130.84
3	Medium	Neon	125.48	165185.46	221.71	14.67	116.94
4	Medium	Argon	138.01	626895.16	202.00	22.25	142.09
5	Low	Neon	122.15	293819.35	138.81	38.79	127.86

	T2	V2	A2	P2	B2	T3	V3	A3	P3	B3
0	660.21	1.45e+06	468.43	83.22	244.53	370.20	786363.50	236.04	261.51	903.01
1	502.63	4.53e+06	435.62	79.78	211.53	398.50	574039.33	235.94	258.36	784.82
2	501.80	2.89e+06	113.89	60.25	244.97	656.17	538461.10	100.47	144.16	821.06
3	437.94	2.67e+06	148.94	48.69	307.95	450.63	806097.99	551.39	127.93	786.11
4	577.79	8.95e+05	242.34	62.91	326.96	448.58	867053.61	366.61	227.83	841.30
5	584.76	4.53e+06	368.04	43.58	217.88	528.33	811204.00	201.31	142.12	888.82

	T4	V4	A4	P4	B4
0	373.67	551459.56	388.98	192.87	349.56
1	349.54	316078.98	816.77	387.36	340.56
2	344.41	384425.65	220.65	327.44	355.24
3	362.83	461873.55	512.62	444.46	349.99
4	331.01	337105.69	644.09	363.18	332.95
5	222.27	445837.13	588.19	247.37	317.78

	Contamination_Defect	Crystallisation_Defect	Ion_Diffusion_Defect	Burnishing_Defect
0	0	0	0	0
1	0	1	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	0	0
5	0	1	0	0

	m1_cost	m2_cost	m3_cost	m4_cost	ID
0	1139.53	2064.62	1156.56	840.47	0
1	1008.11	NaN	869.71	523.19	1
2	1024.36	4008.28	822.97	614.95	2
3	480.40	NaN	1184.28	719.92	3
4	1565.70	1309.50	1266.84	551.15	4
5	782.76	6215.86	1190.54	697.05	5

1.8 Visual review of category data stored in features of type ‘object’

We want to plot bar charts that indicate the number of occurrence of each category value in a feature.

Write a function with this signature:

```
def count_ordinal(df, feature):
```

1. Use the pandas ‘value_counts()’ function to count the frequency of each ordinal value in the named feature
2. Use the pandas ‘plot(kind=’bar’)’ function to plot the result as a bar chart

 Your code here

Execute that function to get a break-down of the data in the columns called ‘ag’ and ‘bg’

 Your code here

1.9 Using seaborn ‘countplot’ to create a chart showing a count of each value in an category data-set

Create a new function, similar to the last called ‘count_ordinal_sns’ For this second function we will use a method from the seaborn library: ‘sns.countplot()’ to plot the results.

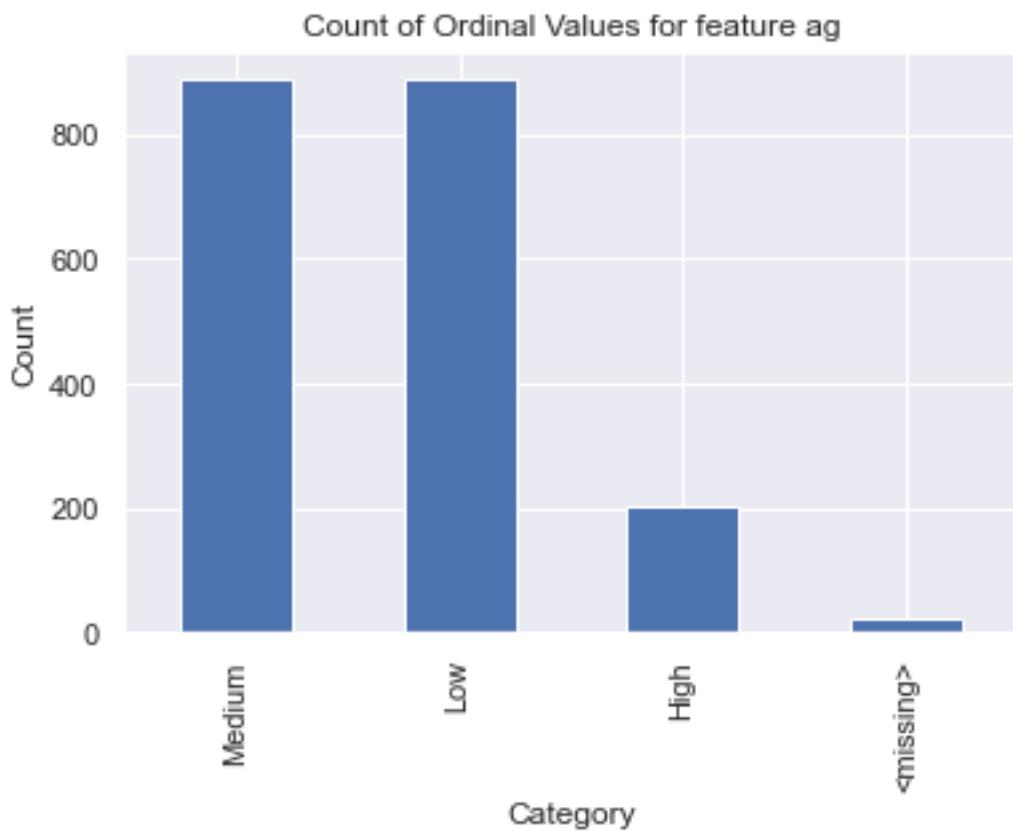


Figure 2: Bar-chart showing the number of items of each type within the 'ag' feature

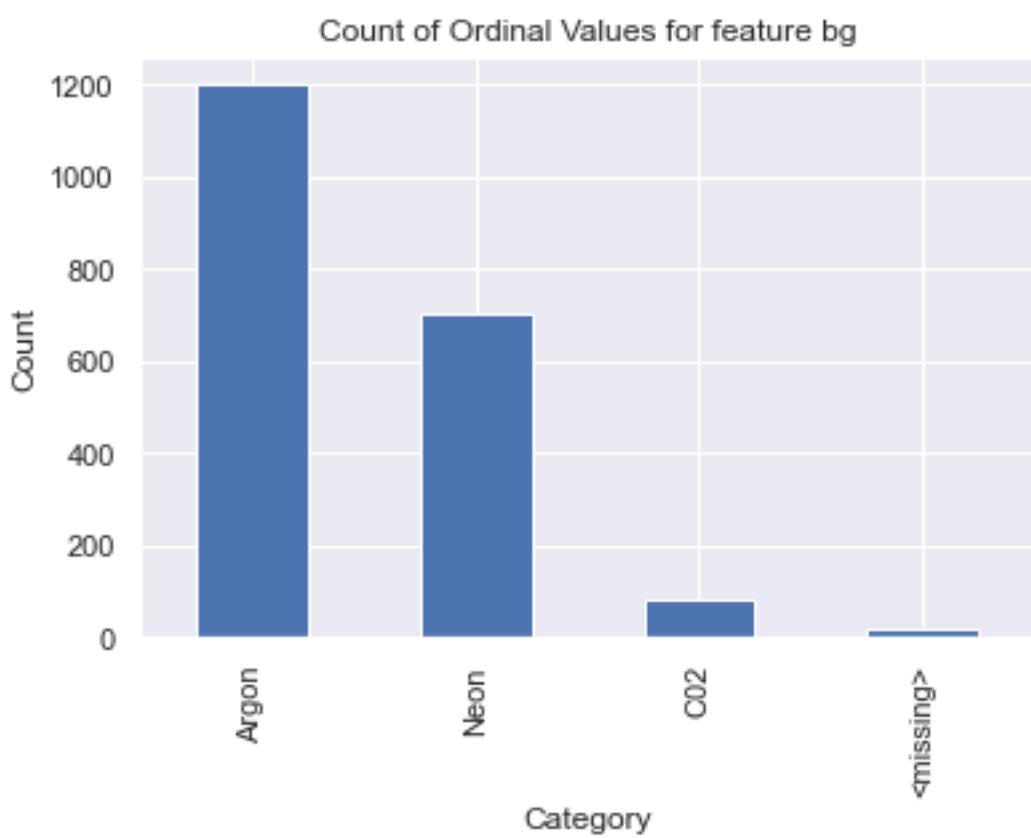


Figure 3: Bar-chart showing the number of items of each type within the 'bg' feature

This time, plot the chart as a horizontal rather than a vertical bar-chart (hint: use the sns.countplot 'y' parameter to pass the name of the feature).

Add cells to execute this function on both the 'ag' and 'bg' feature.

💡 Your code here

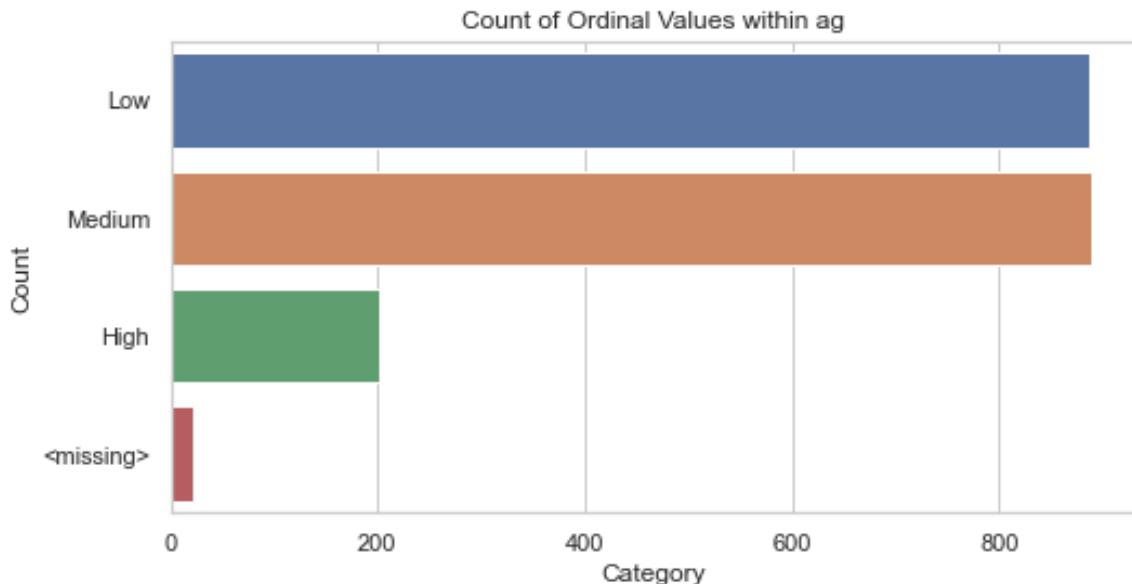


Figure 4: Seaborn version of Bar-chart for 'ag' feature

It is clear that both the 'ag' and 'bg' columns *do* have missing data .. however, in this case it has been marked explicitly as a string '<missing>'. So that we can deal with this missing data in a uniform way we should convert every item labelled as 'missing' into a 'NaN' value. Write a line of code that converts each cell labelled \<missing\> into a 'NaN' value

💡 Your code here

Re-plot the charts for 'ag' and 'bg'

💡 Your code here

1.10 Re-review the number of non-null data items in the dataframe

We have now converted the items to 'NaN' - so we can quickly review the new counts. Use the '.info()' function within Pandas to get a list of the types of each object and a count of the number of non-null items

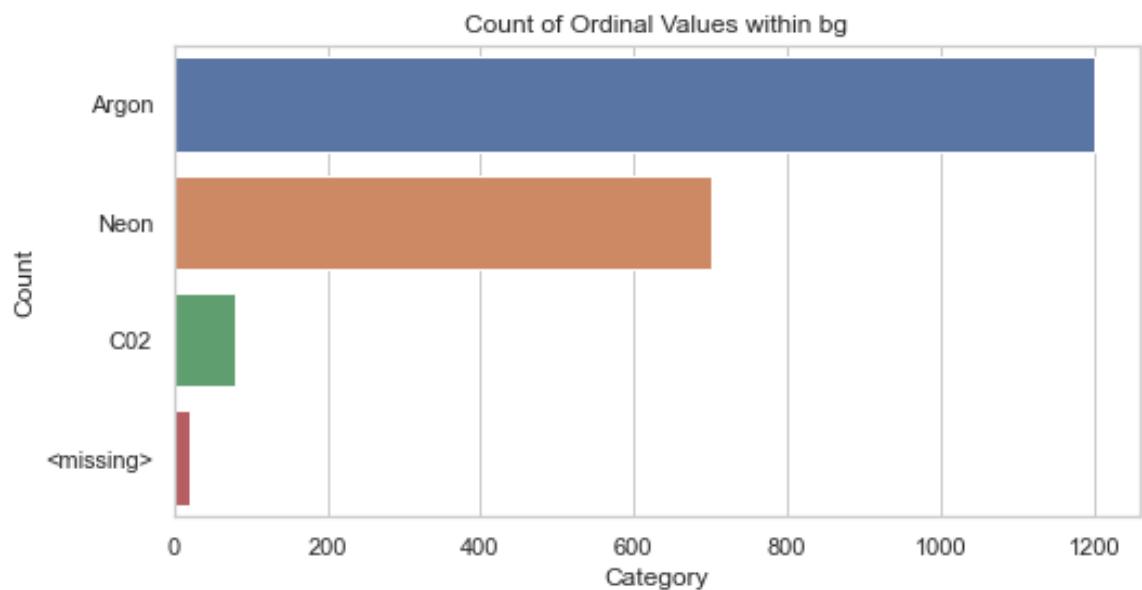


Figure 5: Seaborn version of Bar-chart for ‘bg’ feature

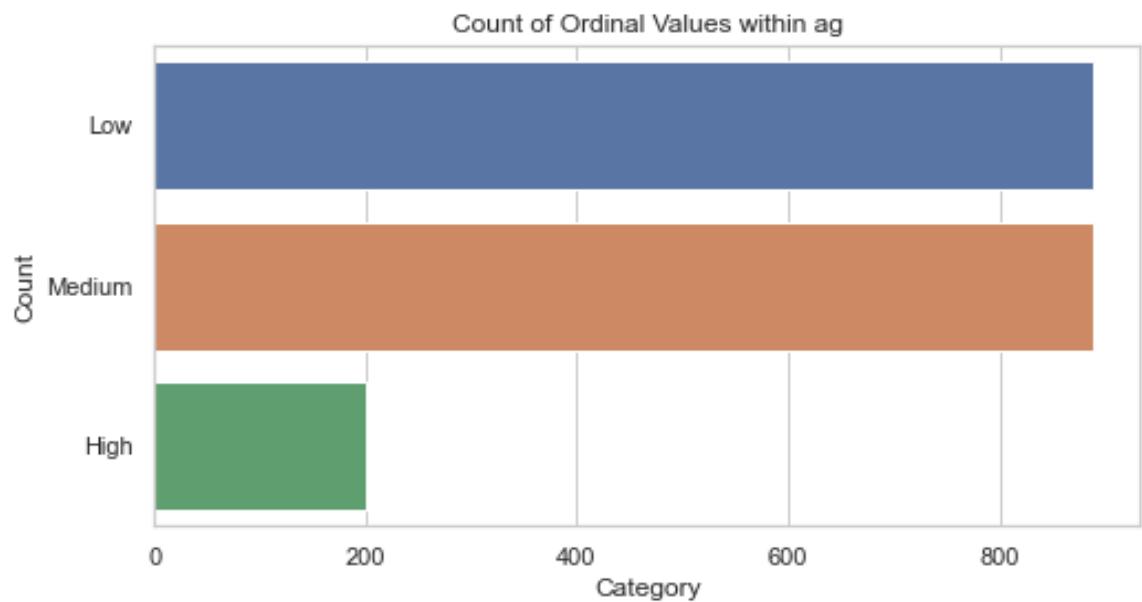


Figure 6: ‘ag’ feature after removal of values

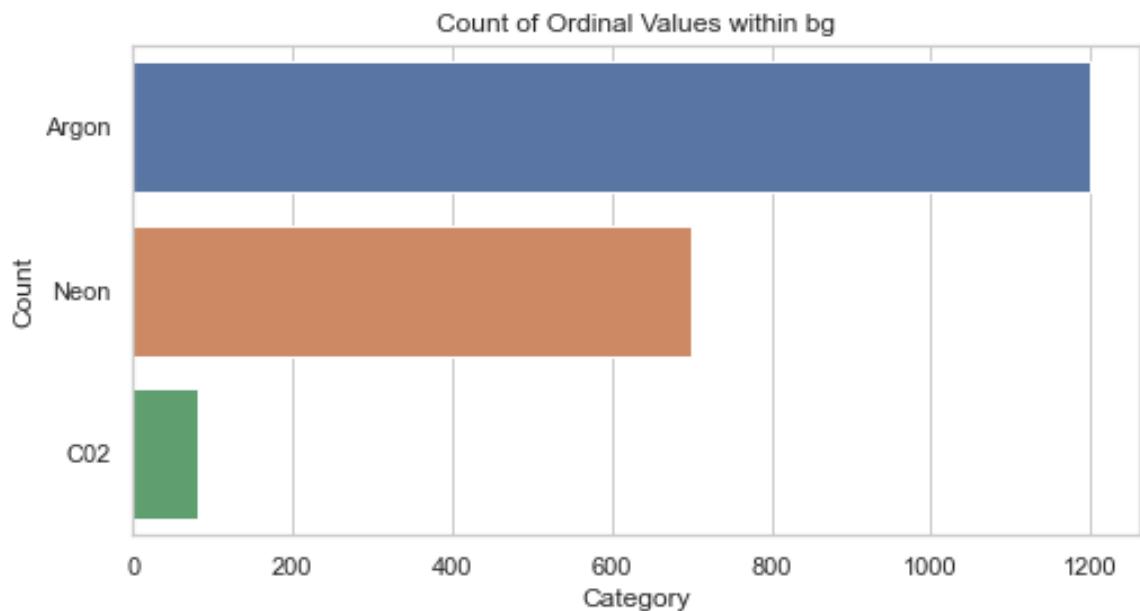


Figure 7: 'bg' feature after removal of values

Your code here

```
df.info() =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   ag              1978 non-null    object 
 1   bg              1980 non-null    object 
 2   T1              1960 non-null    float64
 3   V1              1965 non-null    float64
 4   A1              1964 non-null    float64
 5   P1              1967 non-null    float64
 6   B1              1969 non-null    float64
 7   T2              1963 non-null    float64
 8   V2              1965 non-null    float64
 9   A2              1950 non-null    float64
 10  P2              1955 non-null    float64
 11  B2              1968 non-null    float64
 12  T3              1960 non-null    float64
 13  V3              1967 non-null    float64
 14  A3              1957 non-null    float64
```

```

15 P3                      1963 non-null   float64
16 B3                      1960 non-null   float64
17 T4                      1959 non-null   float64
18 V4                      1972 non-null   float64
19 A4                      1957 non-null   float64
20 P4                      1977 non-null   float64
21 B4                      1964 non-null   float64
22 Contamination_Defect    2000 non-null   int64
23 Crystalisation_Defect   2000 non-null   int64
24 Ion_Diffusion_Defect   2000 non-null   int64
25 Burnishing_Defect      2000 non-null   int64
26 m1_cost                  1966 non-null   float64
27 m2_cost                  1963 non-null   float64
28 m3_cost                  1959 non-null   float64
29 m4_cost                  1957 non-null   float64
30 ID                      2000 non-null   int64
dtypes: float64(24), int64(5), object(2)
memory usage: 484.5+ KB

```

We won't be needing the 'ID' field for our models and we should not treat it as a modelling variable. Therefore remove it using the Pandas 'drop(columns=[feature_to_drop])' method.

 Your code here

Now review the remaining numerical data using the pandas 'describe()' function.

(As previously - you can do this for the whole dataframe, however in my version I am going to present this a few columns at a time to improve layout on a printed page)

 Your code here

	T1	V1	A1	P1	B1
count	1960.00	1.96e+03	1964.00	1967.00	1969.00
mean	99.10	4.14e+05	160.66	30.93	121.62
std	39.51	1.76e+05	103.91	17.62	32.54
min	-5.89	1.13e+05	-4.79	-19.05	27.41
25%	74.86	2.98e+05	82.94	18.92	101.40
50%	97.08	3.85e+05	155.88	30.11	119.06
75%	119.68	5.07e+05	230.08	41.58	139.35
max	372.29	1.67e+06	909.03	143.61	346.74

	T2	V2	A2	P2	B2	T3	V3	A3
count	1963.00	1.96e+03	1950.00	1955.00	1968.00	1960.00	1.97e+03	1957.00
mean	516.63	2.14e+06	258.43	70.30	303.35	421.75	7.63e+05	351.99
std	83.54	1.25e+06	157.43	23.46	49.19	91.56	1.70e+05	203.39
min	286.76	2.72e+05	5.02	18.69	159.16	177.12	3.83e+05	36.60
25%	465.70	1.22e+06	136.73	54.84	272.76	367.53	6.61e+05	202.92
50%	513.71	1.92e+06	253.35	69.22	301.71	417.39	7.73e+05	340.92
75%	561.65	2.85e+06	367.41	84.63	329.98	467.99	8.70e+05	489.61
max	1108.91	1.13e+07	1328.60	245.10	651.37	1087.58	1.87e+06	1746.72

	P3	B3	T4	V4	A4	P4	B4
count	1963.00	1960.00	1959.00	1.97e+03	1957.00	1977.00	1964.00
mean	166.63	822.62	354.76	4.54e+05	558.10	272.99	339.35
std	96.70	109.28	49.88	1.29e+05	317.56	130.86	15.94
min	-1.81	493.03	215.57	2.00e+05	76.35	50.16	278.93
25%	93.09	758.74	324.68	3.72e+05	314.60	159.30	328.25
50%	166.35	815.04	352.44	4.51e+05	547.05	271.76	339.26
75%	234.27	880.06	380.45	5.23e+05	774.63	388.90	350.24
max	815.58	1613.43	709.47	1.45e+06	2725.50	499.50	400.23

	Contamination_Defect	Crystallisation_Defect	Ion_Diffusion_Defect	Burnishing_Defect
count	2000.00	2000.00	2000.00	2000.00
mean	0.03	0.40	0.03	0.05
std	0.17	0.49	0.17	0.22
min	0.00	0.00	0.00	0.00
25%	0.00	0.00	0.00	0.00
50%	0.00	0.00	0.00	0.00
75%	0.00	1.00	0.00	0.00
max	1.00	1.00	1.00	1.00

	m1_cost	m2_cost	m3_cost	m4_cost
count	1966.00	1963.00	1959.00	1957.00
mean	1040.60	2887.71	1112.66	698.42
std	336.87	1397.33	189.27	142.04
min	356.98	473.90	612.32	365.92
25%	789.12	1733.70	985.96	596.77
50%	995.24	2661.52	1137.73	702.08
75%	1275.79	3900.20	1266.22	798.29
max	1950.90	6657.34	1442.03	1035.02

1.11 Using missingno to identify missing data

It is often useful to see if there are any patterns in missing data - either within or between features. ‘Missingno’ is a useful library for visualising missing data.

Use the ‘matrix()’ function of missingno to visually review any missing data.

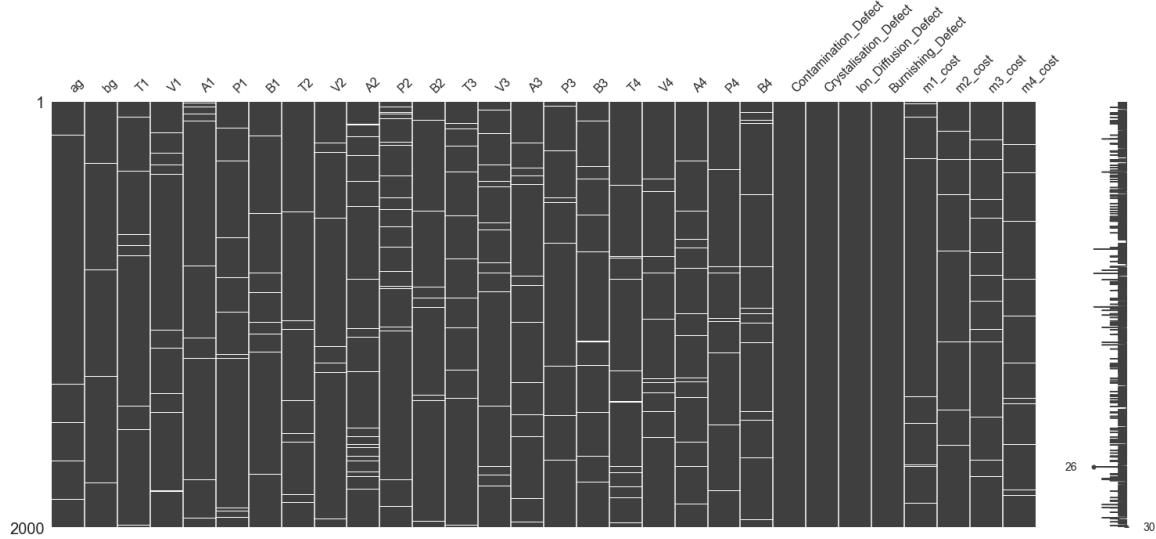


Figure 8: missingno ‘matrix’ function showing missing data in the df dataframe

Add a cell to use the alternate visualization provided by the ‘bar()’ function of missingno:

💡 Your code here

1.12 Is there a correlation in missing data?

We can look for correlations in missing data. Sometimes we find that having data available in one feature (column) is linked with (correlated to) data being present in another feature.

Use the missingno ‘heatmap(df)’ functon to create a correlation matrix of missing values

In the resulting graphic, any dark blue squares indicate a high correlation between two features.

💡 Your code here

Reviewing this heatmap, there does not appear to be any particularly systematic (correlated) gaps in the data.

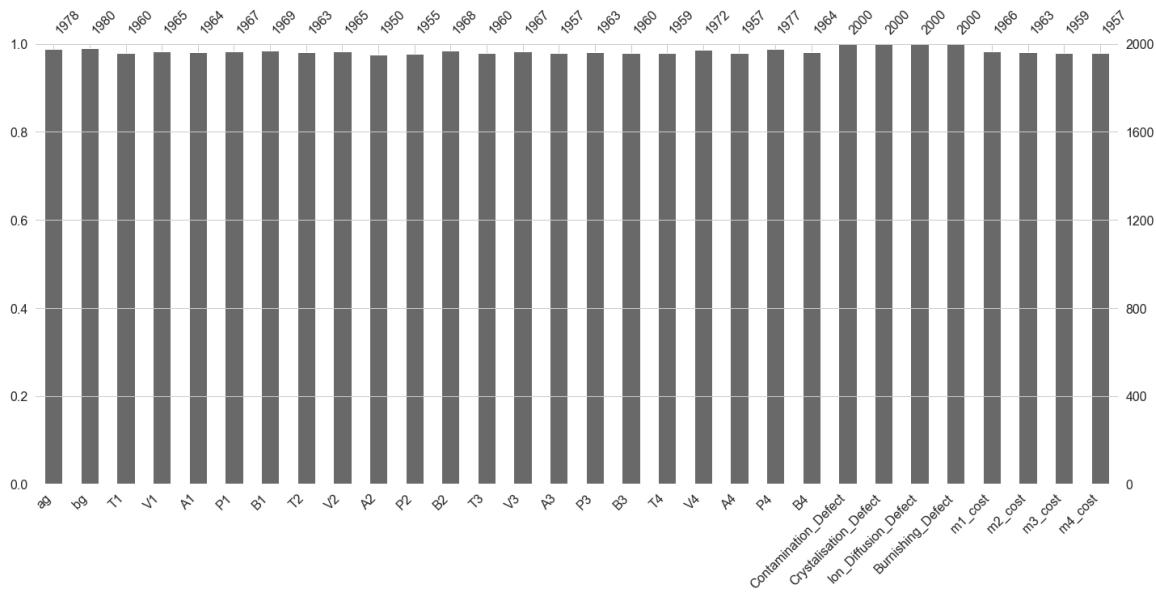


Figure 9: missingno ‘bar’ function showing missing data in the df dataframe

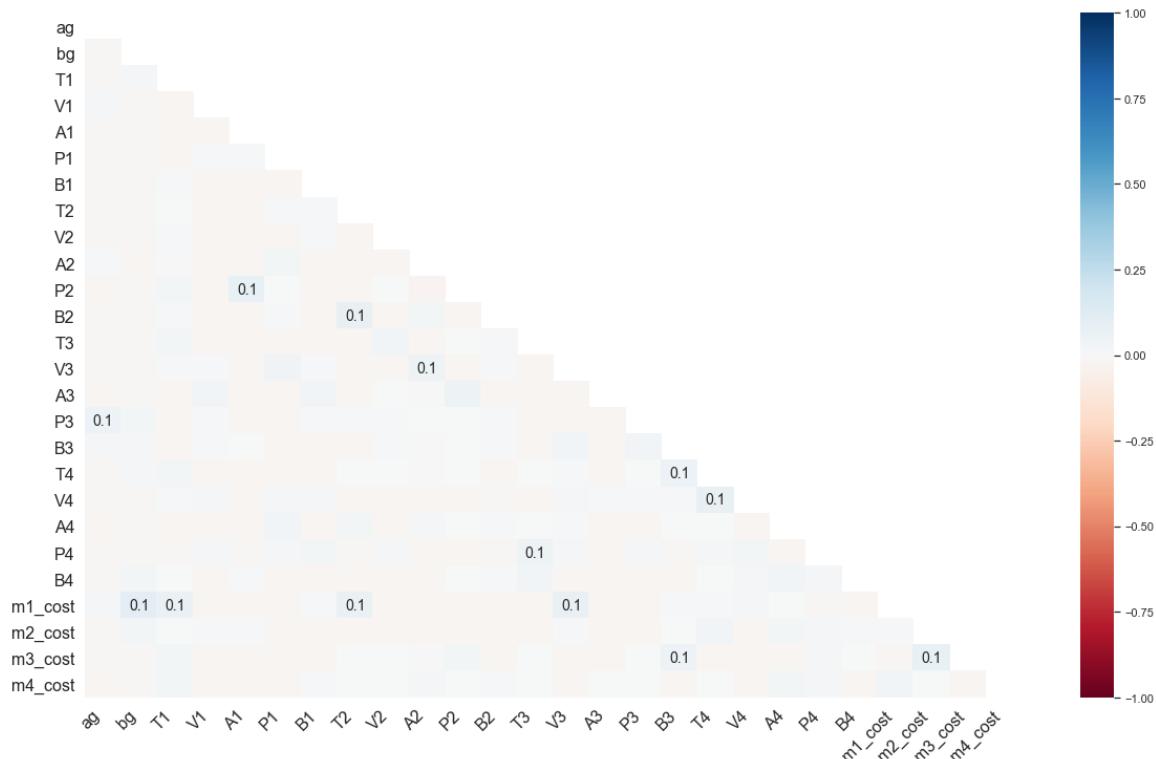


Figure 10: missingno ‘heatmap’ function showing missing data in the df dataframe

1.13 Is bulk deletion of rows with missing data a good strategy?

Just as an experiment, let's try deleting all rows of data that contain 'NaN' values, noting that this is not normally an efficient way of dealing with the problem unless there is a lot of data and few 'NaN' values. Create a new variable 'df_no_nan' containing a copy of the 'df' dataframe that has been filtered to remove all rows containing 'NaN' values.

Hint : use the pandas 'dropna' function to achieve this.

Then replot using missingno.bar() to visualise the impact.

Your code here

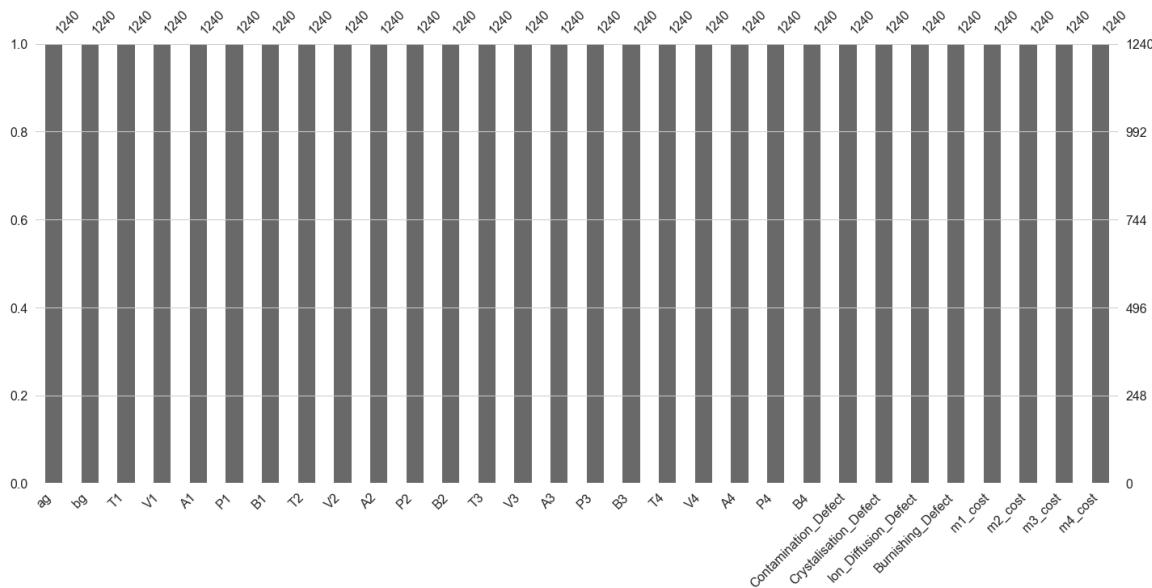


Figure 11: missingno 'bar' chart showing impact of dropping rows containing missing values

It is clear that we have lost many rows of data here - nearly 800. This is a large fraction of our overall data-set, and data is hugely valuable. The conclusion here is that bulk deletion of rows may be a costly operation in terms of data loss. In many cases we will want to use a more sophisticated approach to missing data. Such approaches are covered in later sections of this workbook.

1.14 Overview of data distribution and potential relationships between features

Obtain an overall visualization of the distribution of each feature and the relationships between features using the 'scatter_matrix' function from 'pandas.plotting'.

 Your code here

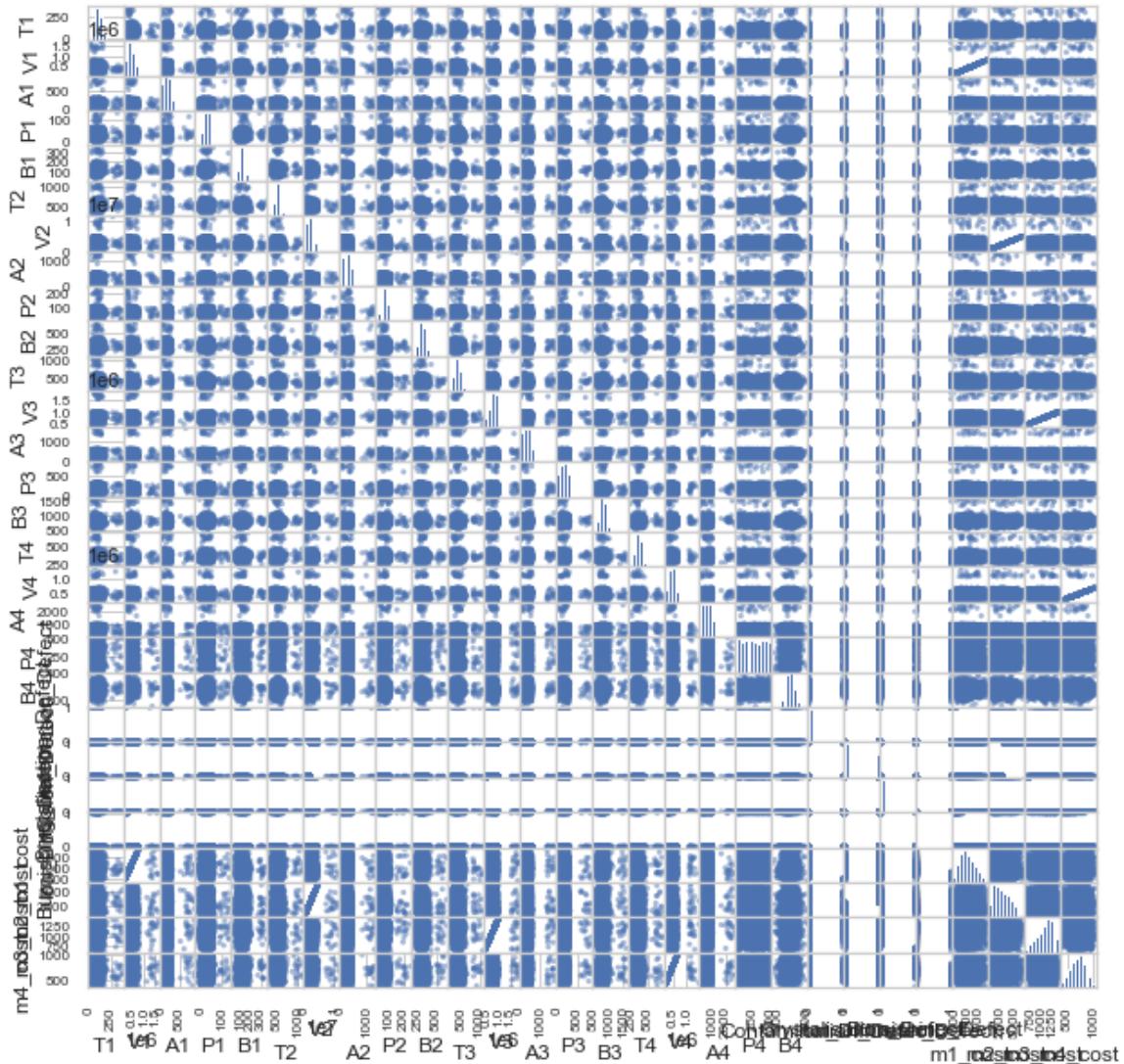


Figure 12: High-level overview of all data using the ‘scatter_matrix’ function

If you have a dataframe with many features you will often find that text display becomes corrupted in this chart. Additionally, the details are really too small to review effectively. Thus you may also find it useful to view sub-sets of your entire dataframe.

Create a list of the features you want to display in your subset (in this case, let's choose T1, B1, A1 and V1) and call the variable 'features'.

Then make another call to ‘scatter_matrix’ passing in ‘df[features]’ rather than the complete dataframe

 Your code here

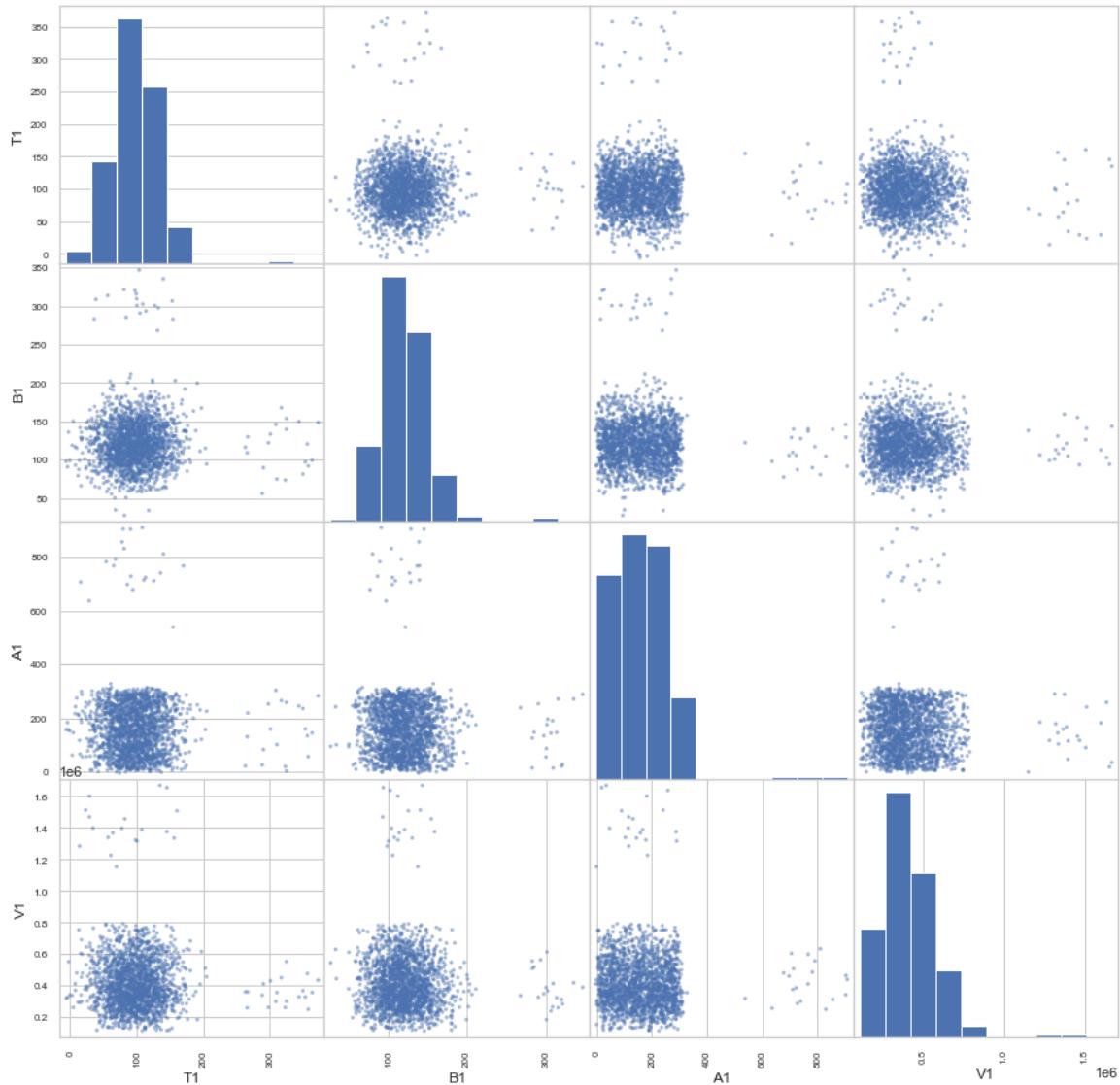


Figure 13: Visualisation of a sub-set of the dataframe using ‘scatter_matrix’ function

1.15 Visual review of the distribution (shape) of data

In order to get a better view of the distribution of specific features, create a function called ‘histogram’ that takes three parameters: 1. ‘df’ : the dataframe containing the data 2. ‘feature’ : the column (feature) name we wish to plot 3. ‘bins’ : The number of frequency bins (vertical bars) we wish to show in our plot

 Your code here

Use the function ‘histogram’ to review some of the features in more detail. In particular, plot histograms of T1, V1 and A1 with 20 bins

 Your code here

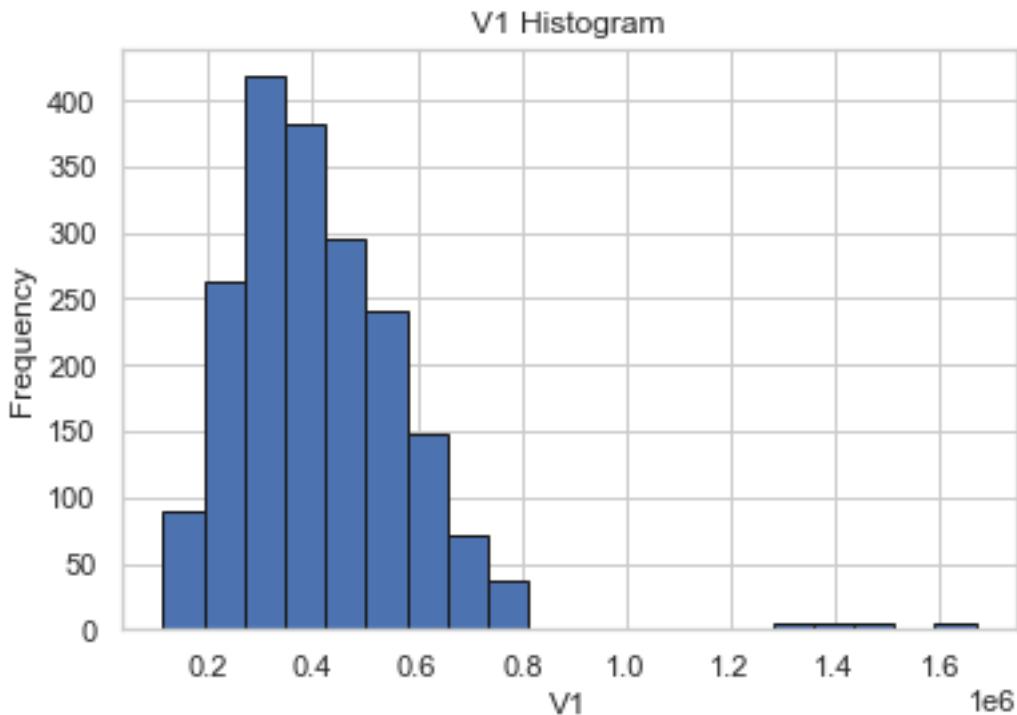


Figure 14: Histogram showing detailed view of data distribution for V1

We should notice some things from this:

- The features have a variety of distributions (normal, triangle (log-normal?) and uniform)
- There are a significant number of outliers
- We already know that there are missing values

Should we be removing those outliers? If we do we might delete specific, ‘high-information’ data from the data-sets. I.e. those outliers may mean something important!

1.16 Reviewing outliers using box-plots

Let’s do a closer review of those outliers...

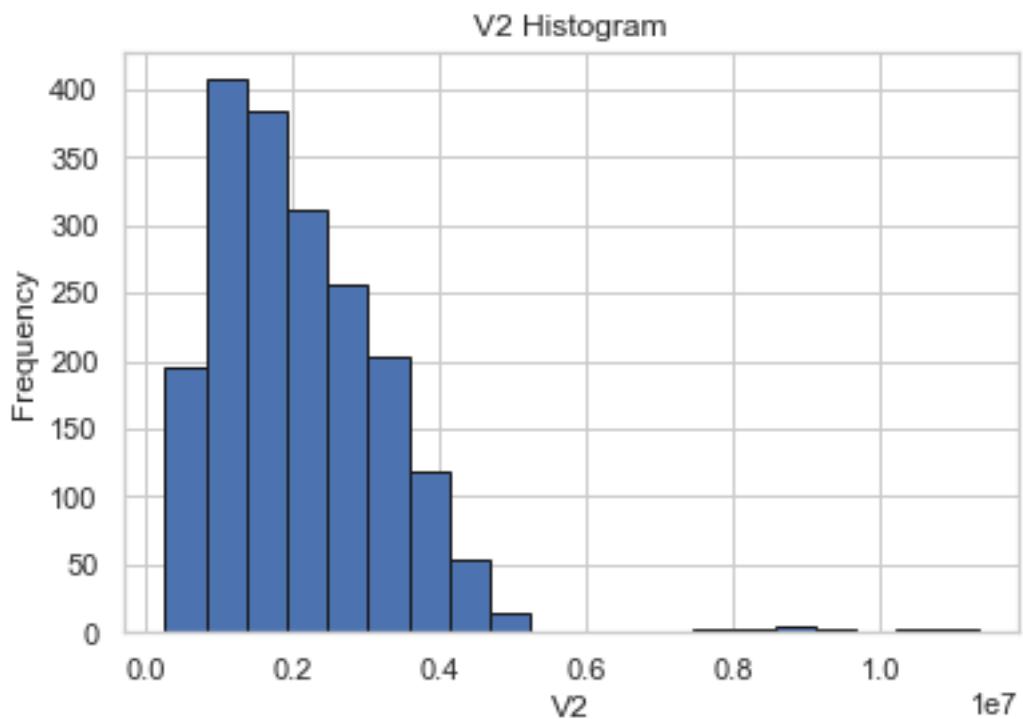


Figure 15: Histogram showing detailed view of data distribution for V2

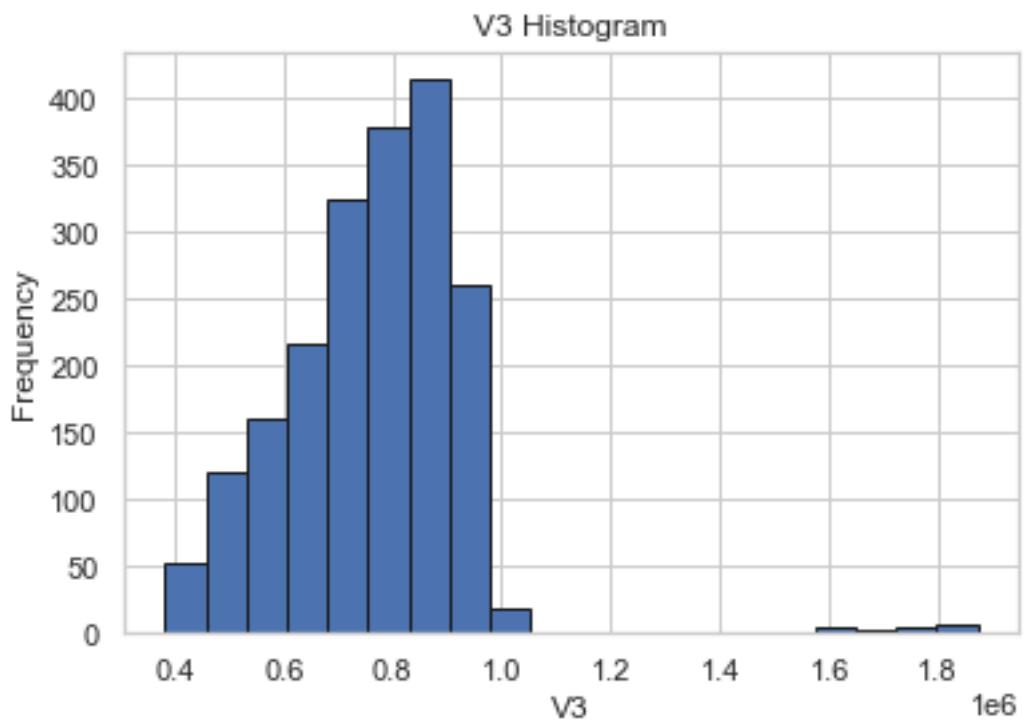


Figure 16: Histogram showing detailed view of data distribution for V3

Define a function with the following signature that uses the seaborn ‘.boxplot()’ method to produce a box-plot chart for a list of features:

```
def plotBox(df, features):
```

Your code here

Now apply that function to each of the Temperature variables in the factory.

Your code here

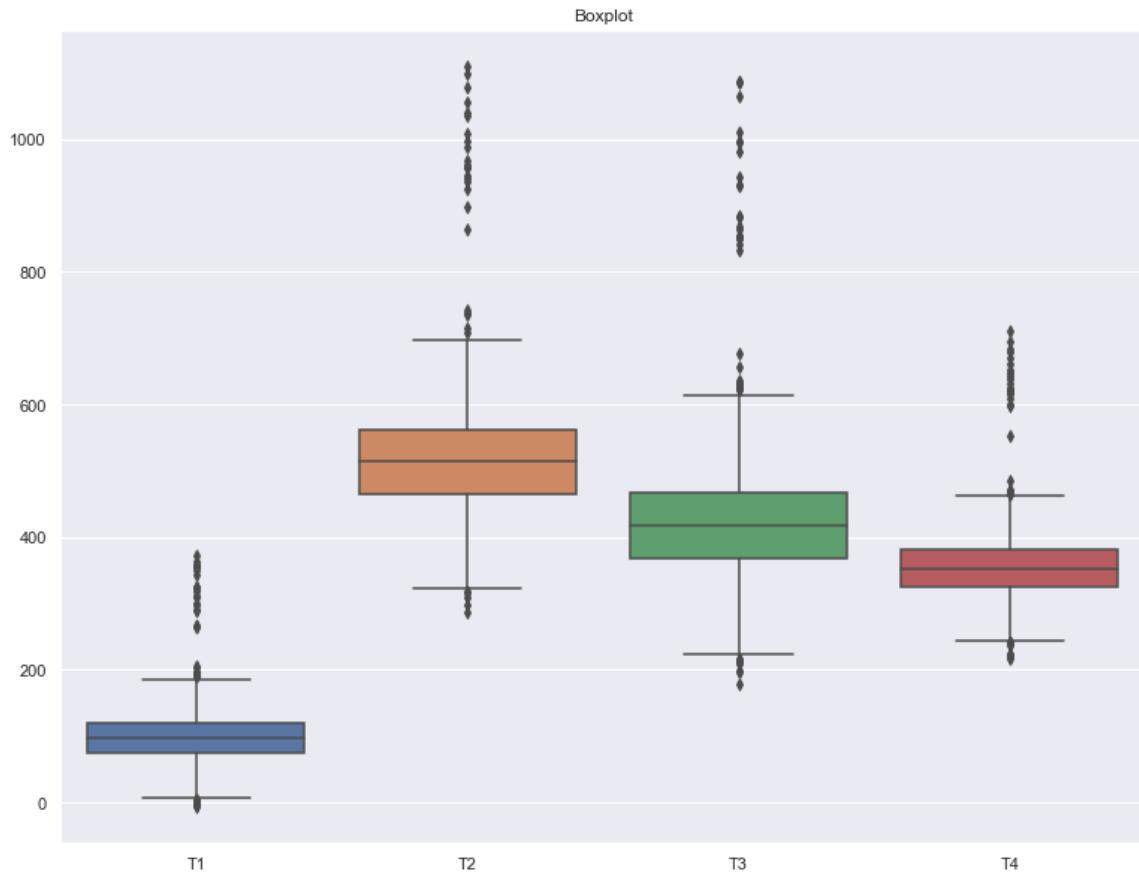


Figure 17: Box-plot showing relative distribution of features and outliers

The box-plot may be understood as follows:

- The central block represents the ‘Inter Quartile Range’ (IQR) i.e. 50% of all values sit within the solid box. If all values were sorted in order, the lowest 25% would sit below the solid box and the largest 25% would fall above it
- The lower whisker is the value: $Q1 - 1.5 * (Q3 - Q1)$

- The upper whisker is the value: $Q3 + 1.5 * (Q3-Q1)$

These definitions are from Tukey - the original inventor of the box-plot. You will sometimes see that same general plot, but with different definitions for the values of the box-ends and whiskers .. so do always check definitions before you draw conclusions about this chart.

 Your code here

Apply the same function to the other sets of variables Vn, An, Pn

 Your code here

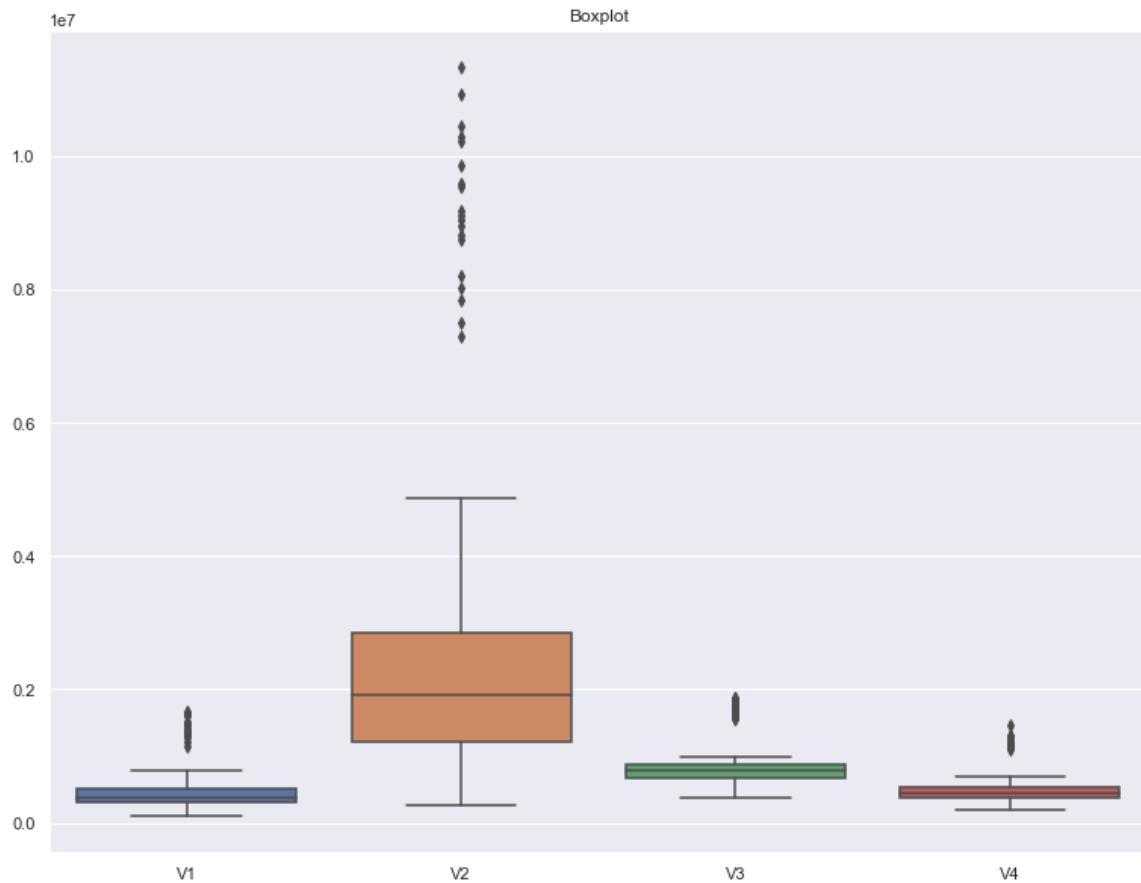
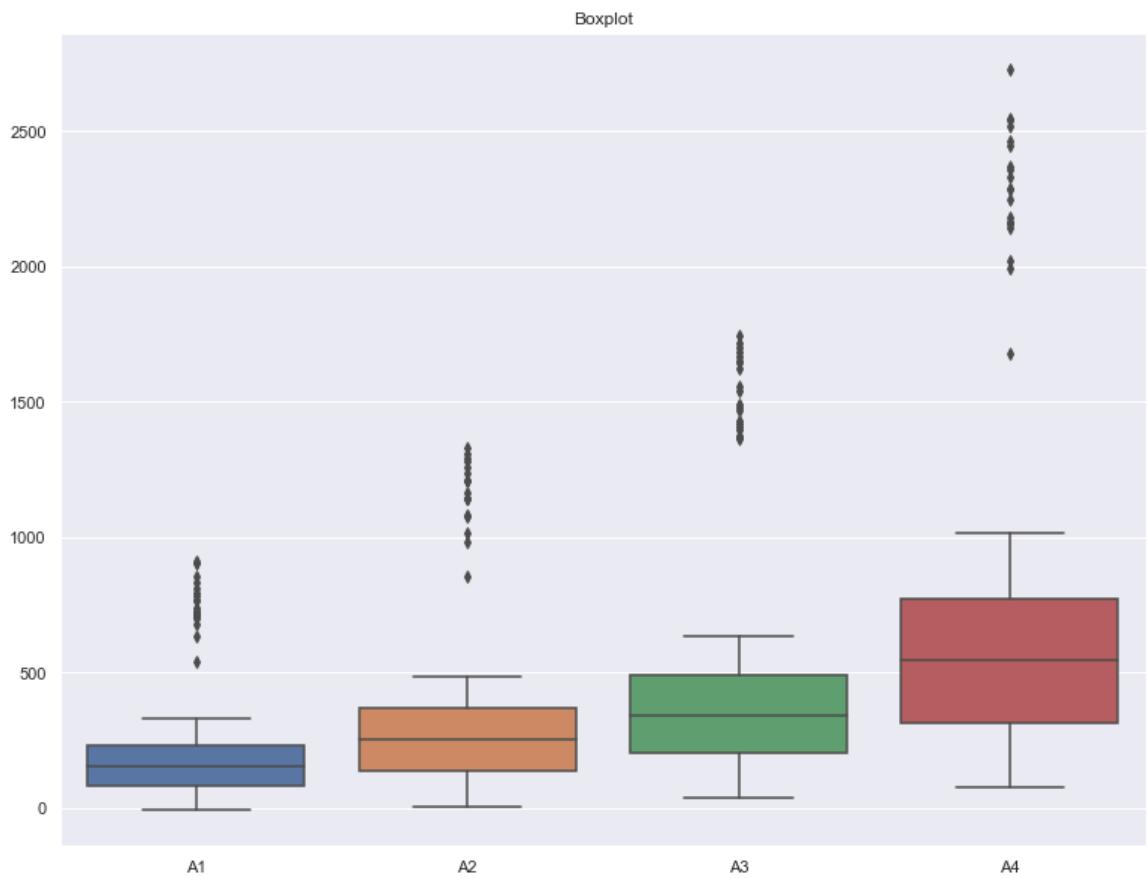
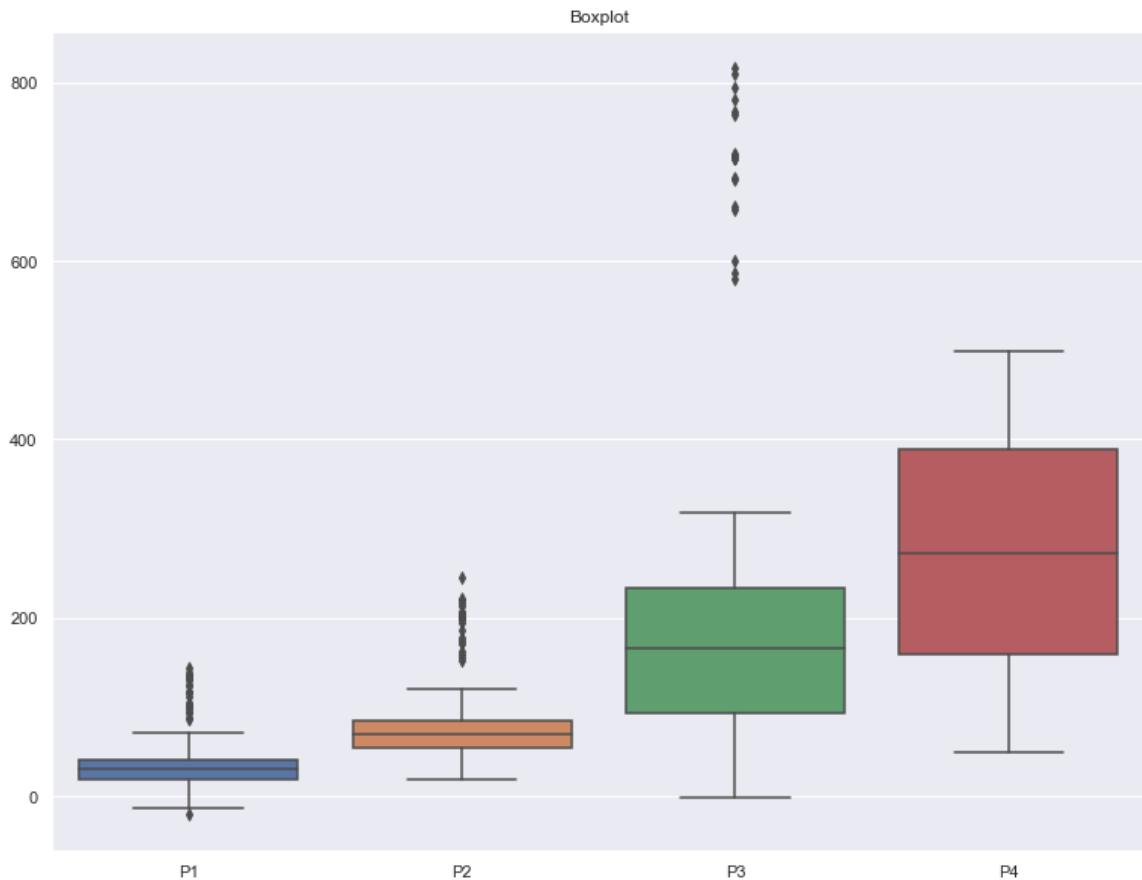


Figure 18: Box-plots showing relative distribution of features and outliers





[Optional] Just to confirm understanding, the key numerical values for these box-plots may be calculated using the numpy ‘percentile’ function:

💡 Tutor provided code

```
def calculate_quartiles_and_iqr(df, feature_name):
    # Extract the values of the specified feature
    feature_values = df[feature_name]

    # Calculate the 25th percentile (Q1)
    q1 = np.percentile(feature_values, 25)

    # Calculate the 75th percentile (Q3)
    q3 = np.percentile(feature_values, 75)

    # Calculate the interquartile range (IQR)
    iqr = q3 - q1

    return q1, q3, iqr
```

```
q1, q3, iqr = calculate_quartiles_and_iqr(df_no_nan, 'P4')
print("25th Percentile (Q1):", q1)
print("75th Percentile (Q3):", q3)
print("Interquartile Range (IQR):", iqr)
```

```
25th Percentile (Q1): 161.82590358002068
75th Percentile (Q3): 387.44044428041923
Interquartile Range (IQR): 225.61454070039855
```

1.17 Violin Plots - Another way of displaying the distribution of data and outliers

It may also be interesting to view this same data as a ‘violin plot’. A violin plot has similarities to both the box-plot and to a histogram. Create a function that uses the seaborn violinplot function to display a series of violin plots for a list of features. Define your function with the following signature:

```
def plotViolin(df, features):
```

💡 Your code here

Apply that function to each of the Temperature variables

💡 Your code here

1.18 Outliers : To Remove or not to remove?

This is an important point to reflect on the root causes of those outliers. We might also run multiple experiments in model building - with and without outlier removal to see what impact it has on any models we build. For now, let’s assume that these really are noise and remove the outlier values.

1.19 Changing outliers to NaN based on the number of standard deviations from the mean

Write a function with the following signature:

```
outliers_to_nan(df, features, n=3):
```

The parameters to this function are:

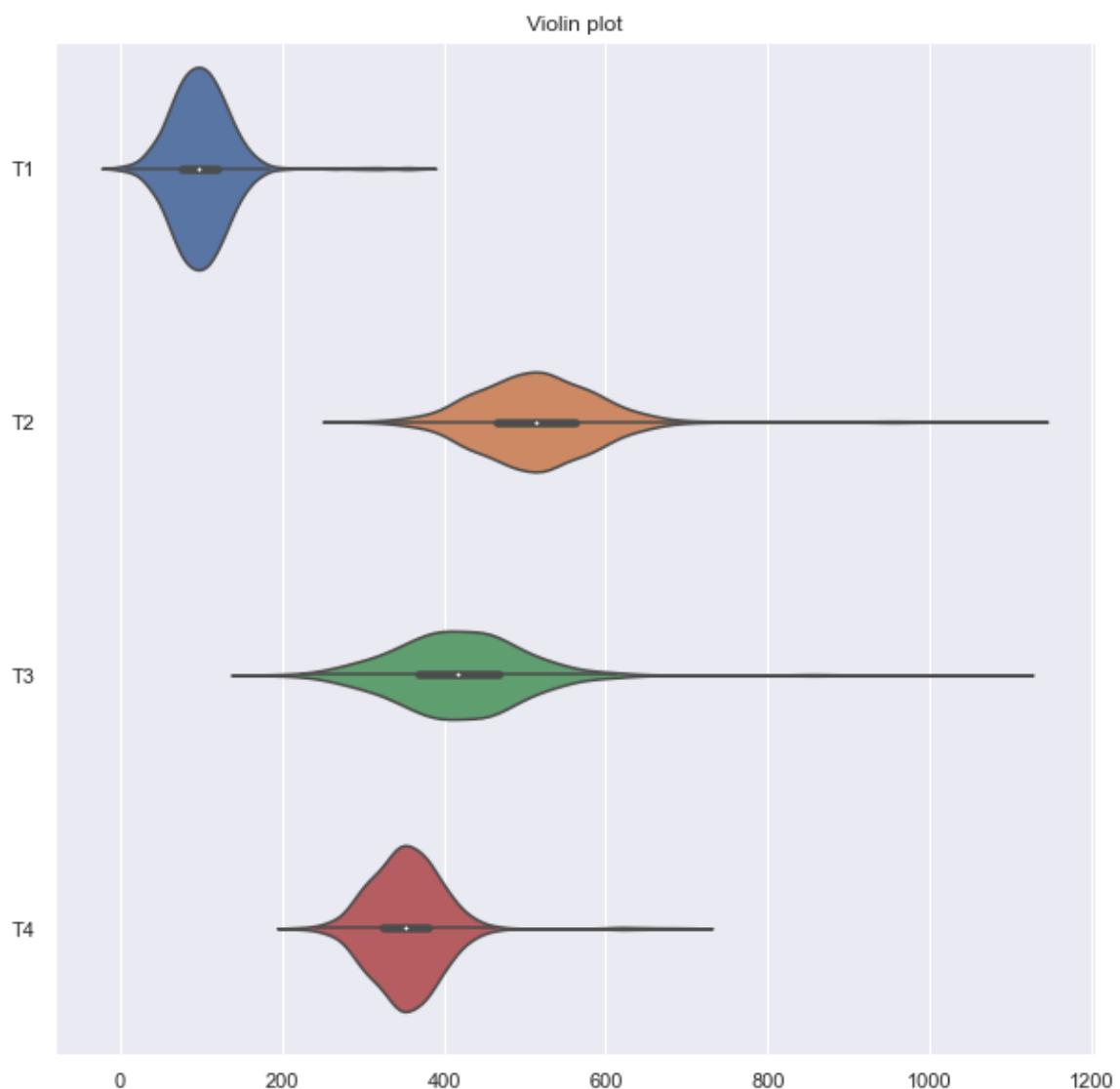


Figure 19: Violin plots: A different view of data distributions and outliers

- df : The dataframe we are cleaning
- features : a list of feature (column) names
- n : The number of standard deviations from the mean we are identifying as being an outlier (with a default of 3)

Hint Somewhat annoyingly, Python passes dataframes to functions ‘by reference’ meaning that any changes you make to a dataframe within a function will effect the dataframe ‘outside’ of the function. This is obviously faster than making a copy of the dataframe for each call (dataframes may be huge!). But in this case, given we have a small amount of data and I want to keep our original data intact, make a copy of the passed dataframe to a new dataframe within the function and manipulate that instead. In other words, do this: df_to_clean = df.copy()

 Your code here

Now call that function with a full list of the ‘input’ variables. We see from above analysis that the output cost variables do not have outliers.

 Your code here

Experiment by replotting histograms of the various input variables to convince yourself that outliers have been removed.

 Your code here

However, we now have even more ‘Nan’ values in our dataframe. Demonstrate this by again using the .info() method of Pandas:

 Your code here

```
df.info =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 30 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   ag              1978 non-null    object 
 1   bg              1980 non-null    object 
 2   T1              1960 non-null    float64
 3   V1              1965 non-null    float64
 4   A1              1964 non-null    float64
```

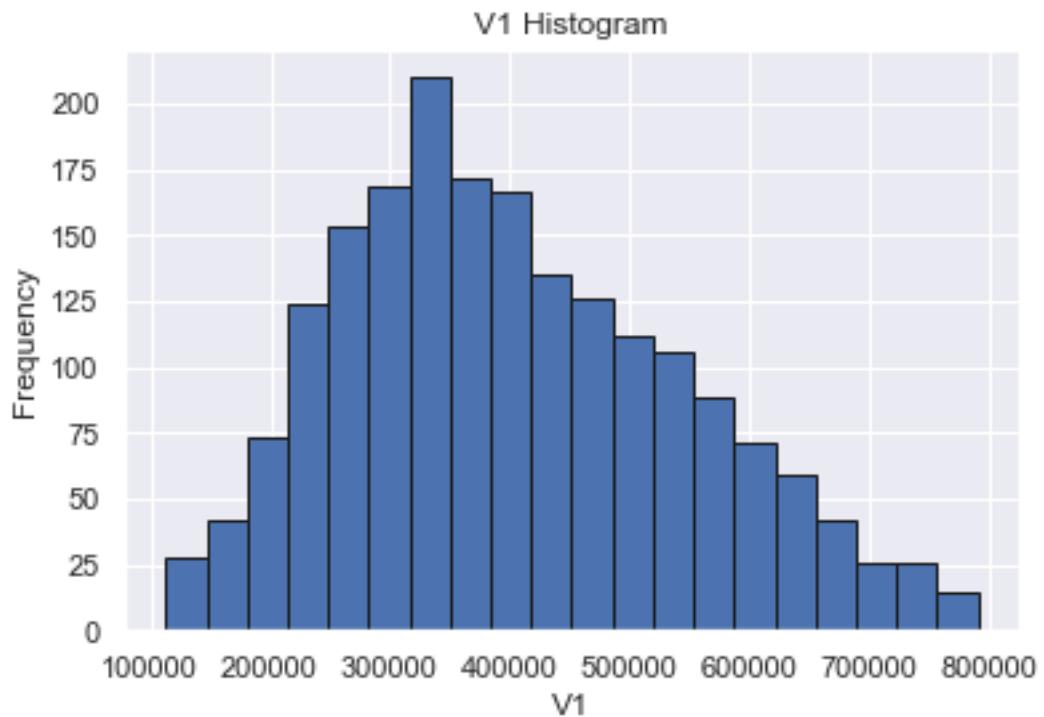


Figure 20: Histograms of data after removal of outliers

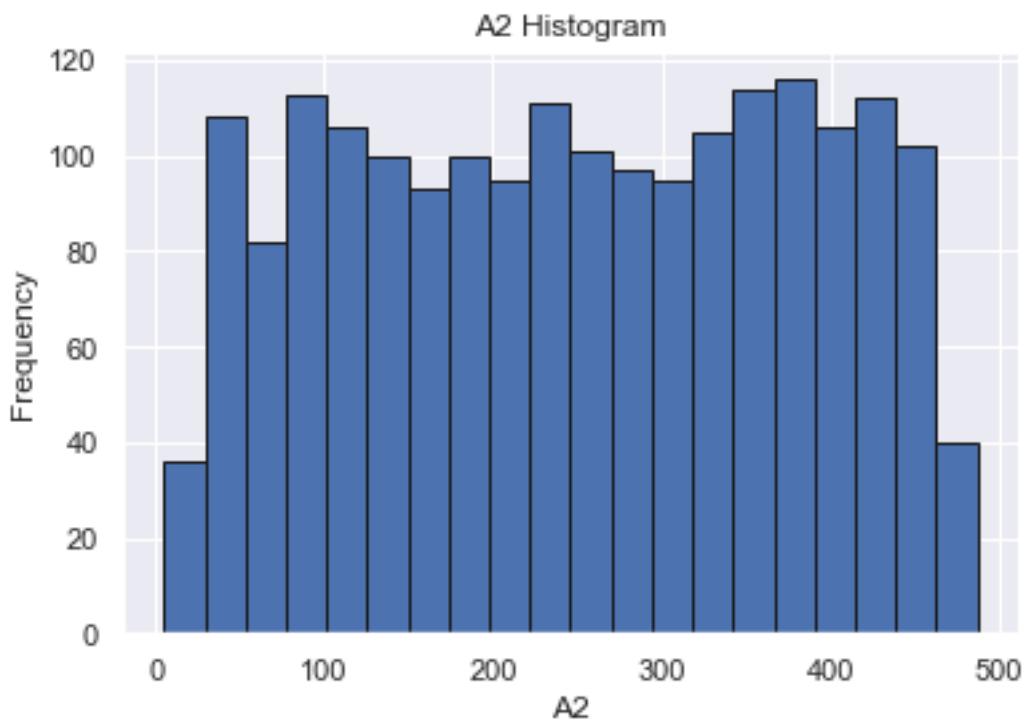


Figure 21: Histograms of data after removal of outliers

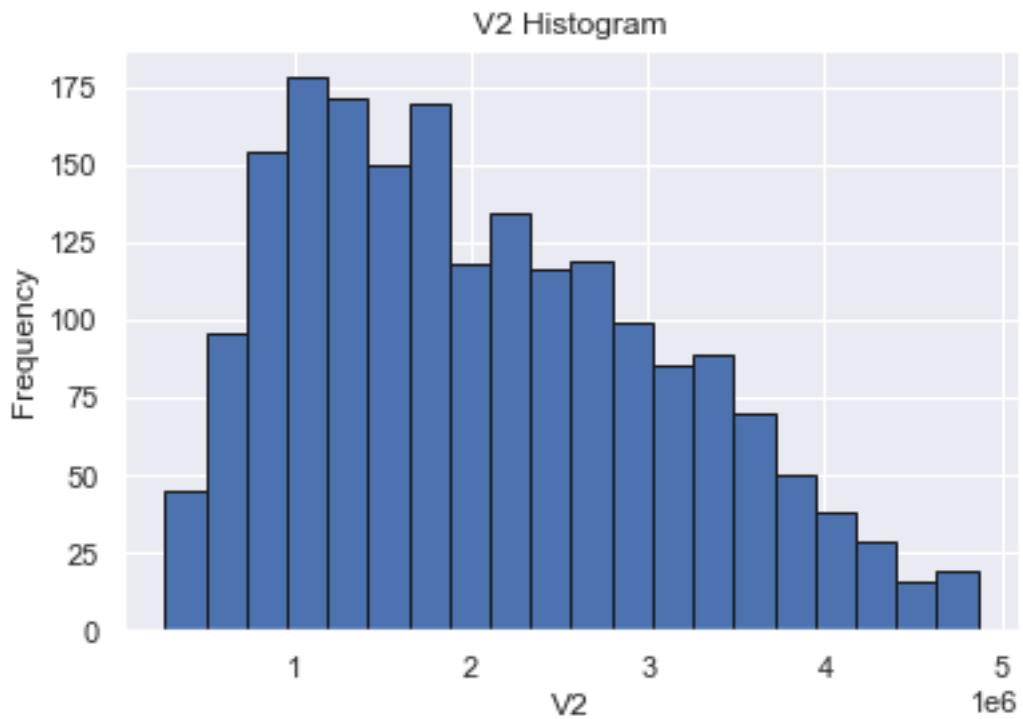


Figure 22: Histograms of data after removal of outliers

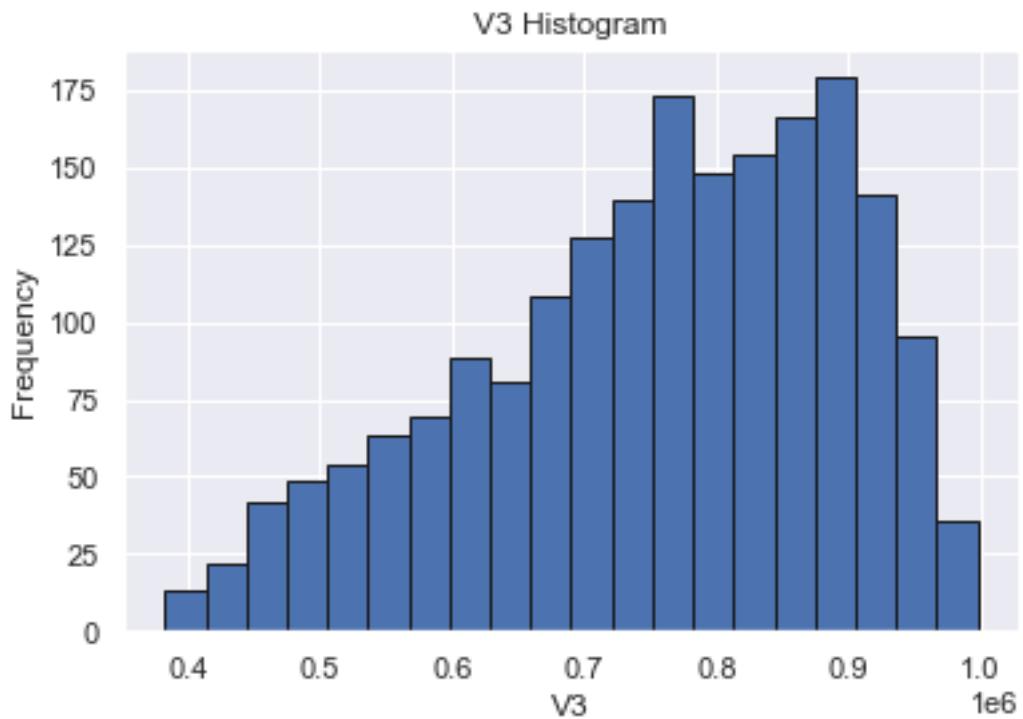


Figure 23: Histograms of data after removal of outliers

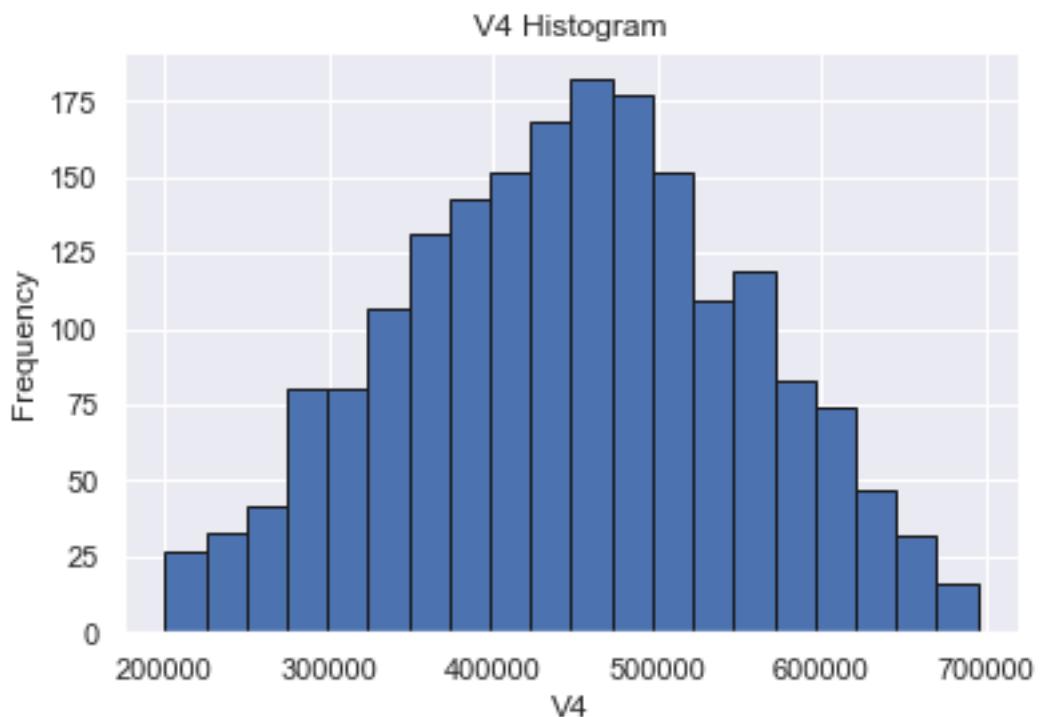


Figure 24: Histograms of data after removal of outliers

5	P1	1967	non-null	float64
6	B1	1969	non-null	float64
7	T2	1963	non-null	float64
8	V2	1965	non-null	float64
9	A2	1950	non-null	float64
10	P2	1955	non-null	float64
11	B2	1968	non-null	float64
12	T3	1960	non-null	float64
13	V3	1967	non-null	float64
14	A3	1957	non-null	float64
15	P3	1963	non-null	float64
16	B3	1960	non-null	float64
17	T4	1959	non-null	float64
18	V4	1972	non-null	float64
19	A4	1957	non-null	float64
20	P4	1977	non-null	float64
21	B4	1964	non-null	float64
22	Contamination_Defect	2000	non-null	int64
23	Crystallisation_Defect	2000	non-null	int64
24	Ion_Diffusion_Defect	2000	non-null	int64
25	Burnishing_Defect	2000	non-null	int64
26	m1_cost	1966	non-null	float64

```

27 m2_cost           1963 non-null   float64
28 m3_cost           1959 non-null   float64
29 m4_cost           1957 non-null   float64
dtypes: float64(24), int64(4), object(2)
memory usage: 468.9+ KB

```

```

df_no_outliers.info() =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 30 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   ag                1978 non-null    object  
 1   bg                1980 non-null    object  
 2   T1                1940 non-null    float64 
 3   V1                1945 non-null    float64 
 4   A1                1944 non-null    float64 
 5   P1                1947 non-null    float64 
 6   B1                1949 non-null    float64 
 7   T2                1943 non-null    float64 
 8   V2                1946 non-null    float64 
 9   A2                1932 non-null    float64 
 10  P2                1935 non-null    float64 
 11  B2                1949 non-null    float64 
 12  T3                1940 non-null    float64 
 13  V3                1947 non-null    float64 
 14  A3                1937 non-null    float64 
 15  P3                1945 non-null    float64 
 16  B3                1940 non-null    float64 
 17  T4                1939 non-null    float64 
 18  V4                1952 non-null    float64 
 19  A4                1937 non-null    float64 
 20  P4                1977 non-null    float64 
 21  B4                1956 non-null    float64 
 22  Contamination_Defect  2000 non-null  int64  
 23  Crystallisation_Defect 2000 non-null  int64  
 24  Ion_Diffusion_Defect  2000 non-null  int64  
 25  Burnishing_Defect    2000 non-null  int64  
 26  m1_cost             1966 non-null    float64 
 27  m2_cost             1963 non-null    float64 
 28  m3_cost             1959 non-null    float64 
 29  m4_cost             1957 non-null    float64 
dtypes: float64(24), int64(4), object(2)
memory usage: 468.9+ KB

```

1.20 Imputing missing values

1.20.1 A simple method based on either the mean or median

We now want to replace those NaN values with something that will not impact our models too greatly. A common choice is to replace NaN values with either the mean or the median of the feature values. Mean tends to be used for Gaussian data, Median for other data.

Write a function that replaces all NaN values in a feature with the either the ‘mean’ or ‘median’ of value of that feature.

The signature of the function should be as follows:

```
def impute_nan_values(df, feature_name, imputation_type='mean'):
```

 Tutor provided code

```
def impute_nan_values(df, feature_name, imputation_type='mean'):

    # Make a copy of the original DataFrame to avoid modifying it in place
    df_imputed = df.copy()

    # Calculate the imputation value based on the specified type
    if imputation_type == 'mean':
        imputation_value = df_imputed[feature_name].mean()
    elif imputation_type == 'median':
        imputation_value = df_imputed[feature_name].median()
    else:
        raise ValueError("Invalid imputation_type. Use 'mean' or 'median'.")

    # Impute NaN values in the specified feature
    df_imputed[feature_name].fillna(imputation_value, inplace=True)

    return df_imputed
```

Now we will impute values to replace the Nan values within the Temperature Features using mean imputation:

- Create a new dataframe called ‘df_imputed’
- Call the function ‘impute_nan_values’ (defined above) returning the results into this new dataframe
- In the first instance, impute missing values in the ‘T1’ feature
- Impute based on mean values in that feature

 Your code here

Repeat this process for each of the Temperature variables.

Hint : Remember that in each subsequent call you will need to pass the ‘df_imputed’ dataframe to the ‘impute_nan_values’ function since this is the one that is accumulating the results of your repeated actions.

💡 Your code here

Then use missingno to check that we have, in fact, removed those missing values:

💡 Your code here

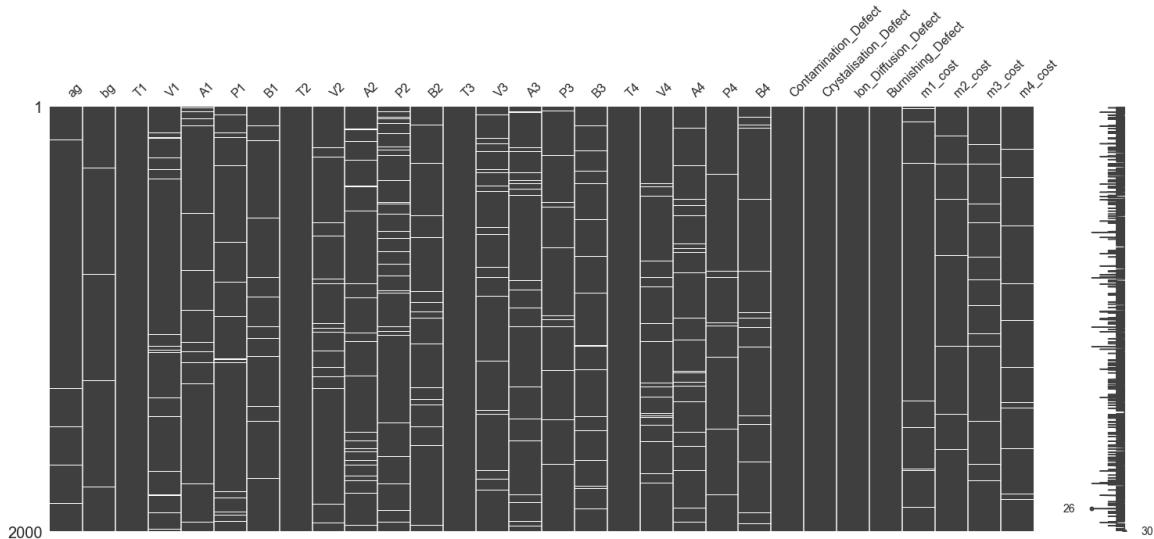


Figure 25: Results of imputing missing values of Tn

The cost variables also appear to be normally (Gaussian) distributed, so impute their missing values based on the mean.

💡 Your code here

💡 Your code here

And check again using the missingno matrix function:

💡 Your code here

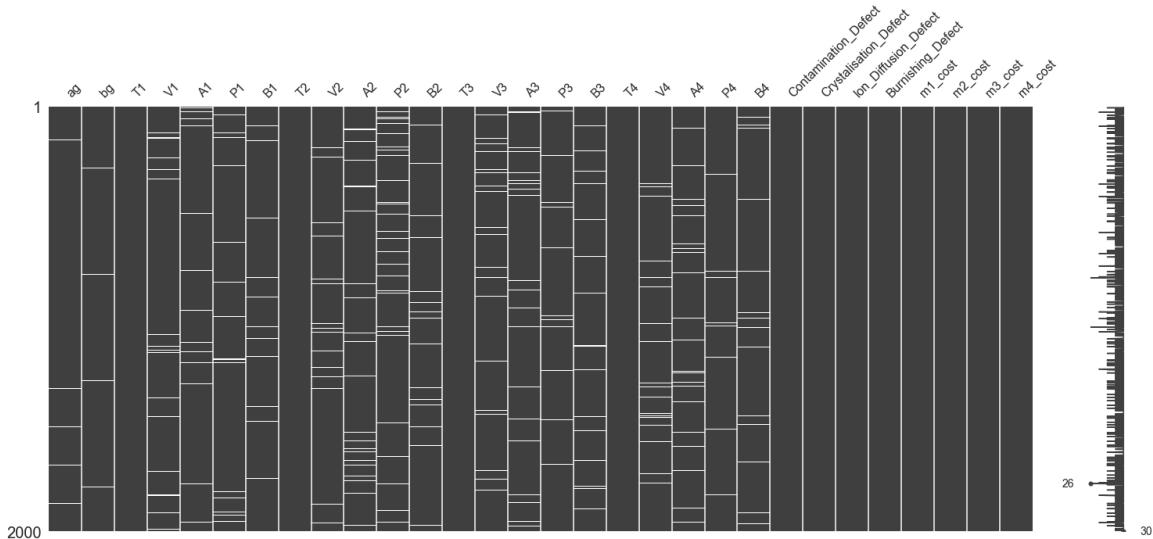


Figure 26: Results of imputing missing values for cost variables

Now impute other missing values based on the median of each feature

Your code here

And display the results using missingno

Your code here

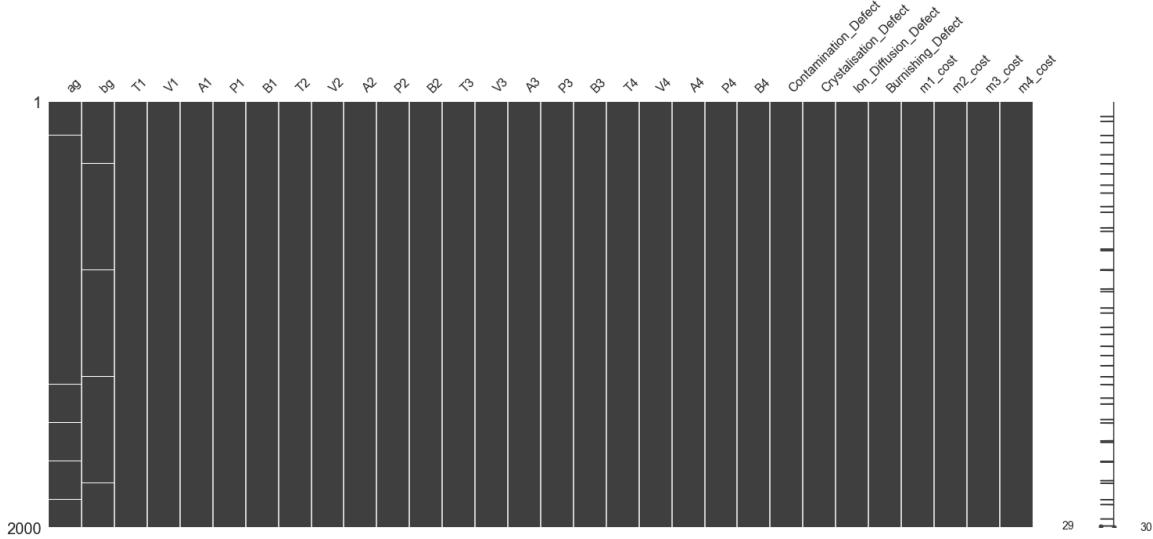


Figure 27: Final results of imputing missing values for all numerical variables

1.20.2 A more sophisticated approach to Imputation

This section introduces a more sophisticated approach to important called ‘Multivariate feature imputation’. Rather than simply using the ‘mean’, ‘median’ or ‘mode’ of a column the algorithm uses row context to infer a most likely value.

Details of the algorithm used in this section can be found at:

- Multivariate feature imputation:
 - <https://scikit-learn.org/stable/modules/impute.html#multivariate-feature-imputation>
- sklearn.impute.IterativeImputer:
 - <https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html>

To experiment with this imputer let’s re-use an earlier version of our dataframe that contains missing values: ‘df_no_outliers’

 Your code here

```
df_no_outliers.info() =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 30 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   ag               1978 non-null    object 
 1   bg               1980 non-null    object 
 2   T1               1940 non-null    float64
 3   V1               1945 non-null    float64
 4   A1               1944 non-null    float64
 5   P1               1947 non-null    float64
 6   B1               1949 non-null    float64
 7   T2               1943 non-null    float64
 8   V2               1946 non-null    float64
 9   A2               1932 non-null    float64
 10  P2               1935 non-null    float64
 11  B2               1949 non-null    float64
 12  T3               1940 non-null    float64
 13  V3               1947 non-null    float64
 14  A3               1937 non-null    float64
 15  P3               1945 non-null    float64
 16  B3               1940 non-null    float64
 17  T4               1939 non-null    float64
```

```

18 V4                      1952 non-null   float64
19 A4                      1937 non-null   float64
20 P4                      1977 non-null   float64
21 B4                      1956 non-null   float64
22 Contamination_Defect    2000 non-null   int64
23 Crystalisation_Defect   2000 non-null   int64
24 Ion_Diffusion_Defect    2000 non-null   int64
25 Burnishing_Defect       2000 non-null   int64
26 m1_cost                  1966 non-null   float64
27 m2_cost                  1963 non-null   float64
28 m3_cost                  1959 non-null   float64
29 m4_cost                  1957 non-null   float64
dtypes: float64(24), int64(4), object(2)
memory usage: 468.9+ KB

```

Create an object called ‘imp’ from the sklean ‘IterativeImputer’ class. Set the ‘max_iter’ parameter to 50, and the ‘random_state’ parameter to 0

 Your code here

```
IterativeImputer(max_iter=50, random_state=0)
```

We will be using the ‘fit_transform’ method. This will accept a pandas dataframe as a parameter. In this case, apply it to the ‘df_no_outliers’ dataframe created above.

Note however, that the algorithm will only work on numerical values. Therefore, you will have to pass a ‘slice’ of your dataframe that runs from feature ‘T1’ to feature ‘m4_cost’.

Store the result in a variable called ‘imputed’

```
array([[9.39888613e+01, 4.45746580e+05, 2.99436975e+02, ...,
       2.06462176e+03, 1.15656113e+03, 8.40469242e+02],
      [5.71618762e+01, 3.89648469e+05, 2.75276163e+02, ...,
       6.22104318e+03, 8.69714043e+02, 5.23187572e+02],
      [8.12557094e+01, 3.96609840e+05, 2.35951773e+02, ...,
       4.00828438e+03, 8.22968657e+02, 6.14949484e+02],
      ...,
      [1.10841477e+02, 5.90151683e+05, 2.55159715e+02, ...,
       2.03642517e+03, 9.12390999e+02, 4.73057867e+02],
      [4.86128177e+01, 4.10247004e+05, 2.30213514e+02, ...,
       4.20286098e+03, 1.20868143e+03, 8.21615172e+02],
      [1.23779047e+02, 1.80230435e+05, 1.64804261e+02, ...,
       1.51756993e+03, 1.07950148e+03, 8.62817029e+02]])
```

You will now need to turn the returned numpy array back into a dataframe using ‘pd.DataFrame’.

Hint : The second parameter will need to be the list of column names for the dataframe. These can simply be a copy of those from the dataframe slide you sent to the imputer.m

💡 Your code here

We will also need to copy back in the ‘ag’ and ‘bg’ features into this dataframe as we lost those in the process.

💡 Your code here

Finally, you can display the ‘msno.matrix()’ for ‘df_imputed’ to get a view of the dataframe with missing data having been removed.

💡 Your code here

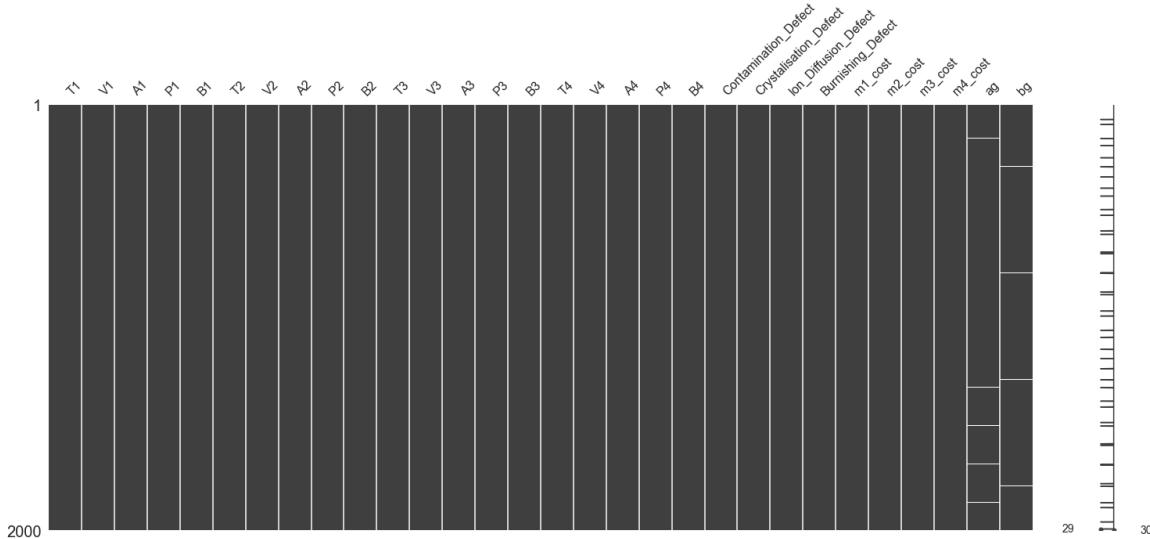


Figure 28: msno matrix of datafram after imputing with IterativeImputer

And at this point we should removed all of the NaN values from the numerical portion of the dataframe.

1.21 Imputing categorical variables

We still have the problem that there are missing values in the ‘ag’ and ‘bg’ features. Given the small number of missing values it would not be too costly to simply delete the whole rows

containing missing data. However, it is instructive to think about how we might impute categorical values.

One fairly common strategy is to replace NaN values with the most common category from the feature. This can be easily achieved using ‘sklearn.impute.SimpleImputer’ (<https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html>)

Add a cell that uses sklearn’s ‘SimpleImputer’ with the ‘strategy’ parameter set to ‘most_frequent’ to impute missing values for the ‘ag’ and ‘bg’ features:

- Create an object called ‘imputer’ from the sklearn class ‘SimpleImputer’ with a ‘strategy’ parameter set to ‘most_frequent’
- Call the ‘fit’ method on ‘imputer’ passing the ‘df_imputed’ features ‘ag’ and ‘bg’
- Set ‘df_imputed[['ag', 'bg']] = imputer.transform(df_imputed[['ag', 'bg']])

💡 Your code here

Then again plot the missing data using the missingno matrix method.

💡 Your code here

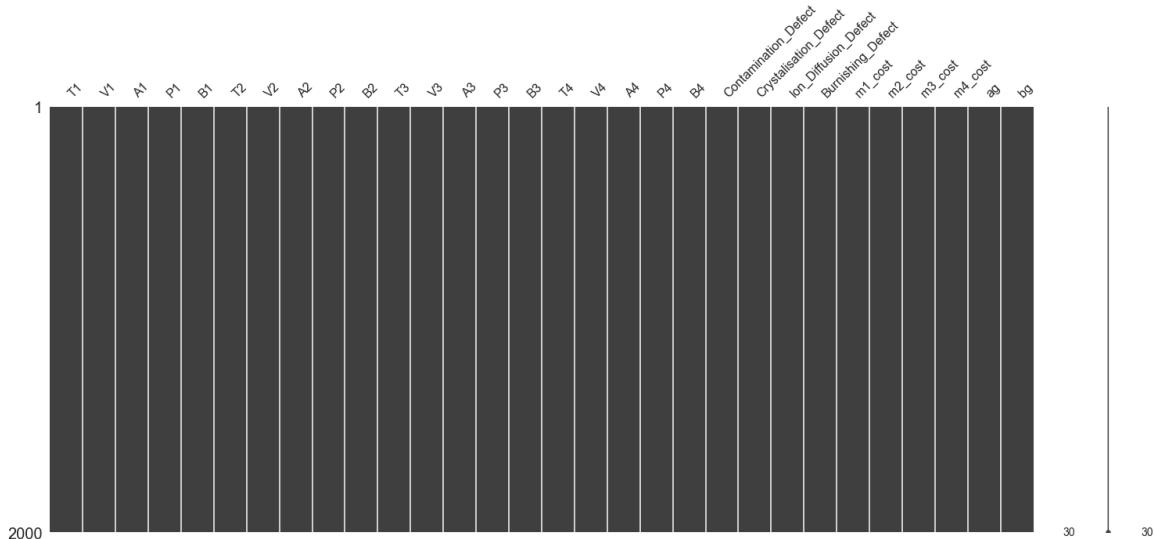


Figure 29: Final results of imputing all missing values

At last we have a dataframe with no missing values!

1.22 Transforming the data to a standard scale

It is generally useful in machine learning to transform data so that each feature has broadly the same range. The rationale for this is many fold and includes:

- **Equalizing Scale:** Standardization brings all features to the same scale. Features measured in different units or with different scales can have a disproportionate impact on the model. Standardization helps to ensure that each feature contributes equally to the learning process.
- **Facilitating Regularization:** Regularization techniques, such as L1 (Lasso) and L2 (Ridge) regularization, involve penalizing large coefficients. Standardization helps to ensure that all features are penalized equally, making regularization more fair and effective.
- **Improving Convergence:** Many machine learning algorithms, especially those based on gradient descent, converge faster when features are standardized. The uniform scale of features helps optimization algorithms find the optimal solution more efficiently.
- **Avoiding Dominance of Large Values:** Features with larger magnitudes might dominate those with smaller magnitudes. Standardization prevents this dominance, ensuring that the influence of each feature is determined by its relative importance rather than its scale.
- **Enhancing Interpretability:** For models like linear regression, the coefficients represent the change in the dependent variable for a one-unit change in the corresponding independent variable. Standardizing features makes it easier to interpret the coefficients in a meaningful way.
- **Improving Model Performance:** Some algorithms, like k-nearest neighbors (KNN) and support vector machines (SVMs), are sensitive to the scale of features. Standardization can lead to better performance for these algorithms.
- **Assisting Distance-Based Algorithms:** Algorithms that rely on distances between data points, such as k-means clustering or hierarchical clustering, can be influenced by the scale of features. Standardizing features ensures that distances are computed appropriately.
- **Preventing Numerical Instability:** Standardization can prevent numerical instability issues that might arise when working with algorithms that involve matrix inversions or singular value decompositions.

If it has not become obvious already .. our data actually does have a rather significant difference in scale between features. This can easily be observed using a box-plot of key features as above.

Create a list of all numerical features, Tn, Vn, An and Bn

 Your code here

```
features_to_be_standardized =  
[  
T1,  
V1,  
A1,  
B1,  
T2,  
V2,  
A2,  
B2,  
T3,  
V3,  
A3,  
B3,  
T4,  
V4,  
A4,  
B4,  
]
```

Then re-use the ‘plotBox’ function defined earlier to display a box plot of all of the numerical input variables

 Your code here

Clearly, the features in the data-set have a significantly different scale. This can be fixed by ‘standardizing’ the data.

Write a function that applies standardScaler to a list of features in a dataframe with the signature:

```
feature_standardize(df, features_to_standardize)
```

The following reference page may be useful:

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

 Your code here

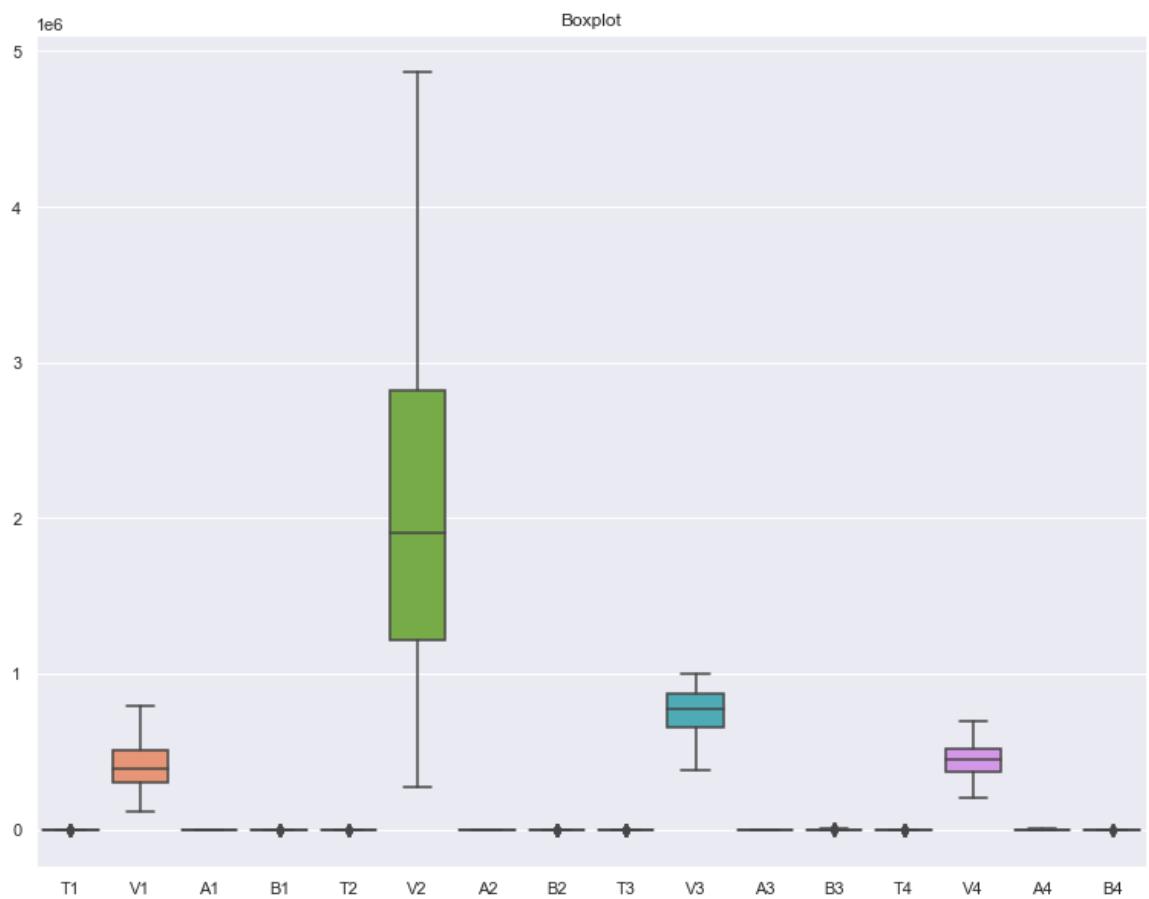


Figure 30: Box-plot of all variables showing significant scale differences

Apply that function to dataframe ‘df_imputed’ using the list of numerical features defined previously (‘features_to_be_standardized’):

💡 Your code here

Then re-plot the box-plot to see the impact

💡 Your code here

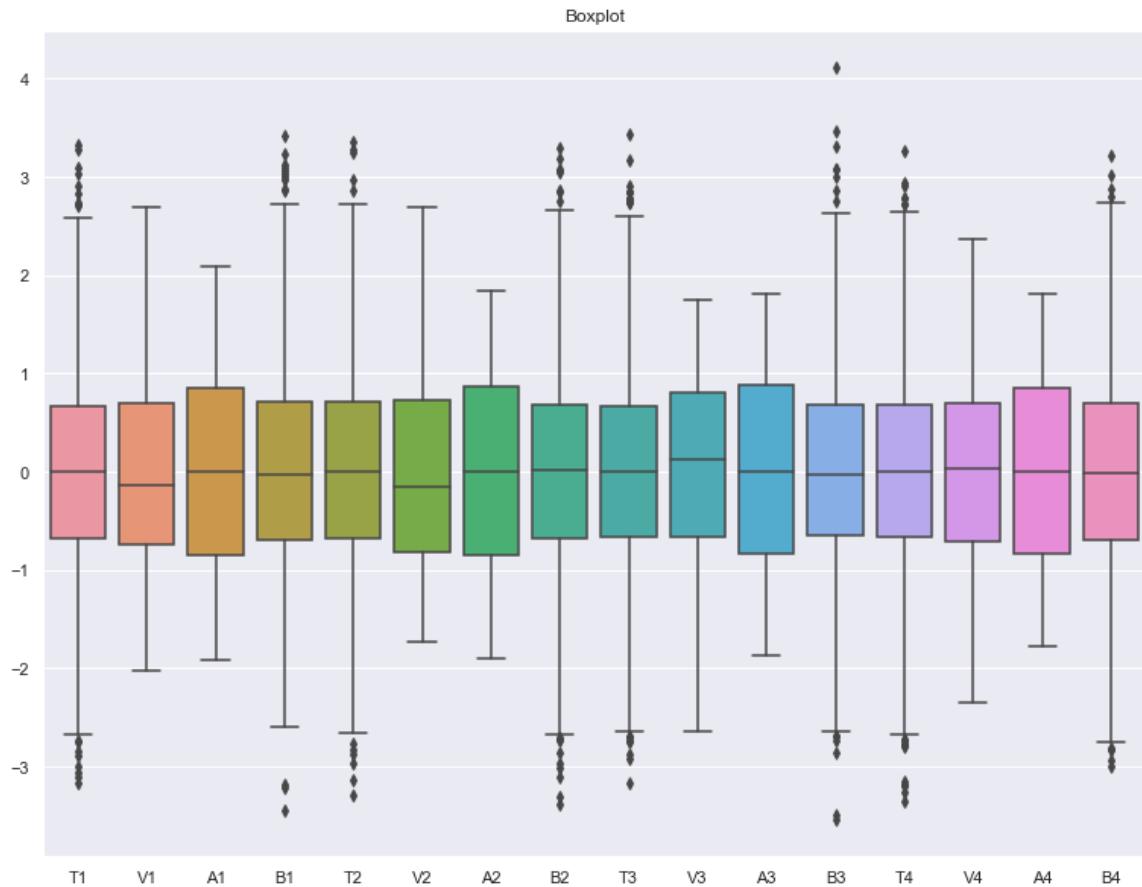


Figure 31: Box-plot of all variables after standardization

Note that this diagram also provides another clue that some of these features are not normally (Gaussian) distributed. If they were Gaussian, then the central, horizontal line in each bar would be at zero. However, in some cases the bar is above or below zero. This corresponds to the cases where the median value is not at zero. In other words, these are skewed distributions (Triangle or log-normal for example).

1.22.1 We need to record the values used for the transformation

An important thing to consider here is that any model we produce will be based on the transformed data. That would mean, the model could not be directly applied to any new data that was not similarly transformed. For that reason, it is normally important to retain a record of the actual scaling that was applied to each attribute to achieve the above standardisation.

Re-write your standardisation function so that as well as returning an transformed dataframe, it also returns a list of the mean and variation transform values for each feature. These can be obtained from StandardScaler using the ‘scale_’ and ‘mean_’ attributes.

💡 Your code here

Call this new version of the function and print out the scaling parameters

💡 Your code here

List of scaling parameters for each feature:

```
[3.24370227e+01 1.43685666e+05 8.29906815e+01 2.67744856e+01  
6.84009019e+01 1.03774933e+06 1.28801810e+02 4.19086650e+01  
7.55453637e+01 1.40022577e+05 1.61671155e+02 8.87805845e+01  
4.06970216e+01 1.05014443e+05 2.61014756e+02 1.57331526e+01]
```

List of means for each feature

```
[9.70304514e+01 4.04359429e+05 1.54484183e+02 1.19812150e+02  
5.11913346e+02 2.06813519e+06 2.49944844e+02 3.01310587e+02  
4.16376284e+02 7.53372005e+05 3.39766537e+02 8.16474195e+02  
3.518333919e+02 4.46270971e+05 5.40301051e+02 3.39382438e+02]
```

At this point, however, we have lost some of the features from our original dataframe. Write a few lines of code to copy those lost features back into our standardized dataframe.

(**Note** that in practice many of the above operations would normally have been executed ‘in-place’ to reduce memory usage and the computational overhead of copying large sets of data. In this case I have kept them separate in order to make it very clear what is happening at each stage)

💡 Your code here

	T1	V1	A1	B1	T2	V2	A2	B2
0	-0.09	0.29	1.75	-2.17	2.17	-0.59	1.70	-1.35

	T1	V1	A1	B1	T2	V2	A2	B2
1	-1.23	-0.10	1.46	0.48	-0.14	2.38	1.44	-2.14
2	-0.49	-0.05	0.98	0.41	-0.15	0.80	-1.06	-1.34
3	0.88	-1.66	0.81	-0.11	-1.08	0.58	-0.78	0.16
4	1.26	1.55	0.57	0.83	0.96	-1.13	-0.06	0.61
...
1995	-0.67	-0.77	1.71	0.05	-1.29	-0.35	-0.27	0.10
1996	0.93	1.08	-0.70	0.48	2.29	1.15	-1.02	2.62
1997	0.43	1.29	1.21	0.51	-0.67	-0.61	-1.25	1.15
1998	-1.49	0.04	0.91	-0.18	0.36	0.94	-0.38	0.25
1999	0.82	-1.56	0.12	-0.28	0.39	-0.98	1.65	-0.01

	T3	V3	A3	B3	T4	V4	A4	B4
0	-0.61	0.24	-0.64	0.97	0.54	1.00	-0.58	0.65
1	-0.24	-1.28	-0.64	-0.36	-0.06	-1.24	1.06	0.08
2	3.17	-1.53	-1.48	0.05	-0.18	-0.59	-1.22	1.01
3	0.45	0.38	1.31	-0.34	0.27	0.15	-0.11	0.67
4	0.43	0.81	0.17	0.28	-0.51	-1.04	0.40	-0.41
...
1995	-0.37	-2.56	0.42	-2.03	1.54	-0.63	1.51	0.35
1996	1.32	-1.44	0.63	-0.27	-0.46	-1.20	-0.18	0.18
1997	0.38	-1.06	0.20	-0.05	0.99	-1.59	-1.74	1.63
1998	1.11	0.50	0.90	-1.24	0.78	0.87	-0.49	-1.73
1999	-0.84	-0.17	-0.79	0.69	-1.57	1.16	0.25	0.21

	Contamination_Defect	Crystallisation_Defect	Ion_Diffusion_Defect	Burnishing_Defect
0	0.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	0.0	1.0	0.0	0.0
3	1.0	1.0	0.0	0.0
4	0.0	0.0	0.0	0.0
...
1995	0.0	0.0	1.0	0.0
1996	0.0	1.0	0.0	0.0
1997	0.0	0.0	0.0	0.0
1998	0.0	1.0	0.0	0.0
1999	0.0	0.0	0.0	0.0

	ag	bg	m1_cost	m2_cost	m3_cost	m4_cost
0	Low	Argon	1139.53	2064.62	1156.56	840.47

	ag	bg	m1_cost	m2_cost	m3_cost	m4_cost
1	Low	Argon	1008.11	6221.04	869.71	523.19
2	Medium	Argon	1024.36	4008.28	822.97	614.95
3	Medium	Neon	480.40	3698.99	1184.28	719.92
4	Medium	Argon	1565.70	1309.50	1266.84	551.15
...
1995	Medium	Argon	782.01	2406.01	629.39	609.98
1996	Low	Argon	1408.13	4500.85	841.29	528.37
1997	High	Argon	1479.36	2036.43	912.39	473.06
1998	High	CO2	1056.18	4202.86	1208.68	821.62
1999	Low	Argon	515.76	1517.57	1079.50	862.82

1.23 Translating Categorical Data into a Form that can be used in Models

We now turn our attention back to the two categorical variables ‘ag’ and ‘bg’. We will want to use them for modelling but many models require numerical inputs - they are essentially statistical / mathematical operations. For this reason we will need to transform those variables into a form that can be consumed by numerical algorithms.

We note some difference between each of the variables:

- ‘ag’ contains the values ‘High’, ‘Medium’ and ‘Low’. Although they are categorical we can also infer some ordering in the values
- ‘bg’ contains the values ‘Argon’, ‘Neon’ and ‘CO2’. These do not carry any suggestion of ordering (or if they do, that is beyond my knowledge in Chemistry)

Features such as ‘ag’ are referred to as ‘ordinal’ : Their values indicate an ordering.

We need to transform both of these variables numerical values that can be used for modelling. We are going to use a different strategy in each case.

1.23.1 Transforming Ordinal data

Considering first ‘ag’ ..

First create a dictionary named ‘ag_mapping’ that defines the relationship between an ordinal term and a numerical value.

Hint : the answer will look something like this ..

```
my_map = {'small' : 1, 'large' : 2, 'XL' : 3}
```

Your code here

... write a function with the following signature:

```
def transform_ordinal_to_numeric(df, feature, mapping_dict):
```

- ‘df’ is the source dataframe
- ‘feature’ is the ordinal feature to be transformed
- ‘mapping_dict’ is the dictionary that shows the mapping between strings and numerical values (as shown above)

The function should first copy the provided dataframe (remembering that dataframes are passed to functions by reference) and return the transformed dataframe.

 Tutor provided code

```
def transform_ordinal_to_numeric(df, feature, mapping_dict):  
  
    df_transformed = df.copy()  
    df_transformed[feature] = df_transformed[feature].map(mapping_dict)  
  
    return df_transformed
```

Then apply this function to the ‘ag’ feature of your dataframe and display the result

 Your code here

ag	
0	1
1	1
2	2
3	2
4	2
...	...
1995	2
1996	1
1997	3
1998	3
1999	1

1.23.2 Transforming Categorical Data to ‘1-hot-encoded’ data

We have to deal with the ‘bg’ feature somewhat differently. We can’t translate those into numbers, because the numbers imply an ordering - and these values are not ordered. In this case, we use another trick called ‘on-hot-encoding’ Or at least that’s what most people call it. The people who designed the pandas library however decided to call their function to do this ‘get_dummies’ referring to ‘dummy variables’

In a single line of code, use the pandas ‘get_dummies’ method to transform the ‘bg’ feature into a series of on-hot-encoded features.

💡 Your code here

Display those new features using the code:

```
df_final.loc[:, 'bg_Argon': 'bg_Neon']
```

💡 Your code here

	bg_Argon	bg_C02	bg_Neon
0	1	0	0
1	1	0	0
2	1	0	0
3	0	0	1
4	1	0	0
...
1995	1	0	0
1996	1	0	0
1997	1	0	0
1998	0	1	0
1999	1	0	0

1.24 Other data visualizations that may provide insight during early exploratory data analysis

A ‘kdeplot’ (kernel density estimation) displays a smoothed representation of the distribution of a dataset. It is somewhat like a histogram. However, histograms suffer from an issue in that the choice of the bin size can drastically alter the appearance of the chart.

kde plots provide a more ‘realistic’ display of the shape of the distribution. The seaborn library function also provides a parameter that allows the user to select the level of smoothing.

💡 Tutor provided code

```
for column in df_final.columns:  
    plt.figure(figsize=(12, 2))  
    sns.kdeplot(df_final[column], label=column)  
    plt.title(f'KDE Plot for {column}')  
    plt.legend()
```

```
plt.show()
```

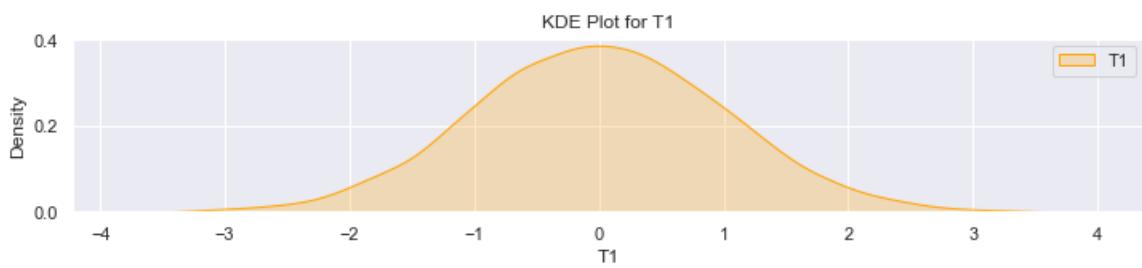
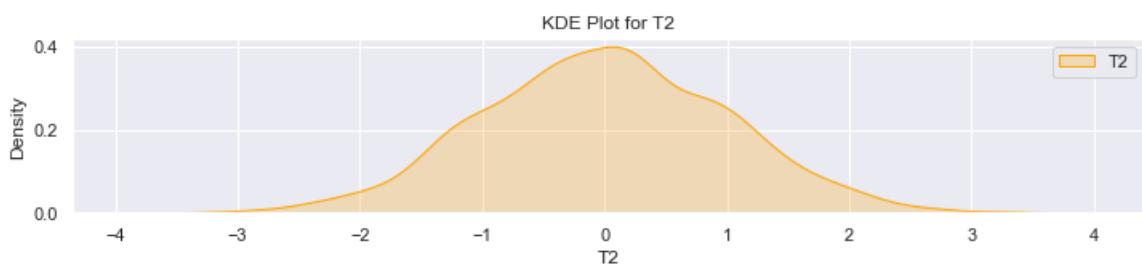
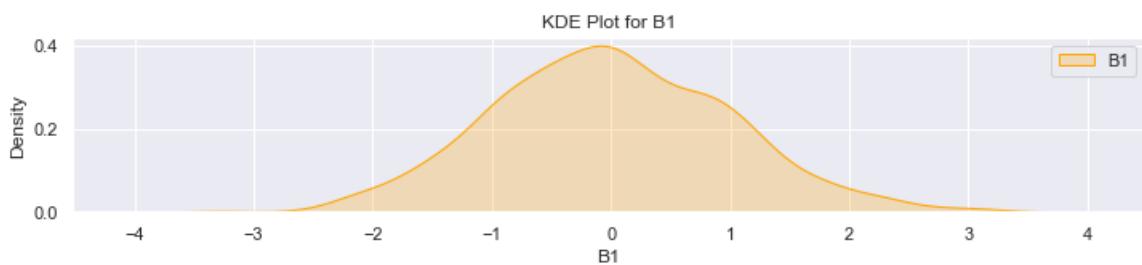
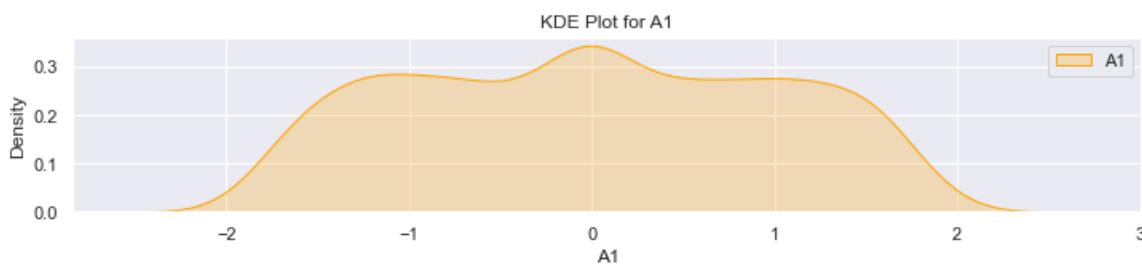
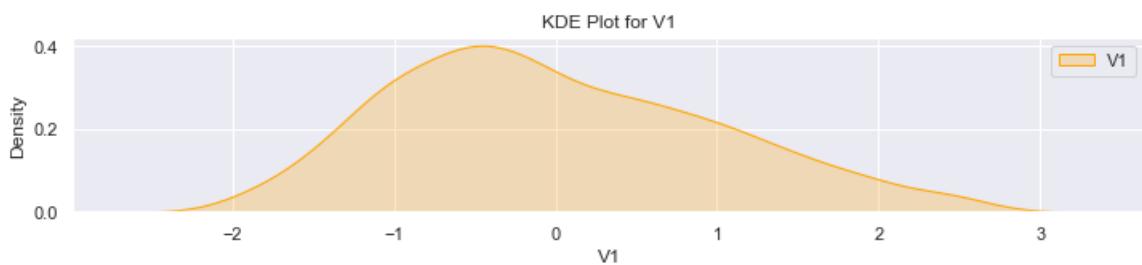
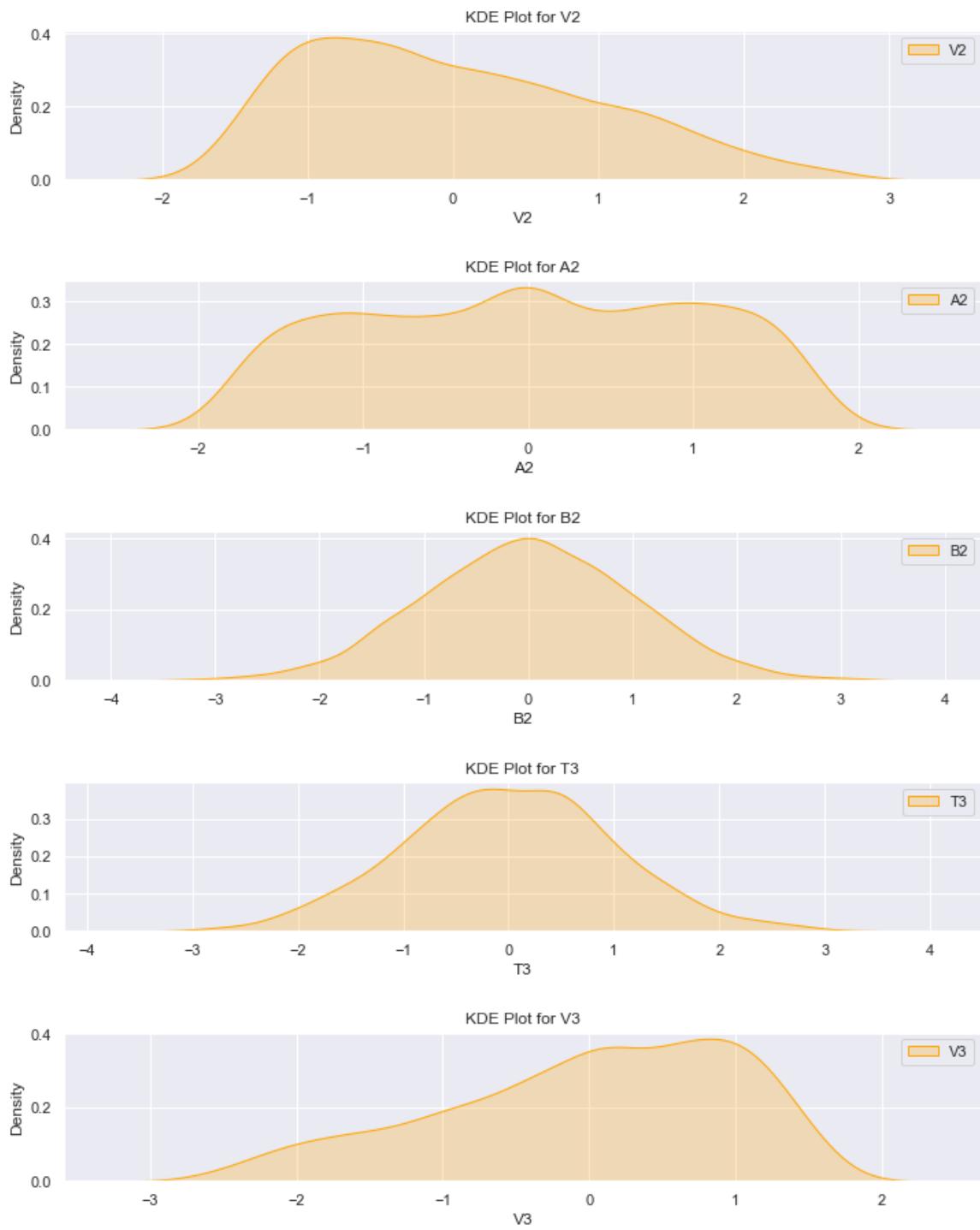
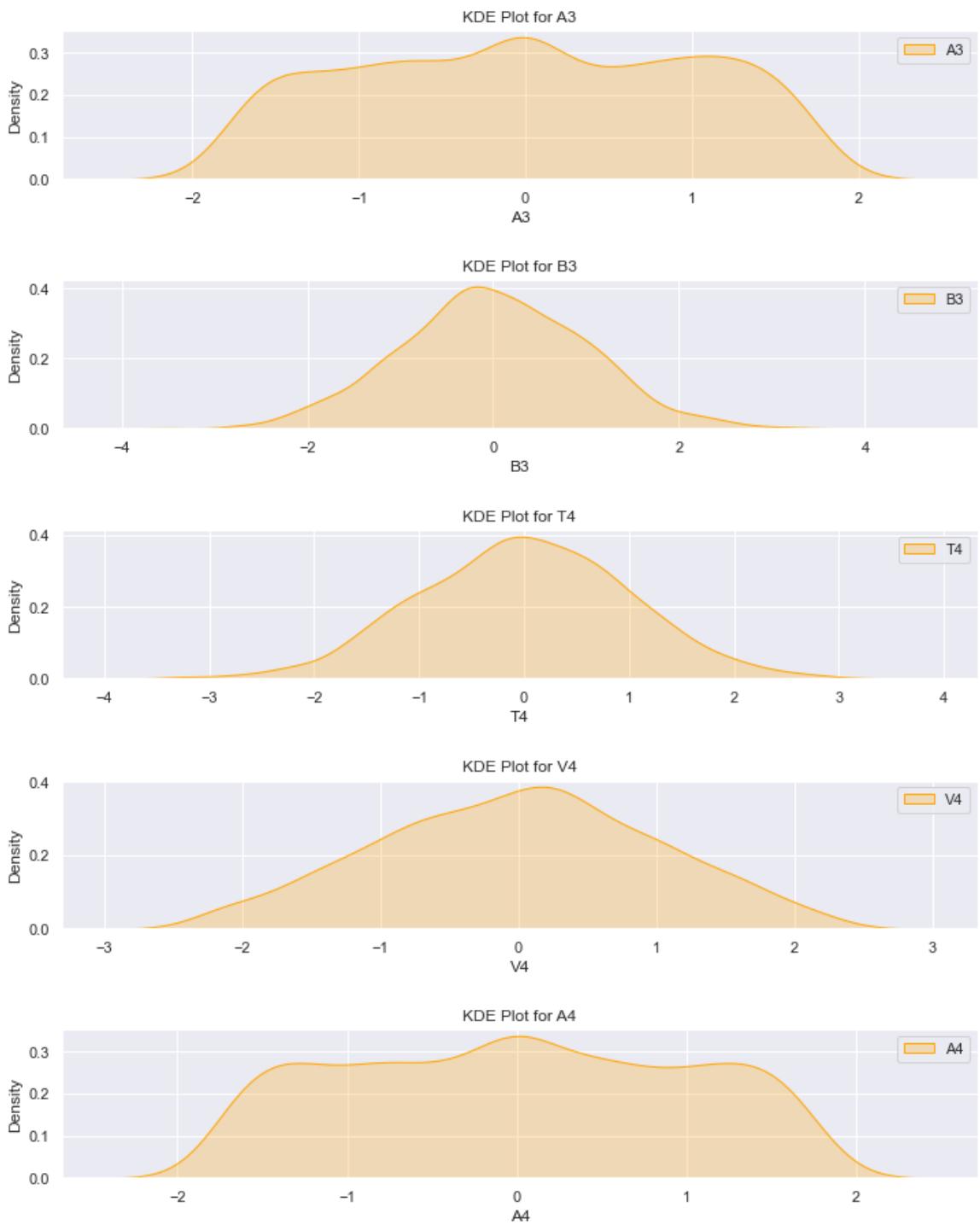
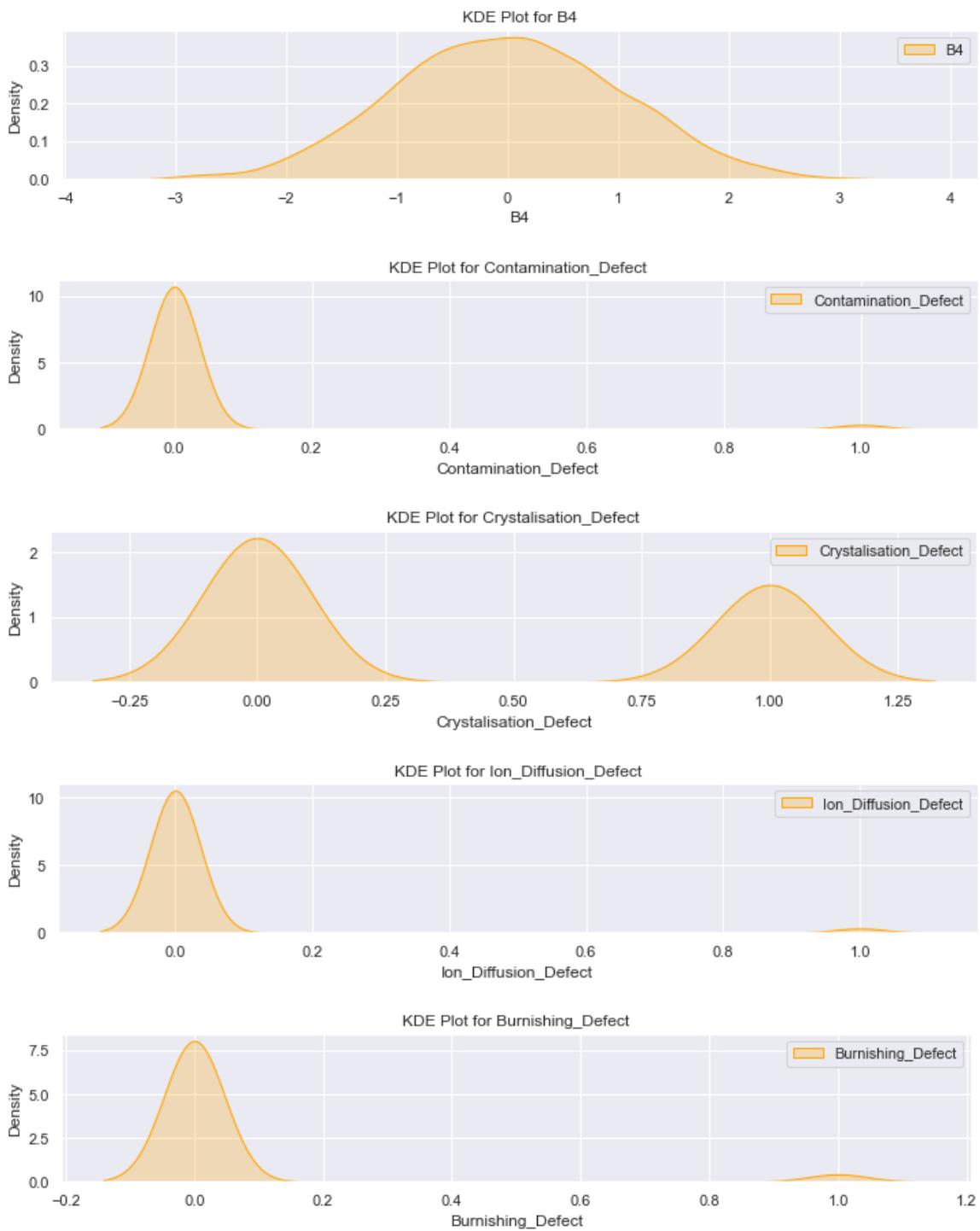


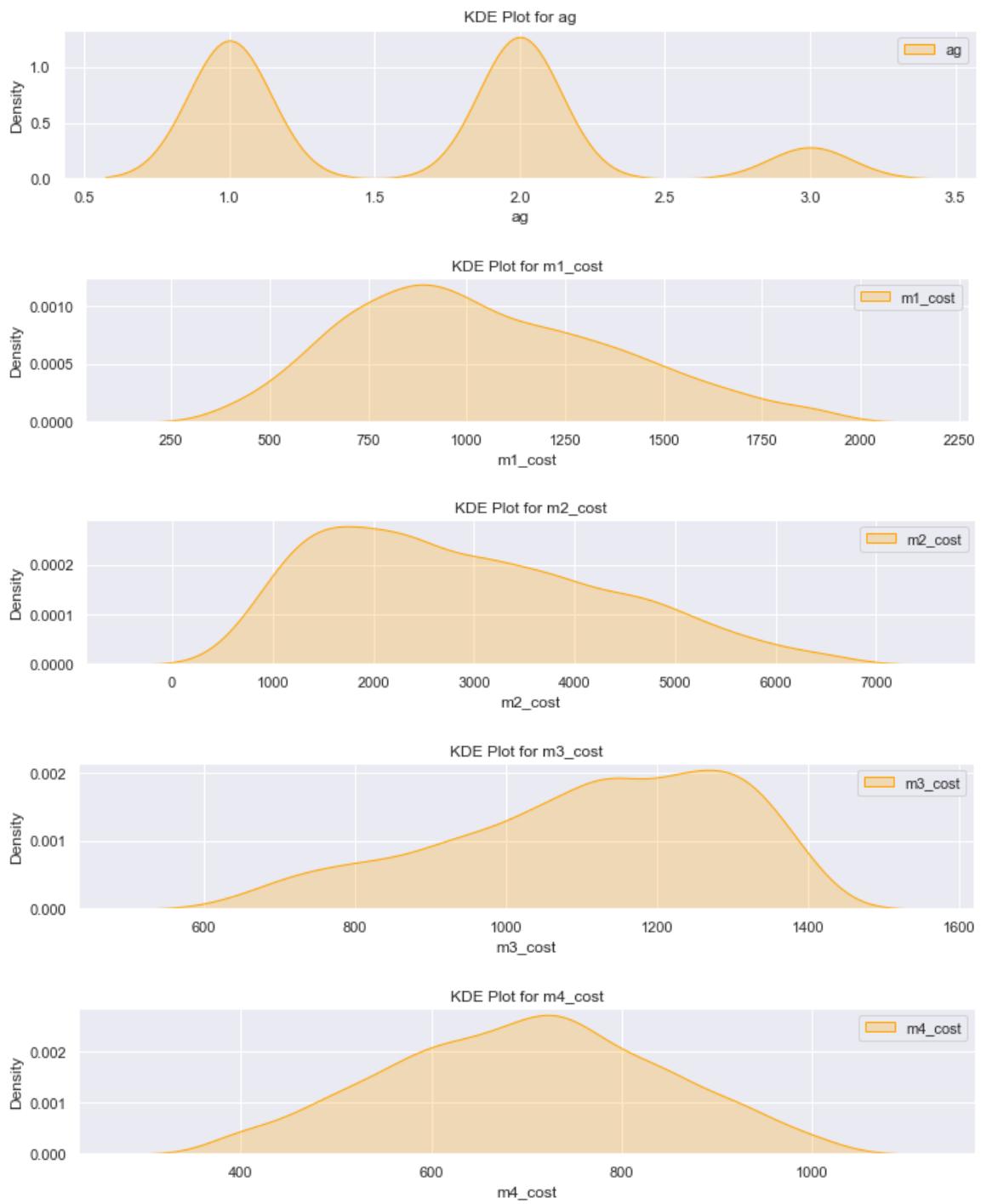
Figure 32: kde plot of the data

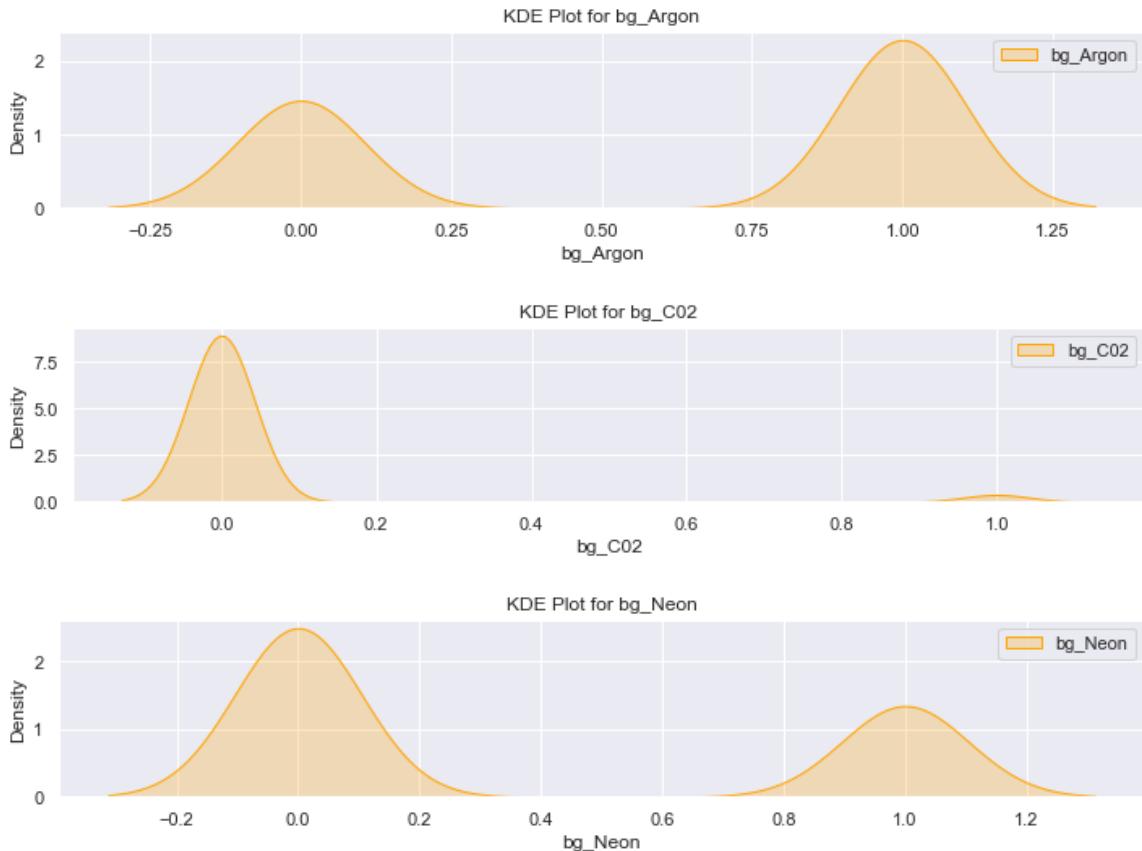












The above charts are usefully illustrative of the distributions of each feature. However, they are drawn with a different x scale. This means that similar groups of features are less easy to compare. The following code plots groups of features onto a single axis to enable easy scale comparison.

Tutor provided code

```
features_to_plot = ['m1_cost', 'm2_cost', 'm3_cost', 'm4_cost']

# Create subplots with shared x-axis
fig, axes = plt.subplots(nrows=len(features_to_plot),
                        figsize=(12, 2 * len(features_to_plot)),
                        sharex=True)

# Iterate through each specified feature and create a KDE plot
for i, column in enumerate(features_to_plot):
    sns.kdeplot(df_final[column],
                label=column,
                ax=axes[i],
                shade=True,
```

```

        color="orange")
axes[i].set_title(f'KDE Plot for {column}')
axes[i].legend()

```

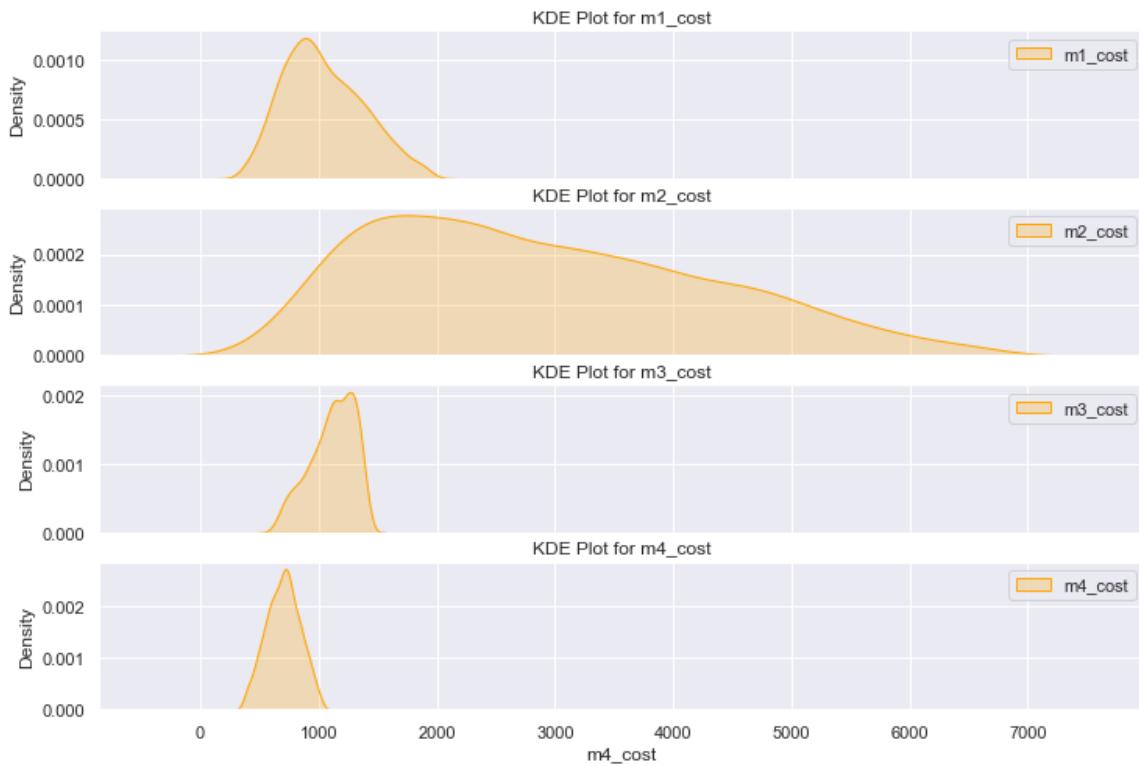


Figure 33: kde plots of selected features using a common x axis scale

You can also use a 2-Dimensional KDE plot to illustrate the relationship between pairs of features.

Tutor provided code

```

plt.figure(figsize=(7, 7))
sns.kdeplot(x=df_final['m4_cost'],
             y=df_final['m1_cost'],
             label=column,
             shade=True, color="blue")
plt.title(f'KDE Plot for {column}')
plt.legend()
plt.show()

```

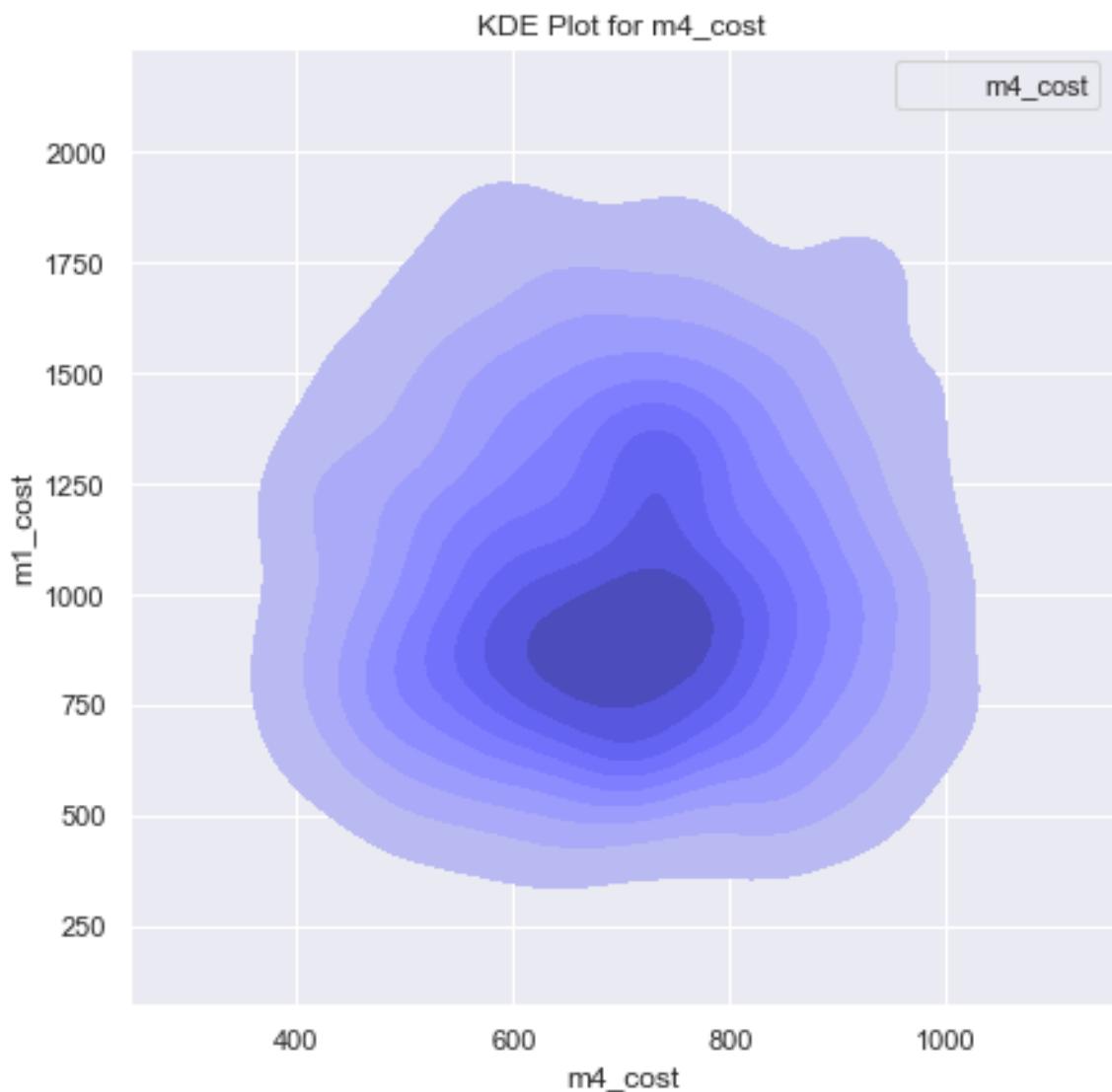


Figure 34: 2-Dimensional kde (surface contour) plot of selected features

1.25 Other Feature engineering tools (out of scope for this course)

1.25.1 QQ Plot

QQ plot from Chapter 3 of “Feature Engineering Cookbook”. Visual display of how much the data differs from a Gaussian distribution.

 Tutor provided code

```
import scipy.stats as stats
def diagnostic_plots(df, variable):

    plt.figure(figsize=(15,6))
    plt.subplot(1, 2, 1)
    df[variable].hist(bins=30)
    plt.title(f"Histogram of {variable}")
    plt.subplot(1, 2, 2)
    stats.probplot(df[variable], dist="norm", plot=plt)
    plt.title(f"Q-Q plot of {variable}")
    plt.show()
```

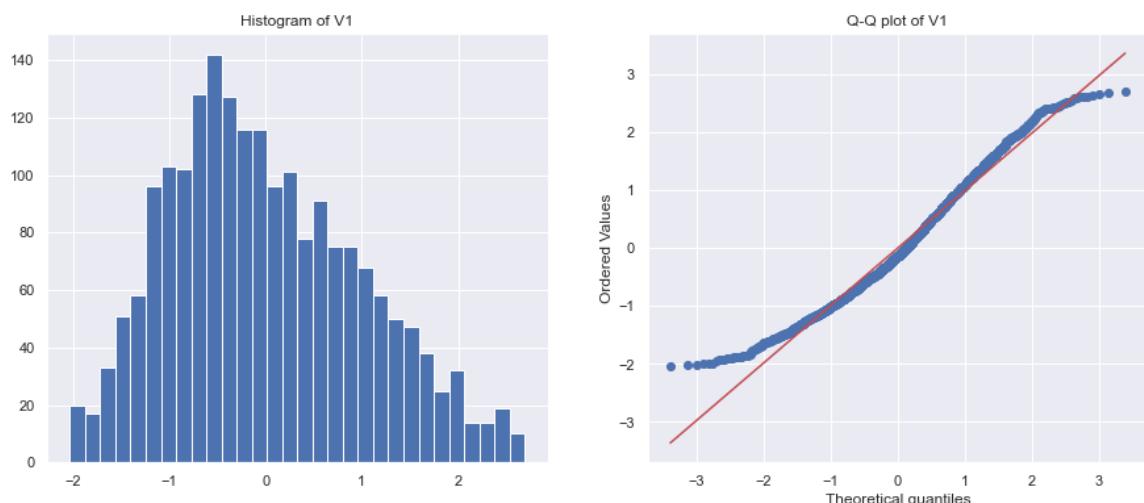


Figure 35: Q-Q plot to help show deviation from a Gaussian distribution

Chapter 2 - Implementing Gradient Descent

No-Code' Version for Student Practical Classes

Dr Rob Collins 2023

2025-01-11

Table of contents

2 Gradient Descent	2 - 4
2.1 Introduction	2 - 4
2.2 Instructions for Students	2 - 4
2.3 Load the required libraries	2 - 5
2.4 Generating some data which we can use to experiment with fitting	2 - 5
2.5 Fitting a model to the data	2 - 8
2.5.1 Initial parameter values	2 - 8
2.6 Calculating the Hypothesis function	2 - 9
2.7 Comparing the Hypothesis with our source data	2 - 10
2.8 Updating the parameters to improve the hypothesis	2 - 12
2.8.1 A short-cut method for calculating the gradients	2 - 13
2.9 Reviewing results of the first iteration	2 - 14
2.10 Iterative improvement of the model	2 - 14
2.11 Wrapping Gradient Descent as a single function	2 - 17
2.12 Extending to multiple dimensions	2 - 18
2.12.1 Step 1 : Generate synthetic data for testing our algorithm	2 - 19
2.12.2 Step 2 : Visualise	2 - 19
2.12.3 Step 3 : Apply gradient descent to improve the model	2 - 20
2.12.4 Step 4 : Visualize the model	2 - 21
2.12.5 <i>Student Task</i> : Experiment with the above:	2 - 21
2.13 Conclusion	2 - 22
2.14 Stochastic Gradient Descent	2 - 22
2.14.1 Testing the Stochastic Gradient Descent Function	2 - 23
2.14.2 A more direct chart display using Pandas	2 - 24

List of Figures

1 Famed explorer Liu Qiaoxuan calculating route down Yunwushan (c. 1783) .	2 - 3
2 Scatter-plot of Synthetic linear data with some Gaussian noise	2 - 7

3	Scatter-plot of Synthetic linear data and the current model	2 - 11
4	Scatter-plot of data vs model after model improvement	2 - 15
5	Scatter-plot of data vs model after many iterations	2 - 16
6	Scatter-plot of data vs model after many iterations (2)	2 - 18
7	3D plot of Synthetic Data before model fitting	2 - 20
8	3D plot of Synthetic Data and Corresponding Model	2 - 21
9	Line charts showing evolution of k and A towards best fit	2 - 24
10	Scatter-plot of model generated using SGD	2 - 25
11	Pandas version of line chart showing evolution of thetas	2 - 26



Figure 1: Famed explorer Liu Qiaoxuan calculating route down Yunwushan (c. 1783)

2 Gradient Descent

2.1 Introduction

In this workshop you will first build code that fits a straight line to a data-set. Later in the tutorial I will show how the same algorithm can be used to fit a hyper-planes to higher-dimensional data.

The method used to achieve this is called ‘Gradient Descent’ - and this is covered in the technical lectures for this course.

In practice, when building Machine Learning models you are unlikely to need to build your own data fitting algorithms. The various Machine Learning libraries include a range of data fitting algorithms. In the vast majority of cases you will choose to use those since they are much easier to use and almost always faster than the implementation below.

However, it is important to have some understanding of how the various fitting algorithms actually work. This is core material for Machine Learning. Without understanding how data fitting actually works many of the algorithms will appear simply as ‘magic’ - things that seem to work without any real scientific understanding of how they work.

You are thus advised to study this section carefully. Also, consider reading around this subject to learn about other data fitting algorithms.

2.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things

7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

2.3 Load the required libraries

For this practical session we will need the following libraries:

- ‘pandas’ : Conventionally given the reference name ‘pd’
- ‘numpy’ : Conventionally given the reference name ‘np’
- ‘random’
- ‘seaborn’ : Conventionally given the reference name ‘sns’
- ‘matplotlib.pyplot’ : Conventionally given the reference name ‘plt’
- ‘Axes3D’ : from mpl_toolkits.mplot3d

Create a cell to import all of these libraries.

 Your code here

2.4 Generating some data which we can use to experiment with fitting

The following function generates a set x and y data. The y data has a slope defined by ‘A’, a constants term (a ‘bias’ defined by ‘K’ and a level of random noise defined by ‘varience’.

There is one thing to note about the following function which may surprise you. We actually generate two ‘columns’ (features) of x data. The first column will be populated only with ‘1’s, rather than ‘real’ (variable / measured) x data. This is a common ‘trick’ that arises for the following reason:

We will fitting a straight line of the form $y = A.x + k$. Note that there are two terms to be identified in this case - the ‘A’ (slope of the line) and ‘k’ - the constant term or ‘bias’. Only one of these (‘A’) is multiplied by the x values.

As we evaluate trial (hypothesis) versions of fitted values, it will be very convenient to use library-based matrix multiplication operations - which are very fast and easy to use. Creating the ‘x’ table as two columns enables us to simply perform a matrix operations between our hypothesised values for ‘A’ and ‘k’ and the table of ‘x’ data. In that case, the ‘A’ will be multiplied by real (variable / measured) x values and our hypothesised ‘k’ values will all be multiplied by 1 - the value in the first column.

You will see in many lectures and implementations of data-fitting algorithms use this ‘trick’. By adding a vector of ’1’s in the first column, we simply enable more succinct matrix operations. You will see this used later in the workshop.

At this point, create a function to generate data in the form described above:

 Tutor provided code

```
def genData(numPoints, A, K, variance):
    x = np.zeros(shape=(numPoints, 2))
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i][0] = 1
        x[i][1] = i
        y[i] = (A * i + K) + random.uniform(0, 1) * variance
    return x, y
```

Use the ‘genData’ function to build a data-set of the form:

$$y = A.x + K + noise$$

Where in this specific case, the values are:

$$y = 2.x + 100 + random.uniform(30)$$

Remember that ‘genData’ has two return values.

 Your code here

We can now look at the data graphically using a ‘scatterplot’. There are several ways of plotting scatterplot in Python but my suggestion here is to use the ‘seaborn’ and ‘matplotlib.pyplot’ libraries.

Create a scatterplot chart that displays the data just created.

Note : You may wish to include the ‘%matplotlib inline’ command in your code. This forces the chart to appear as part of your Jupyter notebook rather than in a separate window

 Your code here

..and as explained above, the ‘shape’ of x is ‘m’ rows and 2 columns. You can demonstrate that using the numpy ‘shape’ function:

 Your code here

m (Length)= 100, and n (width)= 2

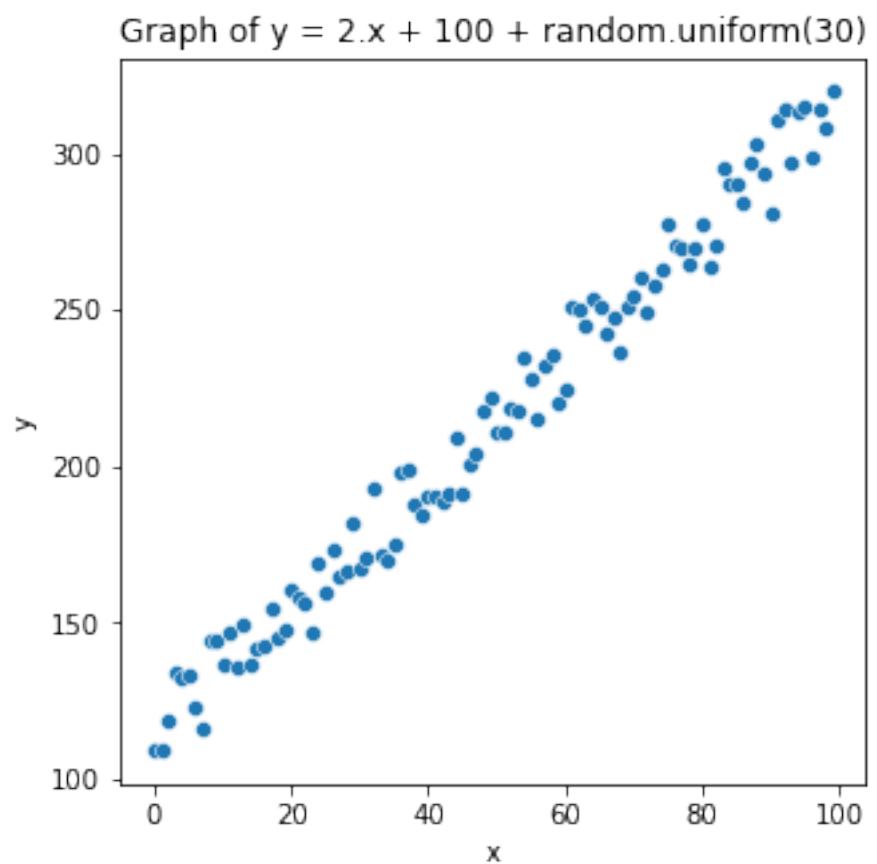


Figure 2: Scatter-plot of Synthetic linear data with some Gaussian noise

Add code to display the first few rows of x and note that the first column of x data contains only '1's

💡 Your code here

```
[[1. 0.  
 [1. 1.  
 [1. 2.  
 [1. 3.  
 [1. 4.]]
```

2.5 Fitting a model to the data

2.5.1 Initial parameter values

We set up the basic parameters for the data fitting. 'alpha' is the learning rate - the step size taken on each iteration.

'theta' is an array that will contain the parameter values we are searching for. theta[0] will contain the constant term ('k') we are searching for. theta[1] will contain the value of 'A'.

Now, in the following I could have chosen to use the variables 'k' and 'A' directly in the code. That would have made this tutorial somewhat easier to follow! However, it is not very extensible. Shortly, we will want to consider multi-dimensional linear models .. that is, models with multiple 'x' values:

$$y = k + A.x_1 + B.x_2 + C.x_3 + \dots$$

You can see that very quickly, all of these different variable names would become confusing and difficult to manage. Far better to give those constants (A, B, C ..) a uniform name and store them in an array. This reflects a common way of writing the above equation, which you will see in many text-books:

$$y = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \theta_3.x_3 + \dots + \theta_n.x_n$$

Where $x_0 = 1$

This formula also helps explain why the first column of our 'x' matrix was all set to the value 1.

Define a variable 'alpha' to represent the initial learning rate and set it to some small value.

Create a numpy array called ‘theta’ that will contain the learnt parameters of your model. The length of this array has to equal the number of parameters in the model - in this case 2.

Initially, set every element of theta equal to '1'.

(note Another common choice for starting values for parameters is to set the elements of the theta vector to a random number. It really makes no difference - we will be continually updating the values in the theta vector until they represent the best fit.)

 Your code here

```
alpha = 0.0005  
theta = [1. 1.]
```

2.6 Calculating the Hypothesis function

We are going to attempt to calculate a ‘Hypothesis’ function. This will be a current, ‘guess’ at the best solution to the problem.

We will update the values of ‘theta’ step-by-step to improve the hypothesis. Thus, the hypothesis will become a better and better approximation to the ‘correct’ solution.

In order to evaluate the ‘current’ version of our Hypothesis function, the x data needs to be the correct shape. You will often see this in Machine Learning books as computing the Transpose of the data: x^T . Using Numpy this can be written `x.T`:

 Your code here

This means that we can ‘multiply’ our source data (x) with the current best-guess parameters (theta) using the numpy ‘dot’ operator, to calculate the hypothesis.

Using the mathematical formulation above we are calculating:

$$\vec{y} = \theta_0 \cdot \vec{x}_0 + \theta_1 \cdot \vec{x}_1$$

Where I have indicated that the x’s and y’s in this formula are actually vectors (arrays) of numbers by using the common vector notation \vec{y}

So add a block of code that uses the numpy ‘dot’ operator to calculate a ‘best_guess’ by computing the dot-product of our thetas and the transposed data array x.

 Your code here

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.,
       12., 13., 14., 15., 16., 17., 18., 19., 20., 21., 22.,
       23., 24., 25., 26., 27., 28., 29., 30., 31., 32., 33.,
       34., 35., 36., 37., 38., 39., 40., 41., 42., 43., 44.,
       45., 46., 47., 48., 49., 50., 51., 52., 53., 54., 55.,
       56., 57., 58., 59., 60., 61., 62., 63., 64., 65., 66.,
       67., 68., 69., 70., 71., 72., 73., 74., 75., 76., 77.,
       78., 79., 80., 81., 82., 83., 84., 85., 86., 87., 88.,
       89., 90., 91., 92., 93., 94., 95., 96., 97., 98., 99.,
      100.])
```

It is worth taking a moment to work out what the above ‘np.dot’ is achieving. Essentially, it is applying our current best guess parameters (our ‘A’ and ‘k’) to each of the x values in turn. It is saying: What would our y look like if we accepted those parameters as our solution?

In this first iteration, our approximation to the correct solution is rather poor (of course it is, we simply guessed at ‘1’ for each of our parameters!)

2.7 Comparing the Hypothesis with our source data

We can compare this first ‘guess’ with the real y data by plotting them both on the same graph. Add some code to plot both the original data and the current model.

Define a function that plots two scatter plots on the same chart with the following signature:

```
def show_model(x, y, y_model, caption = "Model scatter plot"):
```

 Your code here

Use that function to plot a chart showing your data and your model.

hint : Remember that your x data is stored in an array with 2 columns (see above). You will need to extract the second column (column 1!) from x to pass to the function for plotting.

 Your code here

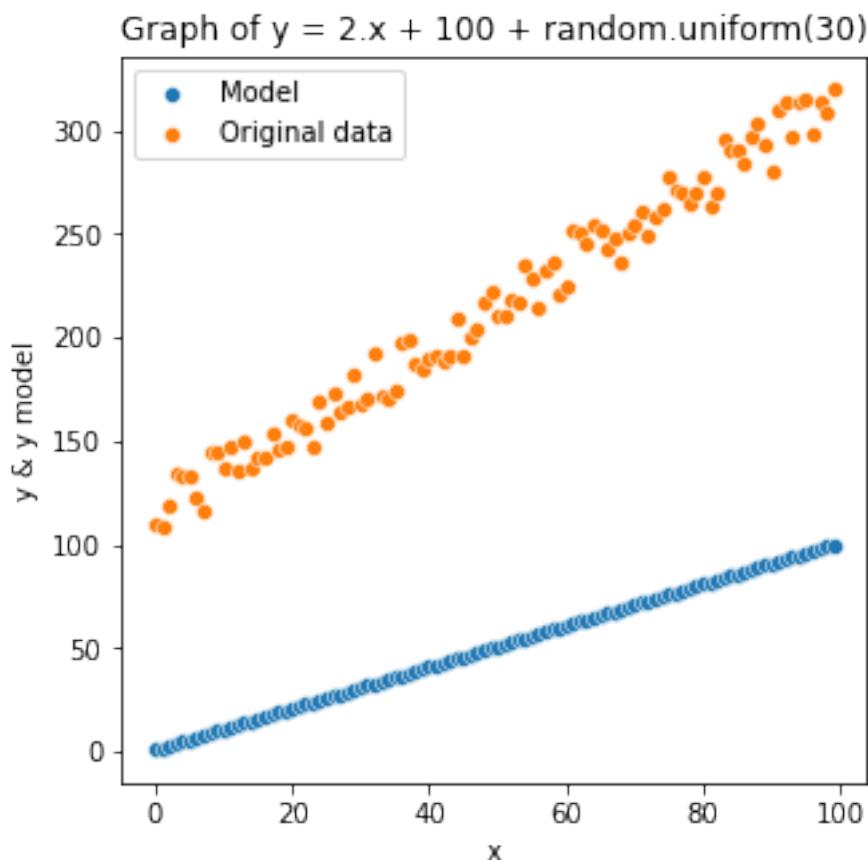


Figure 3: Scatter-plot of Synthetic linear data and the current model

An important question is .. how ‘wrong’ is our Hypothesis? What is the ‘distance’ between our hypothesis and the true set of y values? In the following we are using the l2 norm .. or Euclidean distance between the hypothesis and the true y values.

The difference for each data point can be calculated as the ‘epsilon’. Add code to create an array that represents the difference between the ‘best_guess’ and the true (measured) y values.

 Your code here

```
epsilon = [-108.70157221 -107.0948408 -115.99201135 -130.56732698 -127.8489362  
-127.23029528 -115.85609288 -108.61043665 -135.77696985 -134.72337021  
-126.00356144 -134.88557525 -123.24656315 -135.85387184 -121.81194152  
-125.99733295 -125.85467545 -136.3130653 -126.54752974 -127.60365882  
-139.75629607 -135.7708841 -133.51464586 -122.92931543 -144.28493605  
-133.43906698 -146.08757919 -136.58412056 -137.81246313 -151.91942018  
-136.71199931 -138.53054836 -159.95462119 -138.0825547 -135.20738594  
-138.79243741 -160.84708554 -160.6782652 -148.51053169 -144.53039432  
-149.49598975 -148.74657826 -145.52300763 -147.17395909 -164.19156326  
-145.53827118 -153.68806031 -156.13274569 -168.36322925 -172.35173722  
-159.87606059 -158.78591507 -165.57403861 -163.4949577 -179.79340728  
-172.40915068 -157.82177757 -174.20869009 -176.83388814 -160.41633982  
-163.52943465 -189.25965435 -187.35444469 -181.20811825 -188.93712248  
-185.41506556 -175.39309852 -179.90245902 -167.83344129 -180.9188989  
-183.17975931 -188.75840502 -176.70044394 -183.7184172 -187.69343771  
-201.39628273 -193.7626343 -191.92076108 -185.61970204 -190.02254979  
-196.52514047 -181.50652973 -187.43475634 -211.32776695 -205.48981171  
-204.16180161 -197.4964076 -209.0216591 -214.45988153 -203.5861462  
-189.52571429 -218.56648479 -220.83390258 -203.42623296 -218.50493663  
-219.02534813 -201.72299212 -215.85973489 -209.47004679 -219.80490364]
```

The average ‘cost’ as the sum of the squares of this array divided by the number of items.
Add code to produce a single value which is the sum of the squares of the loss.

 Your code here

```
Cost = 27742.41
```

Again, this is worth considering for a moment. The above number represents a ‘score’ for our hypothesis. It is the is a measure of the difference between the given y values, and a set of hypothetical y values that are generated by our current best-guess parameters for θ_0 and θ_1 . We need to minimize this value if we are to get a good fit.

2.8 Updating the parameters to improve the hypothesis

We now need to work out how to adjust θ_0 and θ_1 to improve our Hypothesis. As described in the lecture, this means determining the gradient of the cost function. The gradients are calculated using the following formula.

$$\frac{\partial}{\partial M} = \frac{2}{n} \sum_{i=1}^n (y_i - (k + M \cdot x_i)) \cdot x_i$$

$$\frac{\partial}{\partial k} = \frac{2}{n} \sum_{i=1}^n (y_i - (k + M \cdot x_i))$$

So thinking about this in terms of code ... first create a numpy array called ‘gradient’ that will be used to store the calculated gradients.

 Your code here

```
gradient = [0. 0.]
```

Now iterate through the x and y arrays in each case applying the above partial differential to calculate the gradients

 Your code here

```
gradient = [-163.71157875 -8956.62147806]
```

2.8.1 A short-cut method for calculating the gradients

In many examples of demonstration code for Gradient Descent you will see a ‘short-cut’ formula for the gradient. This formula gives exactly the same result as above, but is shorter to write and much faster.

An explanation for this formula is outside of the scope of the current course. However, if you wish to read more, about this **after class** then there is a nice explanation here:

<https://stats.stackexchange.com/questions/396735/intuition-behind-computing-gradient-for-a-model>.

Here, I print the results again, simply to demonstrate that the short-cut version gives the same answer as above:

 Tutor supplied code

```
gradient = np.dot(x.T, epsilon) / m
gradient
```

```
gradient = [-163.71157875 -8956.62147806]
```

This gives us the ‘slope’ in the cost function for a specific set of thetas (hypothesis parameters). We can therefore use this to update our set of hypothesised parameters (remembering that each of the variables in the following formula is actually a vector).

Add code to update your array of thetas using the gradient value you just calculated. Remember to scale your gradient by ‘alpha’ your learning rate.

💡 Your code here

```
theta = [1.08185579 5.47831074]
```

The two values above are a closer approximation to the ‘best fit’ values than were our originals (which were just [1,1].

These new parameters are the basis of our updated model.

Add code to re-calculate the ‘best_guess’ array based on those new thetas. **hint** this is just a copy of code you have already used above. Then print out the first few values of the ‘best_guess’.

💡 Your code here

```
Best guess[0:10] = [ 1.08185579 6.56016653 12.03847727 17.51678801 22.99509875 28.4734094  
33.95172022 39.43003096 44.9083417 50.38665244]
```

2.9 Reviewing results of the first iteration

Now review the results of this iteration by calling the function ‘show_model’ defined previously.

💡 Your code here

2.10 Iterative improvement of the model

Having taken all of the steps to improve our estimation of the parameters in one iteration, we can now apply exactly the same procedure to improve it again.

Because the learning-rate alpha is small, we need to iterate many times before the solution gets closer to the best fit value.

Add code that iterates over the ‘model improvement’ code multiple times then plots a chart of the updated model.

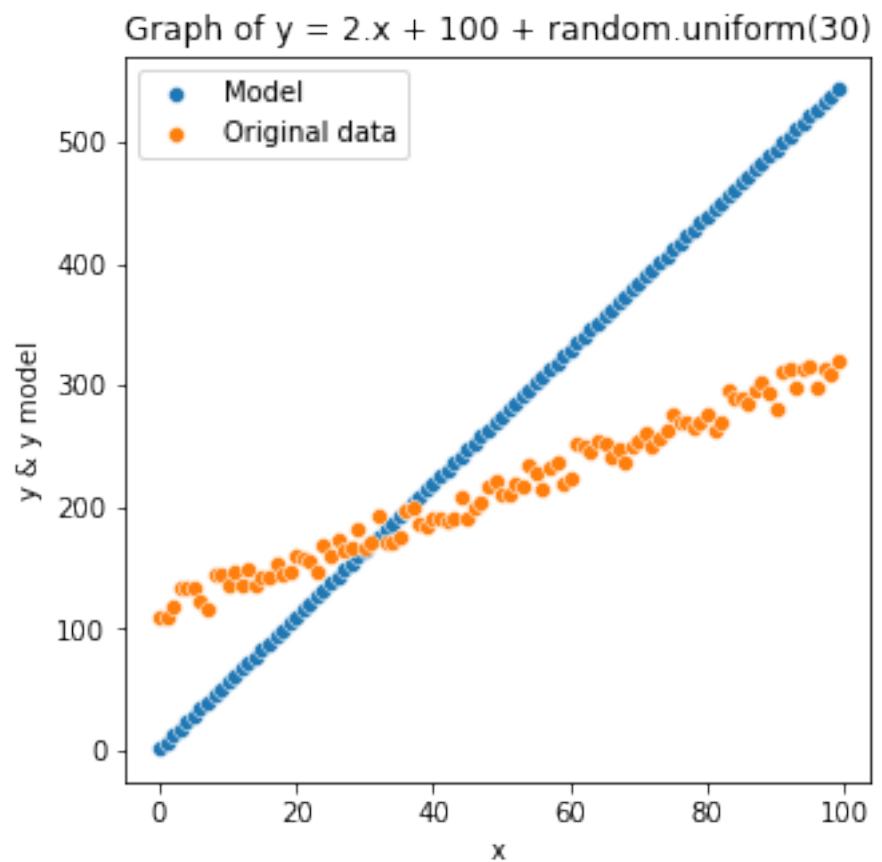


Figure 4: Scatter-plot of data vs model after model improvement

💡 Your code here

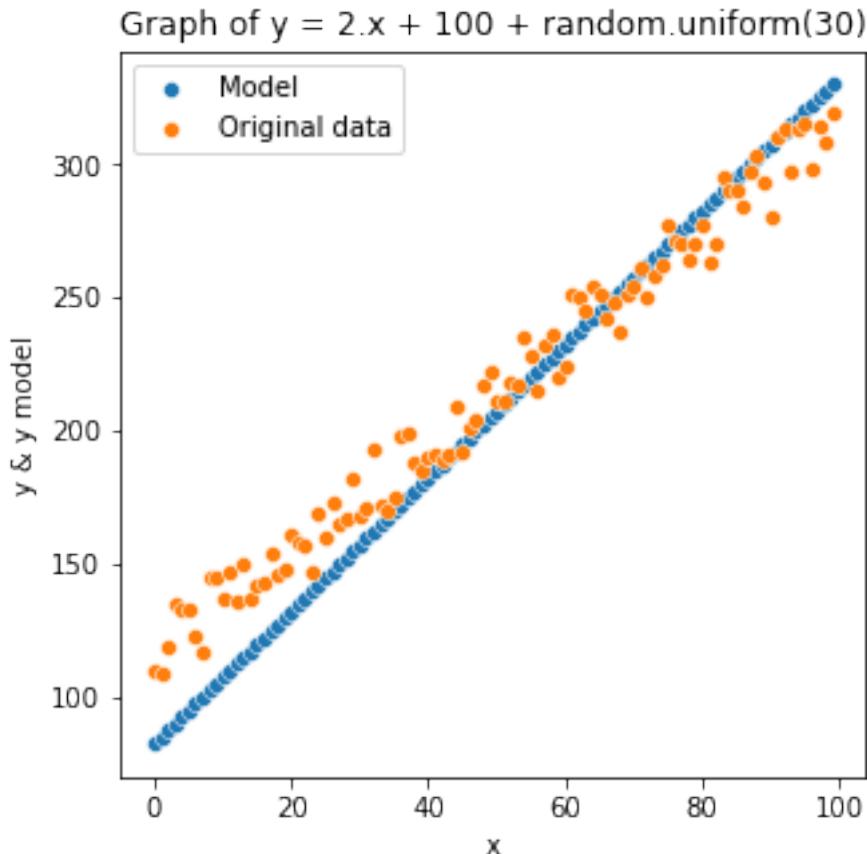


Figure 5: Scatter-plot of data vs model after many iterations

Student task : Now run the above code block several times and watch how each time the cost should reduce, and the fitted line should get closer to the data-set.

Eventually you will see that the rate of improvement slows down and the fit is as good as it is going to get.

At this point you can print the values of the parameters you have discovered for your model:

💡 Your code here

```
theta = [82.27019693  2.50262472], A = 2.50, k = 82.27
```

When I ran this I saw:

```
theta = [85.70479873  2.44094766], A = 2.44, k = 85.70
```

(You may see something slightly different since the data matrix is randomised each time this code is run)

The values above are a fairly good match to the original values of A=2 and k=100).

2.11 Wrapping Gradient Descent as a single function

In practice, it is much more convenient to wrap much of the above code into a re-usable function. Define a function with the following signature:

```
def gradientDescent(x, y, theta, alpha, numIterations):
```

Reproduce the code from the above sections into a single, re-usable function.

In your code, include some checking of the parameters:

1. That the x and y parameters are the same length
2. Check that the x parameter is a matrix with at least 2 columns

Your function should return the theta values for the updated model:

 Your code here

We can trial this function by generating some new data. Re-use the ‘genData’ function defined above. Experiment using different values for A, K and the variance

 Your code here

```
A = 2.0, K = 100.0, variance = 30.0
```

Repeatedly call the ‘gradientDescent’ function to improve the model

 Your code here

```
A = 3.66, k = 3.92
total_cost = 326102.39
```

Which we can again plot as a chart:

Your code here

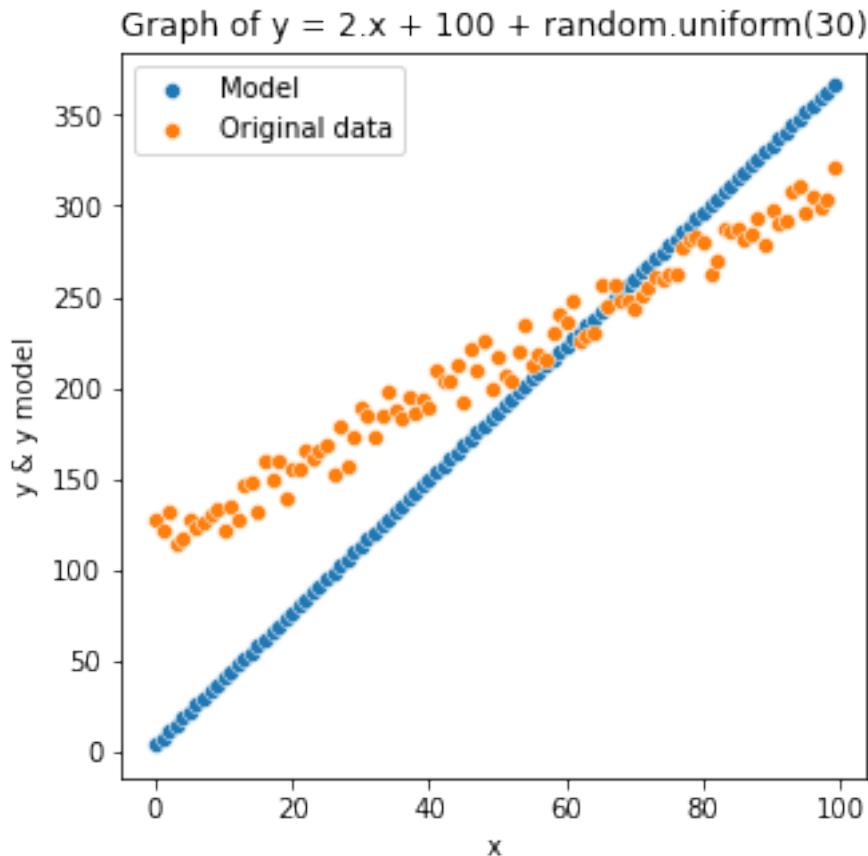


Figure 6: Scatter-plot of data vs model after many iterations (2)

2.12 Extending to multiple dimensions

One of the neat aspects of this algorithm is that it is easily extended to data-sets with multiple features (dimensions) .. creating a model of the form:

$$y = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3 + \dots + \theta_n \cdot x_n$$

Where $x_0 = 1$

We can demonstrate this by creating a new version of our data-set generator that creates multi-dimensional data. In this code we are passing an array of 'theta' values that will be used to create our synthetic data.

 Tutor provided code

```
def genDataM(numPoints, numDimensions, x_upper, thetas, variance):
    x = np.random.rand(numPoints, numDimensions)
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i][0] = 1
        y[i] = 0
        for j in range(0, len(thetas)):
            y[i] += x[i,j] * thetas[j]
        y[i] += random.uniform(0, 1) * variance
    return x, y
```

2.12.1 Step 1 : Generate synthetic data for testing our algorithm

Call ‘genDataM’ with the following parameters:

- ‘n = 3’ : the number of dimensions for this synthetic data
- thetas=[1,5,3] : The gradients in each dimension of the model
- variance = 5 : The spread of the Gaussian noise on each data-point

Create a new parameter array ‘theta’ with length ‘n’

 Your code here

```
n = 3
theta = [1. 1. 1.]
x[0:5] =
[[1.          0.49244429 0.1816613 ]
 [1.          0.49500228 0.56293045]
 [1.          0.59238729 0.74236173]
 [1.          0.60532583 0.41132928]
 [1.          0.73105483 0.90212802]]
```

2.12.2 Step 2 : Visualise

Plot this as a 3D chart:

note : You may use of the ‘magic’ directive ‘%matplotlib’ without a parameter to enable the output as a pop-out, rotatable, 3D chart in a window.

 Tutor supplied code

```
%matplotlib
fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:,1],x[:,2],y)
```

Using matplotlib backend: Qt5Agg

(None,)

3D Scatterplot of data to be modelled

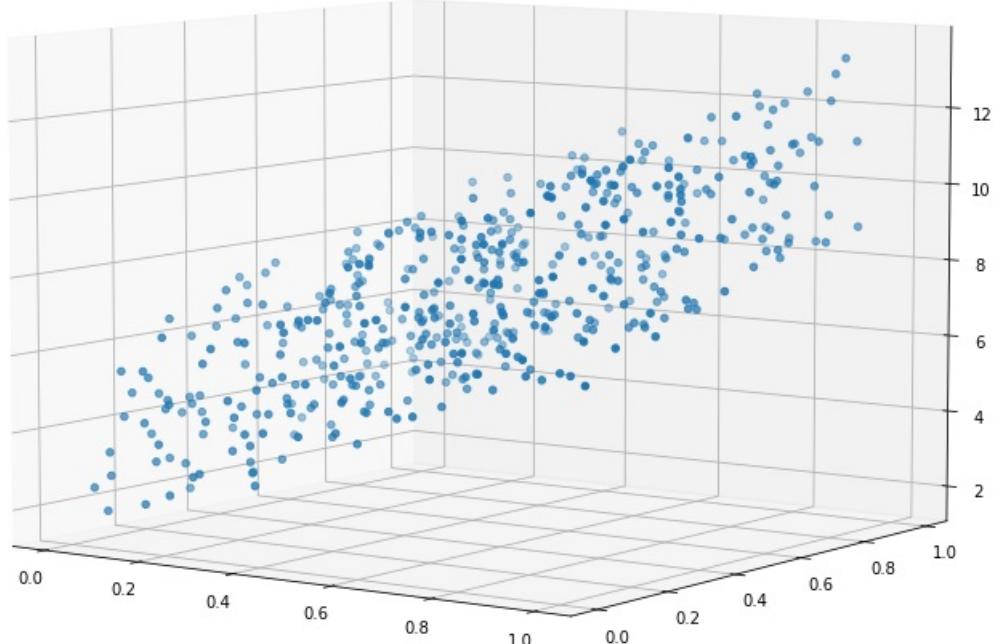


Figure 7: 3D plot of Synthetic Data before model fitting

2.12.3 Step 3 : Apply gradient descent to improve the model

Then use the ‘gradientDescent’ function to determine a best fit linear model for this data

 Your code here

```

A = 2.09, k = 2.91
total_cost = 4659.31

array([2.91094967, 2.08543584, 2.02760835])

```

2.12.4 Step 4 : Visualize the model

Then finally, plot a chart of the best fit model:

 Your code here

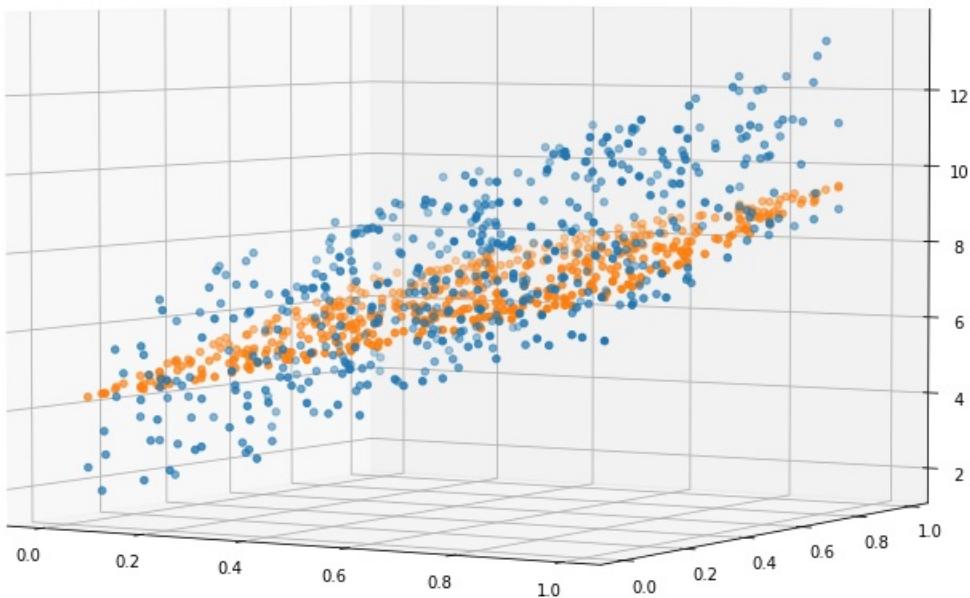


Figure 8: 3D plot of Synthetic Data and Corresponding Model

2.12.5 Student Task : Experiment with the above:

- 1. Repeatedly run the code blocks labelled ‘Step 3’ and ‘Step 4’. Each time you do this, a new set of coloured points should appear. Each of these represents a closer and closer approximation to the best fit model
- 2. If the chart gets too cluttered, re-run the code block labelled ‘Step 3’ - this will re-set the chart
- 3. Edit the code-block labelled ‘Step 1’ and change the theta parameters and the variance. Re-run code blocks 2,3 and 4 to look at the results

2.13 Conclusion

The ‘genDataM’ and the ‘gradientDescent’ functions above, can generate and fit a model to data with higher dimensionality. Of course, the fitting algorithm will get slower as the data set and the model becomes more complex. But nevertheless, it will work successfully over a wide range of data.

The algorithm demonstrated in this workshop is absolutely fundamental to an understanding of Machine Learning. It shows one important method for fitting a model to data. This core algorithm is then applicable to a range of other problems - as we shall see in later workshops.

2.14 Stochastic Gradient Descent

The algorithm for gradient descent described in the previous chapter requires that the model be evaluated for every data-point for every item of data. If the data-set is large, then this represents a significant amount of computation for each update.

Stochastic Gradient Descent (SGD) is a technique that achieves similar results, using far fewer computing resources. It is perhaps a surprising fact that a the model can be improved by iteratively ‘sampling’ from a data set and updating the model after every sample. In this case, the error for only the sample needs to be calculated for each update. Since the sample size is typically much smaller than the total data size this process is typically much more efficient than the full Gradient Descent algorithm described above.

There is also another reason to use SGD although that reason does not really apply in this simple case of linear curve fitting. However, in more complex modelling situations one of the issues may be that the cost function is not convex; there may be local minima in the cost surface and simple gradient descent may converge to one of those local minima. SGD is rather less susceptible to this phenomena since sampled data points introduce a level of ‘noise’ into the descent process than can help escape from local minima. So whilst not being relevant in this specific case, it is useful to understand SGD because of the variety of benefits it provides.

You should now attempt to implement your own Stochastic Gradient Descent function based on this description and the code you have already developed above. Work on the single dimension problem to start with, rather than trying to develop a full multi-dimensional model (my example code is the single dimensional case). In my solution I also use a sample size of 1 for each iteration. This is only a small simplification and you may optionally decide to iterate over larger samples.

It is interesting to record the evolving values of your model parameters (thetas) - so for each iteration of your model, append the values of theta to a list called ‘theta_list’.

The signature for your function should be this:

```
def sgd(x, y, theta, alpha, numIterations):
```

the return value should be:

```
return theta, theta_hist
```

This is a rather harder exercise than those given previously .. but actually you have most of the code you will need in the above sections. The only ‘trick’ here is that, rather than calculating epsilon for a whole array of model_y vs y values, you only need to do this for a sample from the input training set on each iteration. In my solution I selected a single specific item using code something like this:

```
sample_index = np.random.randint(0, y.size)
x_item = x[sample_index][1]
y_item = y[sample_index]
```

💡 Your code here

2.14.1 Testing the Stochastic Gradient Descent Function

Now we can test the above SGD function by generating new test data using your original genData function

💡 Your code here

You will need to define some parameters for your model:

- alpha : learning rate. Suggested value, 0.0001
- n : the number of dimensions, in this case - just 2
- theta : A numpy array of dimension ‘n’, initialized with ones
- number_of_iterations : Suggested value 100,000

Remember that, in experimenting with alpha, the larger the value of alpha the faster the model will converge. **But** : With a larger alpha:

1. The model may become unstable
2. In the case of SGD, the final results will be more ‘noisy’

💡 Your code here

Finally, call your sgd function:

💡 Your code here

```
k = 174.15,  
A = -9.38, mean squared distance = 3659.58
```

It is also interesting to plot the evolving values for 'k' and 'A' (theta[0] and theta[1]).

Write some code to plot two line charts - one for values of k and the other for values of A as returned in the 'theta_hist' list

💡 Your code here

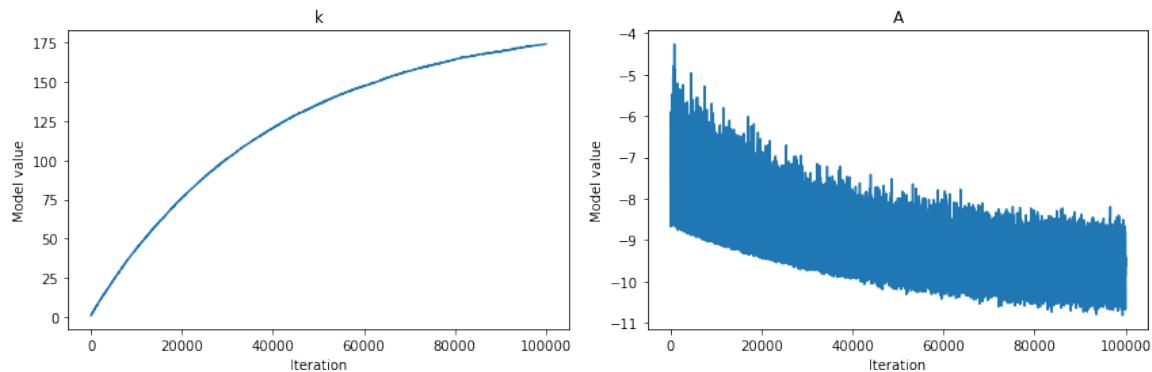


Figure 9: Line charts showing evolution of k and A towards best fit

If you wish, you can plot the actual model using the 'show_model' function developed above.

💡 Your code here

2.14.2 A more direct chart display using Pandas

An alternative, and somewhat simpler method for showing the history of theta values is included below

💡 Tutor provided code

```
%matplotlib inline  
for_plotting_df = pd.DataFrame(theta_hist)  
  
# Plot theta history as a line chart  
for_plotting_df.plot()  
  
plt.xlabel('Iteration')
```

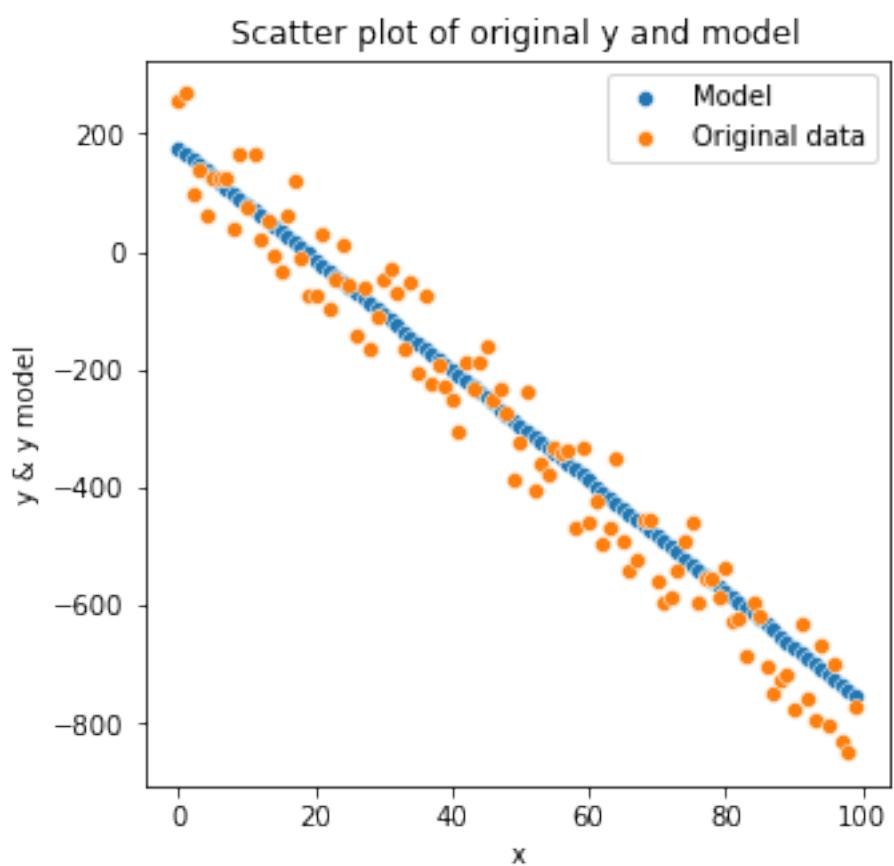


Figure 10: Scatter-plot of model generated using SGD

```
plt.ylabel('Thetas')
plt.title('Line Chart of Values over Time')

plt.show()
```

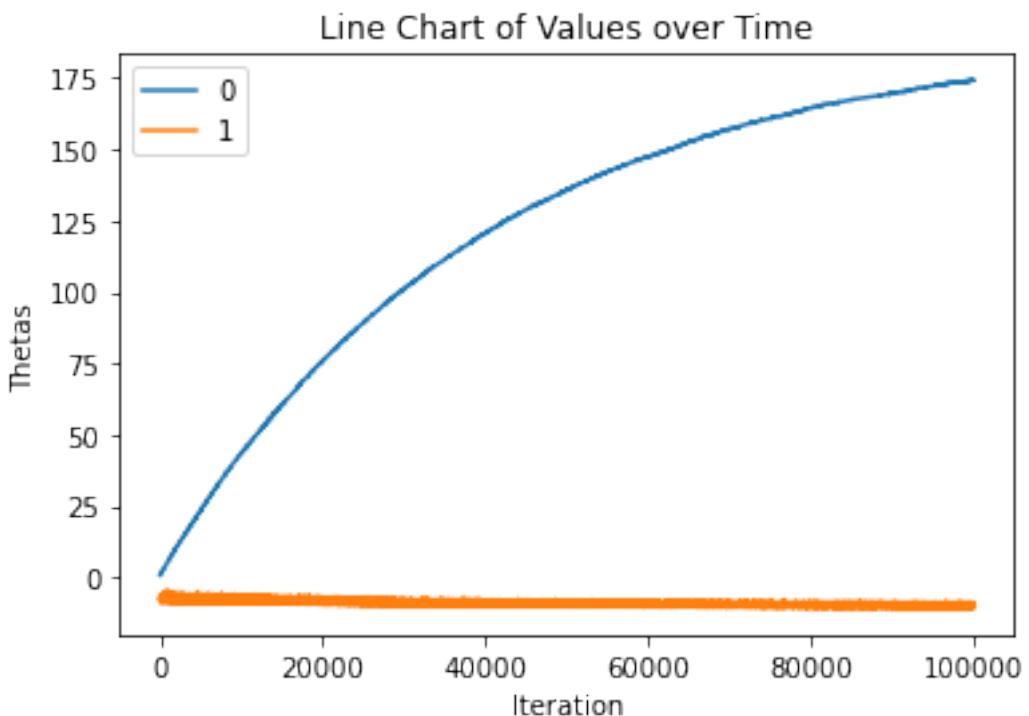


Figure 11: Pandas version of line chart showing evolution of thetas

Chapter 3 - End-to-end Supervised learning

No-Code Version for Student Practical Classes

Dr Rob Collins 2023

2025-01-11

Table of contents

3 End-to-end Supervised Learning	3 - 3
3.1 Introduction	3 - 3
3.2 Instructions for Students	3 - 3
3.3 Load the required libraries	3 - 4
3.4 Load the data	3 - 4
3.5 Do a basic review of the data	3 - 4
3.6 Step 1 : Clean and Tidy Data	3 - 7
3.6.1 Visual Check of Missing Data	3 - 7
3.6.2 Repair missing data	3 - 7
3.6.3 Change all of the category data into numbers	3 - 11
3.7 Step 2 Select the Algorithm	3 - 13
3.8 Step 3 Build the Model	3 - 13
3.9 Step 4 Check Model Quality	3 - 15
3.10 Step 5: Build the model into an application	3 - 19

List of Figures

1 Introduction of the first ‘Pecuniary Judex’ machine into the Pennyworth Provident Trust Bank of London. A most remarkable machine capable of determining good prospects from bad based entirely on scientific measurement	3 - 2
2 missingno matrix of the credit_data dataframe to identify missing data . . .	3 - 8
3 Missingno bar chart	3 - 8
4 Confusion Matrix display using Seaborn library	3 - 18



Figure 1: Introduction of the first 'Pecuniary Judex' machine into the Pennyworth Provident Trust Bank of London. A most remarkable machine capable of determining good prospects from bad based entirely on scientific measurement

3 End-to-end Supervised Learning

3.1 Introduction

In this workshop session we will be creating a loan credit decision model. That is, if we lend money to somebody - then what is the probability that the money will be paid back?

The model uses a very well known data-set called the “German Credit Data” which is widely available on the Internet. The version of the data provided to complete this workshop contains some omissions - so it will require some feature engineering before it can be used..

The model we will build is a ‘logistic regression’. Logistic Regression is a common choice when it is required to predict probabilities from regression models - since the results of the model are in the range 0 to 1.

3.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

3.3 Load the required libraries

For this practical session we will need the following libraries:

- ‘pandas’ : Conventionally given the reference name ‘pd’
- ‘matplotlib.pyplot’ : Conventionally given the reference name ‘plt’
- ‘missingno’ : Conventionally given the reference name ‘msno’
- ‘LogisticRegression’ : from sklearn.linear_model
- ‘train_test_split’ : from sklearn.model_selection
- ‘confusion_matrix’ : from sklearn.metrics
- ‘seaborn’ : Conventionally given the reference name ‘sns’
- ‘classification_report’ : from sklearn.metrics

sklearn ‘LogisticRegression’ will be used to build a classification model. This is documented within the ‘sklearn’ library documentation (See:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

‘train_test_split’ will be used to divide the data-set into two parts - one for model building and the other for model testing.

‘confusion_matrix’ is used to calculate and display classification test results.

‘seaborn’ is used here to provide an attractive display of the confusion matrix.

‘classification_report’ provides another option for summarising and displaying test results.

Create a cell to import all of these libraries.

💡 Your code here

3.4 Load the data

The data-file for this workshop is called ‘german credit data unclean.csv’. Create a cell to load it into a Pandas dataframe called ‘credit_data’

💡 Your code here

3.5 Do a basic review of the data

Let’s get a feel for what this data looks like in Python. List the columns names and brief descriptions using the Pandas ‘info()’ method.

Your code here

```
credit_data.info() =  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 1000 entries, 0 to 999  
Data columns (total 21 columns):  
 #   Column           Non-Null Count  Dtype     
---  --  
 0   check_account_status    1000 non-null   object    
 1   duration             1000 non-null   int64     
 2   credit_history        1000 non-null   object    
 3   purpose              1000 non-null   object    
 4   credit_amount         987 non-null   float64   
 5   savings_account       1000 non-null   object    
 6   employment_duration   1000 non-null   object    
 7   percent_disposable_income 1000 non-null   int64     
 8   gender_marriage       1000 non-null   object    
 9   other_debtors         1000 non-null   object    
 10  recidient_since      1000 non-null   int64     
 11  property              1000 non-null   object    
 12  age                  988 non-null   float64   
 13  other_plans          1000 non-null   object    
 14  housing               1000 non-null   object    
 15  num_credits           1000 non-null   int64     
 16  job                  1000 non-null   object    
 17  dependents            1000 non-null   int64     
 18  telephone             1000 non-null   object    
 19  foreign_worker        1000 non-null   object    
 20  default               1000 non-null   int64     
dtypes: float64(2), int64(6), object(13)  
memory usage: 164.2+ KB
```

Some of those column names are quite long and will make the display of dataframes rather wide. We can change them to something shorter...

Create two lists:

- A list of current feature names you want to change
- A list of the updated feature names

Zip those two lists into a single dictionary that links ‘old name’ -> ‘new name’

Then use the Pandas ‘rename’ method to change the feature names. In that method set ‘inplace = True’ so that you don’t create a completely new dataframe.

 Your code here

Changing feature (column) names:

```
check_account_status --> cs
credit_history --> ch
credit_amount --> ca
savings_account --> sa
employment_duration --> ed
percent_disposable_income --> pd
gender_marriage --> gm
other_debtors --> od
resident_since --> rs
foreign_worker --> fw
other_plans --> op
num_credits --> nc
dependents --> dp
duration --> dr
purpose --> pr
housing --> hs
telephone --> tp
property --> pp
```

Let's now take a look at the actual data ...

Here I have decided to display only the first 12 rows of data .. you may experiment with this to display different portions of the data. I have also separated the display of my table over several cells - this is so that it formats correctly when exported into the pdf book format. In your case, you can simply display this in one cell.

	cs	dr	ch	pr	ca	sa	ed	pd	gm	od
0	A11	6	A34	A43	1169.0	A65	A75	4	A93	A101
1	A12	48	A32	A43	5951.0	A61	A73	2	A92	A101
2	A14	12	A34	A46	2096.0	A61	A74	2	A93	A101
3	A11	42	A32	A42	7882.0	A61	A74	2	A93	A103
4	A11	24	A33	A40	4870.0	A61	A73	3	A93	A101
5	A14	36	A32	A46	9055.0	A65	A73	2	A93	A101
6	A14	24	A32	A42	2835.0	A63	A75	3	A93	A101
7	A12	36	A32	A41	6948.0	A61	A73	2	A93	A101
8	A14	12	A32	A43	3059.0	A64	A74	2	A91	A101
9	A12	30	A34	A40	5234.0	A61	A71	4	A94	A101
10	A12	12	A32	A40	1295.0	A61	A72	3	A92	A101
11	A11	48	A32	A49	NaN	A61	A72	3	A92	A101

	rs	pp	age	op	hs	nc	job	dp	tp	fw	default
0	4	A121	67.0	A143	A152	2	A173	1	A192	A201	0
1	2	A121	22.0	A143	A152	1	A173	1	A191	A201	1
2	3	A121	49.0	A143	A152	1	A172	2	A191	A201	0
3	4	A122	Nan	A143	A153	1	A173	2	A191	A201	0
4	4	A124	53.0	A143	A153	2	A173	2	A191	A201	1
5	4	A124	35.0	A143	A153	1	A172	2	A192	A201	0
6	4	A122	53.0	A143	A152	1	A173	1	A191	A201	0
7	2	A123	35.0	A143	A151	1	A174	1	A192	A201	0
8	4	A121	61.0	A143	A152	1	A172	1	A191	A201	0
9	2	A123	28.0	A143	A152	2	A174	1	A191	A201	1
10	1	A123	25.0	A143	A151	1	A173	1	A191	A201	1
11	4	A122	24.0	A143	A151	1	A173	1	A191	A201	1

You may notice immediately there there is some missing data in this table (Row 11 of the ‘credit_amount’ (ca) feature). This is important and gives us a clue that we need to clean and tidy the data.

3.6 Step 1 : Clean and Tidy Data

3.6.1 Visual Check of Missing Data

We are going to visualize missing data using the ‘missingno’ library. You should be familiar with this library from the earlier ‘Feature Engineering’ workbook.

Add a cell to create a graphic display that allows you to visualize missing data in the ‘credit_data’ dataframe.

💡 Your code here

Use a second missingno function to display a bar-chart of the number of data items in each feature of the credit_data data-set.

💡 Your code here

3.6.2 Repair missing data

You should apply two different strategies to ‘repair’ the missing data:

1. Removing any data records with a missing ‘credit_amount’, and
2. Imputing missing values for the ‘age’ feature - using the average age from the data-set

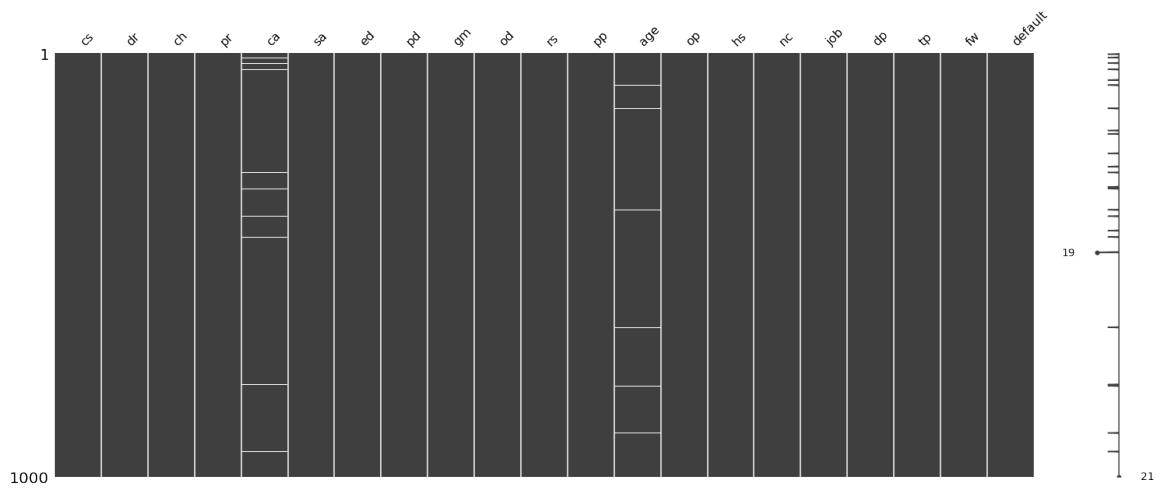


Figure 2: missingno matrix of the credit_data dataframe to identify missing data

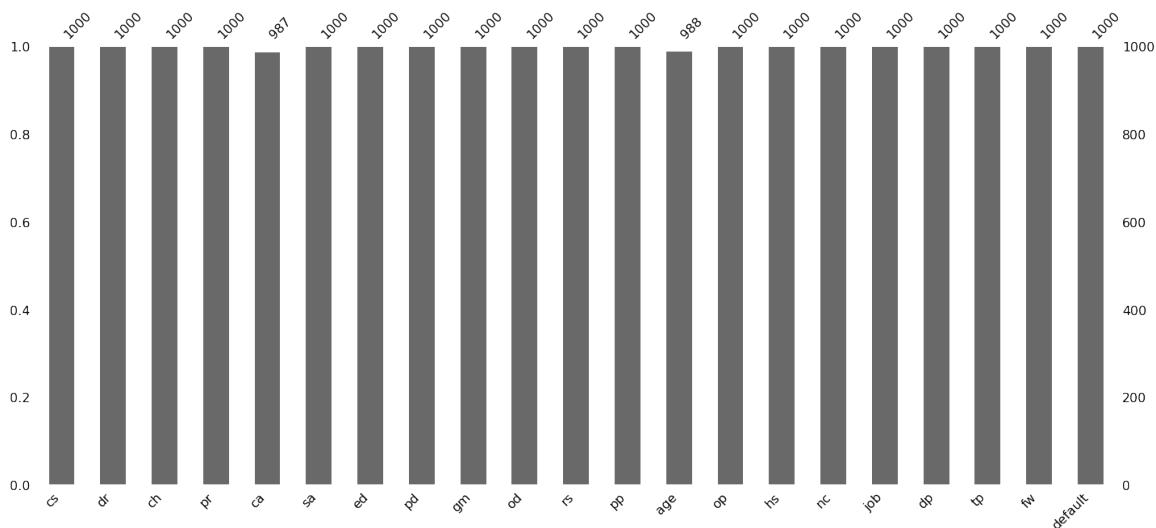


Figure 3: Missingno bar chart

Add Python code that removes any row of data that has a ‘NaN’ in the ‘credit_amount’ feature.

Hint: There reference for the Pandas ‘dropna’ function is here:

[<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html>]

💡 Your code here

	cs	dr	ch	pr	ca	sa
0	A11	6	A34	A43	1169.0	A65
1	A12	48	A32	A43	5951.0	A61
2	A14	12	A34	A46	2096.0	A61
3	A11	42	A32	A42	7882.0	A61
4	A11	24	A33	A40	4870.0	A61
5	A14	36	A32	A46	9055.0	A65
6	A14	24	A32	A42	2835.0	A63
7	A12	36	A32	A41	6948.0	A61
8	A14	12	A32	A43	3059.0	A64
9	A12	30	A34	A40	5234.0	A61
10	A12	12	A32	A40	1295.0	A61
12	A12	12	A32	A43	1567.0	A61

Check the above table ... Row 11, amongst others, should have been deleted and there should be 987 rows remaining in the data-set. The ‘Age’ feature will still contain ‘NaN’ values.

💡 Your code here

```
credit_data.info() =  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 987 entries, 0 to 999  
Data columns (total 21 columns):  
 #   Column   Non-Null Count  Dtype     
---  --  -----  --  
 0   cs        987 non-null    object  
 1   dr        987 non-null    int64  
 2   ch        987 non-null    object  
 3   pr        987 non-null    object  
 4   ca        987 non-null    float64  
 5   sa        987 non-null    object  
 6   ed        987 non-null    object
```

```

7   pd      987 non-null    int64
8   gm      987 non-null    object
9   od      987 non-null    object
10  rs      987 non-null    int64
11  pp      987 non-null    object
12  age     976 non-null    float64
13  op      987 non-null    object
14  hs      987 non-null    object
15  nc      987 non-null    int64
16  job     987 non-null    object
17  dp      987 non-null    int64
18  tp      987 non-null    object
19  fw      987 non-null    object
20  default 987 non-null    int64
dtypes: float64(2), int64(6), object(13)
memory usage: 169.6+ KB

```

Add some code that prints out a few of the rows where the age feature is NaN

 Your code here

For display reasons, I only printed a few columns:

	sa	ed	pd	gm	od	rs	pp	age	op	hs
3	A61	A74	2	A93	A103	4	A122	NaN	A143	A153
76	A61	A72	4	A93	A101	3	A123	NaN	A143	A152
131	A61	A73	4	A93	A101	3	A122	NaN	A142	A152
183	A64	A73	4	A93	A101	4	A121	NaN	A143	A152
268	A61	A75	1	A91	A101	4	A122	NaN	A143	A152
316	A61	A73	2	A93	A103	3	A122	NaN	A143	A152
370	A65	A73	4	A93	A101	4	A121	NaN	A143	A152
419	A65	A73	4	A92	A101	2	A122	NaN	A143	A152
647	A63	A73	2	A92	A101	2	A122	NaN	A143	A152
785	A64	A73	4	A93	A101	2	A122	NaN	A143	A152

Note that row 3 of ‘age’ is a NaN .. we can check that in a moment after fixing.

Add Python code that replaces any missing (‘NaN’) values in the ‘Age’ feature with the average of all ages in the data-set.

hint : The following may be useful:

[<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html>]

 Your code here

Show the results by displaying a few rows of the relevant features of the dataframe.

 Your code here

	sa	ed	pd	gm	od	rs	pp	age	op	hs
0	A65	A75	4	A93	A101	4	A121	67.000000	A143	A152
1	A61	A73	2	A92	A101	2	A121	22.000000	A143	A152
2	A61	A74	2	A93	A101	3	A121	49.000000	A143	A152
3	A61	A74	2	A93	A103	4	A122	35.580943	A143	A153
4	A61	A73	3	A93	A101	4	A124	53.000000	A143	A153

3.6.3 Change all of the category data into numbers

The file includes a number of features that are coded into categories. For example ‘sa’ is coded as A65, A61 etc. We need to convert those categories into numbers ...

Use the Pandas ‘get_dummies’ method introduced in a previous workshop to fix this.

Create two lists:

- ‘categorical_features’ : a list of all of the categorical features
- ‘prefixes’ : The corresponding prefix for the new, one-hot features

Zip these two together into a dictionary called ‘prefix_dict’

 Your code here

This obviously creates a lot more features .. 76 in total:

 Your code here

```
credit_data.info() =  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 987 entries, 0 to 999  
Data columns (total 62 columns):  
 #   Column      Non-Null Count  Dtype     
---  --          --          --          --  
 0   dr          987 non-null    int64
```

1	ca	987	non-null	float64
2	pd	987	non-null	int64
3	rs	987	non-null	int64
4	age	987	non-null	float64
5	nc	987	non-null	int64
6	dp	987	non-null	int64
7	default	987	non-null	int64
8	cs_A11	987	non-null	uint8
9	cs_A12	987	non-null	uint8
10	cs_A13	987	non-null	uint8
11	cs_A14	987	non-null	uint8
12	ch_A30	987	non-null	uint8
13	ch_A31	987	non-null	uint8
14	ch_A32	987	non-null	uint8
15	ch_A33	987	non-null	uint8
16	ch_A34	987	non-null	uint8
17	pr_A40	987	non-null	uint8
18	pr_A41	987	non-null	uint8
19	pr_A410	987	non-null	uint8
20	pr_A42	987	non-null	uint8
21	pr_A43	987	non-null	uint8
22	pr_A44	987	non-null	uint8
23	pr_A45	987	non-null	uint8
24	pr_A46	987	non-null	uint8
25	pr_A48	987	non-null	uint8
26	pr_A49	987	non-null	uint8
27	sa_A61	987	non-null	uint8
28	sa_A62	987	non-null	uint8
29	sa_A63	987	non-null	uint8
30	sa_A64	987	non-null	uint8
31	sa_A65	987	non-null	uint8
32	ed_A71	987	non-null	uint8
33	ed_A72	987	non-null	uint8
34	ed_A73	987	non-null	uint8
35	ed_A74	987	non-null	uint8
36	ed_A75	987	non-null	uint8
37	gm_A91	987	non-null	uint8
38	gm_A92	987	non-null	uint8
39	gm_A93	987	non-null	uint8
40	gm_A94	987	non-null	uint8
41	od_A101	987	non-null	uint8
42	od_A102	987	non-null	uint8
43	od_A103	987	non-null	uint8
44	op_A141	987	non-null	uint8
45	op_A142	987	non-null	uint8
46	op_A143	987	non-null	uint8

```

47 hs_A151    987 non-null    uint8
48 hs_A152    987 non-null    uint8
49 hs_A153    987 non-null    uint8
50 job_A171   987 non-null    uint8
51 job_A172   987 non-null    uint8
52 job_A173   987 non-null    uint8
53 job_A174   987 non-null    uint8
54 tp_A191    987 non-null    uint8
55 tp_A192    987 non-null    uint8
56 fw_A201    987 non-null    uint8
57 fw_A202    987 non-null    uint8
58 pp_A121    987 non-null    uint8
59 pp_A122    987 non-null    uint8
60 pp_A123    987 non-null    uint8
61 pp_A124    987 non-null    uint8
dtypes: float64(2), int64(6), uint8(54)
memory usage: 121.4 KB

```

At this point you can display the whole dataframe in a single cell. As previously, I have truncated my display somewhat so that it fits the workbook format.

 Your code here

	dr	ca	pd	rs	age	nc	dp	default	cs_A11	cs_A12	cs_A13
0	6	1169.0	4	4	67.000000	2	1	0	1	0	0
1	48	5951.0	2	2	22.000000	1	1	1	0	1	0
2	12	2096.0	2	3	49.000000	1	2	0	0	0	0
3	42	7882.0	2	4	35.580943	1	2	0	1	0	0
4	24	4870.0	3	4	53.000000	2	2	1	1	0	0

3.7 Step 2 Select the Algorithm

The first algorithm we will be using for this classification problem will be Logistic Regression .. so we can go ahead and build the model..

3.8 Step 3 Build the Model

In this section we be using the logistic regression classifier provided by sklearn. Initially we will build a demonstration example of model building using all of the data in the data-set. In the next section we will split the data into two parts to enable testing of the model.

Add a cell with code to split the data into two parts. use ‘X’ to represent the known ‘inputs’ to the model. Set ‘Y’ to represent the ‘label’ or known (expected) output from the model.

💡 Your code here

Those new dataframes have the following shapes:

💡 Your code here

```
X.shape = (987, 61)
```

```
Y.shape = (987,)
```

That is, ‘Y’ is a single dimensional vector of 1000 elements.

Create an instance of the specific modelling algorithm called ‘logModel’

💡 Tutor provided code:

```
logModel = LogisticRegression(solver='liblinear')
```

Then use logModel.fit(X,Y) to build the model:

💡 Your code here

Print the ‘score’ attribute from the logModel to show that the model has ‘worked’.

Note This is not really legitimate since it scores (provides one quality measure of) the model - but it does so using exactly the same data that the original model was built from. This would be a bit like setting a student an exam consisting of questions they had already practiced in class.

However, for now it provides an indication that we have, at least, built a model!

💡 Your code here

```
logModel.score(X,Y) = 0.785
```

3.9 Step 4 Check Model Quality

In practice, models will always require a quality check. A common way of achieving this is to split the original data into two parts. One part will be used for building the model, the other part will be used to test how good it is at making predictions. There are various strategies regarding how to make this split - some of which we will cover in later sessions. However, a common and easy way to split data is simply to make a random selection of around 20% of the data and to retain this for testing. The remaining 80% can be used for testing.

Sklearn provides a simple method for random splitting of data called ‘train_test_split’:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Use this function to split the data into two parts in the ratio 80%:20%:

💡 Your code here

Then print the first few rows of the test data-set.

💡 Your code here

```
y_test[0:20] =  
346    0  
506    0  
518    0  
716    0  
689    0  
928    0  
130    0  
416    1  
939    0  
982    0  
83     0  
539    0  
423    0  
734    0  
583    1  
591    0  
427    0  
765    0  
90     0  
855    0  
Name: default, dtype: int64
```

Now we can build a model as above. However this time based only on the training data not the test data.

💡 Tutor provided code

```
logModel = logModel.fit(X_train,y_train)
```

We then want to test this model. That means generating a set of predictions based on the **test** data. To do this you will need to pass the test data you generated to the ‘logModel.predict’ method:

💡 Tutor provided code

```
predictions = logModel.predict(X_test)
```

```
predictions =
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1
 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 1 1 1 1 0 1 0 0 1 0 1 0
 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 0 0
 0 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
 0 0 1 0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 1 0 0]
```

💡 Your code here

Now we are in a position to get a real ‘score’ for the model, this time based on data the model has not seen before. The ‘score’ method of logModel what fraction of the time did the model predict the correct answer.

There are two things to note at this point.

1. ‘score’ is only one of several quality measures for classifiers and on its own can be misleading. So you will generally need to employ other metric.
2. Because the data we used is split randomly every time the model is built .. you may get different results to the ones shown below.

Add a line of code to print the ‘score’ for the model based on the test data obtained above.

```
LogModel.Score = 0.727
```

A more useful tool for measuring quality is the ‘confusion matrix’. The confusion matrix will be described in the lecture portion of this course.

You can access this tool by importing the ‘confusion_matrix’ method from sklearn.metrics.

 Tutor provided code

```
cm = confusion_matrix(y_test, predictions)
print (f"Confusion matrix =\n {cm}")
```

```
Confusion matrix =
[[119  27]
 [ 27  25]]
```

This output might be useful to a person who is familiar with the confusion matrix but in this ‘vanilla’ form it is somewhat difficult to interpret. In most cases you will want to ‘wrap’ this in a more graphically attractive output format and this can be achieved using a seaborn heatmap:

 Tutor provided code

```
import seaborn as sns

plt.figure(figsize=(4, 4))
sns.set(font_scale=1.2)
sns.heatmap(cm, annot=True,
            fmt="d",
            cmap="Blues",
            annot_kws={"size": 14},
            cbar=False,
            xticklabels=["Class 0", "Class 1"],
            yticklabels=["Class 0", "Class 1"])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

Alternatively, you might use decide to use the ‘classification_report’ method from the ‘sklearn.metrics’ library to provide another view of the quality of your classification:

- Import ‘sklearn.metrics’
- Define the names of each classification in a list
- Apply the sklearn ‘classification_report’ method - passing ‘y_test’, ‘predictions’ and the list of classification names as parameters

 Your code here

precision	recall	f1-score	support
-----------	--------	----------	---------

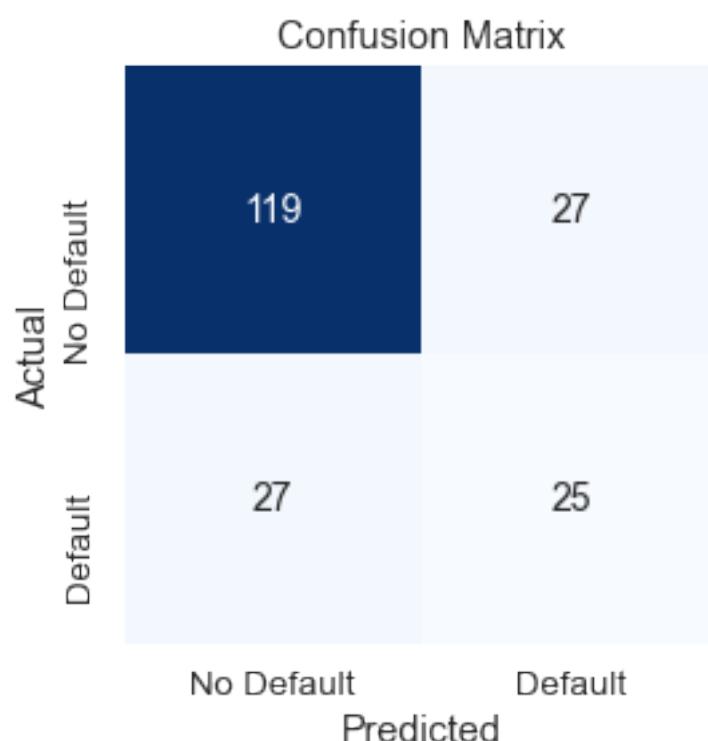


Figure 4: Confusion Matrix display using Seaborn library

No Default	0.8151	0.8151	0.8151	146
Default	0.4808	0.4808	0.4808	52
accuracy			0.7273	198
macro avg	0.6479	0.6479	0.6479	198
weighted avg	0.7273	0.7273	0.7273	198

3.10 Step 5: Build the model into an application

In the case of Logistic Regression the ‘model’ is just a list of coefficients for an equation. You can access the values of these coefficients using the ‘.intercept_’ and ‘.coef_’ attributes of logModel.

💡 Your code here

```
logModel.intercept_ = [-0.57043552]
logModel.coef_ =
[[ 2.96963799e-02  9.78206831e-05  3.25073099e-01 -1.35976489e-02
-1.66025222e-02  2.03618417e-01 -1.36636145e-02  6.64123486e-01
2.21666456e-01 -3.98474170e-01 -1.05775129e+00  1.52644730e-01
7.12528593e-01 -2.08098376e-01 -2.39117840e-01 -9.88392624e-01
5.63951759e-01 -9.52466949e-01 -2.33515281e-01 -8.82043146e-03
-4.44256181e-01 -1.03040568e-02 -7.57096555e-02  6.18237798e-01
2.19731466e-02 -4.95256665e-02  3.42454303e-01  2.38431412e-01
-1.46325870e-01 -5.53013147e-01 -4.51982215e-01 -1.94413980e-01
1.22338372e-02  8.59743685e-02 -5.93447479e-01  1.19217736e-01
2.68234359e-01 -4.65075788e-02 -6.28811150e-01 -1.63351148e-01
-6.68301238e-02  2.74771749e-01 -7.78377142e-01 -2.72460554e-03
6.82536551e-02 -6.35964567e-01 -1.77540975e-02 -2.26352845e-01
-3.26328574e-01 -4.86130964e-01 -7.33270971e-02 -6.07273142e-02
4.97498578e-02 -1.42900808e-01 -4.27534709e-01  1.83914328e-01
-7.54349845e-01 -3.41156746e-01 -1.56548024e-01 -2.52315575e-01
1.79584828e-01]]
```

It will be practically useful when building our application to have a record of each of the feature names and their corresponding indices (column numbers).

Write code to iterate through the ‘X_train’ dataframe. For each column in the dataframe create one element of the dictionary composed of the feature name and the column number.

Hint: the Python ‘enumerate’ function is useful here. ‘Enumerate’ is an iterable object which, when given a list returns pairs of values - an index (position) in the list, and the item at that position.

To aid re-use of this information, format the output from the loop so that it can be directly copy-pasted into the planned ‘Credit Evaluation’ application. This information could easily be written to a file for later use .. but in this case we are simply going to copy-paste the resulting string into our application code.

💡 Your code here

```
Feature_dict = {  
    'dr':0, 'ca':1, 'pd':2, 'rs':3, 'age':4, 'nc':5, 'dp':6, 'cs_A11':7, 'cs_A12':8, 'cs_A13':9, 'cs_A14':10, 'cs_A15':11, 'cs_A16':12, 'cs_A17':13, 'cs_A18':14, 'cs_A19':15, 'cs_A20':16, 'cs_A21':17, 'cs_A22':18, 'cs_A23':19, 'cs_A24':20, 'cs_A25':21, 'cs_A26':22, 'cs_A27':23, 'cs_A28':24, 'cs_A29':25, 'cs_A30':26, 'cs_A31':27, 'cs_A32':28, 'cs_A33':29, 'cs_A34':30, 'cs_A35':31, 'cs_A36':32, 'cs_A37':33, 'cs_A38':34, 'cs_A39':35, 'cs_A40':36, 'cs_A41':37, 'cs_A42':38, 'cs_A43':39, 'cs_A44':40, 'cs_A45':41, 'cs_A46':42, 'cs_A47':43, 'cs_A48':44, 'cs_A49':45, 'cs_A50':46, 'cs_A51':47, 'cs_A52':48, 'cs_A53':49, 'cs_A54':50, 'cs_A55':51, 'cs_A56':52, 'cs_A57':53, 'cs_A58':54, 'cs_A59':55, 'cs_A60':56, 'cs_A61':57}
```

💡 Your code here

```
Len feature dict = 61
```

Rather than build our application within this workbook it is more natural to save the model parameters at this point, then have the application load these parameters when required. The following code will enable the model to be saved as a ‘pickle’ file.

💡 Tutor provided code

```
model_filename = 'germ_cred_model.pkl'  
# Open the file to save as pkl file  
the_file = open(model_filename, 'wb')  
pickle.dump(logModel, the_file)  
# Close the pickle instances  
the_file.close()
```

Chapter 4 - Polynomial Regression

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2024-07-12

Table of contents

4	Polynomial Regression	4 - 3
4.1	Introduction	4 - 3
4.2	Instructions for Students	4 - 3
4.3	Import Libraries	4 - 4
4.4	Load the Data	4 - 4
4.5	Review the data quantitatively	4 - 6
4.6	Review the data visually	4 - 7
4.7	Selecting the data we wish to use for modelling	4 - 12
4.8	Splitting data for training and testing	4 - 13
4.9	First model - linear regression	4 - 14
4.9.1	Build the linear regression model	4 - 14
4.9.2	Test the linear regression model	4 - 15
4.10	Second model - Polynomial regression	4 - 16

List of Figures

1	Master Artisan 'Learnists' Fitting a Complex Mathematical Curve (c. 1590)	4 - 2
2	Scatter matrix for the complete Boston data-set	4 - 8
3	Using .iloc to select columns 4 to 7 to chart as a scatter matrix	4 - 9
4	Correlation matrix for the data-set	4 - 11

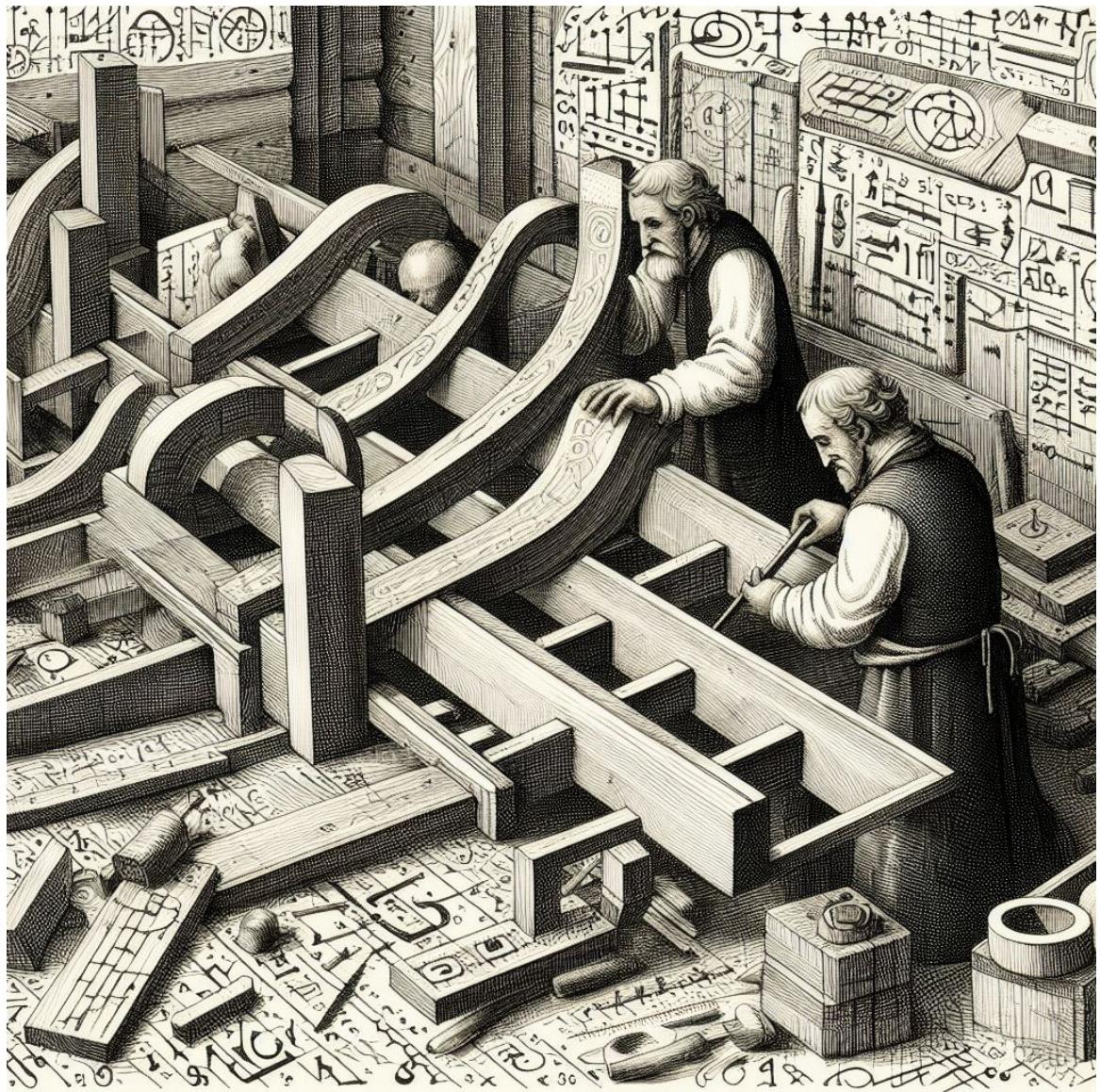


Figure 1: Master Artisan 'Learnists' Fitting a Complex Mathematical Curve (c. 1590)

4 Polynomial Regression

4.1 Introduction

In this workshop we will extend our previous work on Regression. In this session we will review ‘polynomial regression’ - that is, functions of the general form:

$$y = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_1^2 + \theta_3 \cdot x_1^3 + \dots + \theta_n \cdot x_1^n$$

where, as previously we set: $x_0 = 1$

The above general equation is a ‘polynomial’ (it has squared, cubed and fourth-power terms etc.) but it is only in one dimension. Or, to put it another way, in a data-set we were using to build the model, there would only be one feature (that is, one column). To make polynomial models even more general, they need to operate across multiple dimensions. Not only that, but they need to be capable of representing ‘interactions’ between dimensions.

Thus, for a second-order, polynomial model in two dimensions x_1 and x_2 we would see:

$$y = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_1^2 + \theta_4 \cdot x_2^2 + \theta_5 \cdot x_1 \cdot x_2$$

Where the final ‘ $\theta_5 \cdot x_1 \cdot x_2$ ’ term enables us to model any interaction between x_1 and x_2 .

The same logic holds in higher dimensions, and thus you should expect to see a model based on not only powers of each feature, but also terms relating to all permutations of features! This may sound horribly complicated! How on earth would somebody build a model with all of those complicated terms?

Inevitably, there is an easy answer. The answer, in some ways feels like a ‘trick’ to me - but is actually quite obvious when you see it. Not only that, but library functions within the Machine Learning libraries enable this type of model to be built quite easily.

The data-set used in this workshop is the “Boston House Price” data. This is a popular data-set for Machine Learning tutorials. It is widely available on the Internet and also built into sklean as one of its demonstration data-sets.

4.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

4.3 Import Libraries

As always we start by import the required libraries. In this case, those are:

- numpy : Conventionally named as ‘np’
- pandas : Conventionally named as ‘pd’
- matplotlib.pyplot : Conventionally named as plt
- seaborn: Conventionally named as sns
- Axes3D : Imported from mpl_toolkits.mplot3d
- train_test_split : Imported from sklearn.model_selection
- LinearRegression : Imported from sklearn.linear_model
- mean_squared_error : Imported from sklearn.metrics
- r2_score : Imported from sklearn.metrics
- PolynomialFeatures : Imported from sklearn.preprocessing

 Your code here

4.4 Load the Data

Add code to load then display the ‘boston house price’ data set from the provided data file ‘boston.csv’

 Your code here

```
boston.iloc[0:20, 0:7] =
      CRIM    ZN  INDUS   CHAS    NOX     RM    AGE
0  0.00632  18.0   2.31      0  0.538  6.575  65.2
1  0.02731   0.0   7.07      0  0.469  6.421  78.9
2  0.02729   0.0   7.07      0  0.469  7.185  61.1
3  0.03237   0.0   2.18      0  0.458  6.998  45.8
4  0.06905   0.0   2.18      0  0.458  7.147  54.2
5  0.02985   0.0   2.18      0  0.458  6.430  58.7
6  0.08829  12.5   7.87      0  0.524  6.012  66.6
7  0.14455  12.5   7.87      0  0.524  6.172  96.1
8  0.21124  12.5   7.87      0  0.524  5.631 100.0
9  0.17004  12.5   7.87      0  0.524  6.004  85.9
10 0.22489  12.5   7.87      0  0.524  6.377  94.3
11 0.11747  12.5   7.87      0  0.524  6.009  82.9
12 0.09378  12.5   7.87      0  0.524  5.889  39.0
13 0.62976   0.0   8.14      0  0.538  5.949  61.8
14 0.63796   0.0   8.14      0  0.538  6.096  84.5
15 0.62739   0.0   8.14      0  0.538  5.834  56.5
16 1.05393   0.0   8.14      0  0.538  5.935  29.3
17 0.78420   0.0   8.14      0  0.538  5.990  81.7
18 0.80271   0.0   8.14      0  0.538  5.456  36.6
19 0.72580   0.0   8.14      0  0.538  5.727  69.5
```

```
boston.iloc[0:20, 8:] =
      RAD    TAX PTRATIO      B LSTAT MEDV
0     1    296    15.3  396.90   4.98  24.0
1     2    242    17.8  396.90   9.14  21.6
2     2    242    17.8  392.83   4.03  34.7
3     3    222    18.7  394.63   2.94  33.4
4     3    222    18.7  396.90   5.33  36.2
5     3    222    18.7  394.12   5.21  28.7
6     5    311    15.2  395.60  12.43  22.9
7     5    311    15.2  396.90  19.15  27.1
8     5    311    15.2  386.63  29.93  16.5
9     5    311    15.2  386.71  17.10  18.9
10    5    311    15.2  392.52  20.45  15.0
11    5    311    15.2  396.90  13.27  18.9
12    5    311    15.2  390.50  15.71  21.7
13    4    307    21.0  396.90   8.26  20.4
14    4    307    21.0  380.02  10.26  18.2
15    4    307    21.0  395.62   8.47  19.9
16    4    307    21.0  386.85   6.58  23.1
17    4    307    21.0  386.75  14.67  17.5
18    4    307    21.0  288.99  11.69  20.2
19    4    307    21.0  390.95  11.28  18.2
```

4.5 Review the data quantitatively

It is often useful to review a brief summary of the data. The `.describe()` function of Pandas provides summary data for the dataframe, including:

- count : The number of items in the dataframe
- mean : The mean or ‘average’ value .. sum of values / number of values
- std : Standard deviation - a measure of spread in the data
- min : The smallest value
- 25%, 50%, 75% : Quartile values
- max : The largest value

Use the `.describe()` function to provide a summary of the data.

Note : In the following I spread the display over two cells so that they format correctly for this workbook. You may prefer simply to include the name of the dataframe as the last line in the cell - in this case you should obtain a scrollable full-width display of the data. If you wish to do as I have done, then you will need to use the pandas ‘`iloc[]`’ method to select a sub-set of the data-frame for display.

💡 Your code here

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

	RAD	TAX	PTRATIO	B	LSTAT	MEDV
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	9.549407	408.237154	18.455534	356.674032	12.653063	22.532806
std	8.707259	168.537116	2.164946	91.294864	7.141062	9.197104
min	1.000000	187.000000	12.600000	0.320000	1.730000	5.000000
25%	4.000000	279.000000	17.400000	375.377500	6.950000	17.025000
50%	5.000000	330.000000	19.050000	391.440000	11.360000	21.200000
75%	24.000000	666.000000	20.200000	396.225000	16.955000	25.000000
max	24.000000	711.000000	22.000000	396.900000	37.970000	50.000000

As in previous workshops, we should review the data to see if there are any missing values. In this case, we introduce a new and useful function that reviews each column and indicates if it has any null elements:

 Tutor provided code

```
pd.isnull(boston).any()
```

```
CRIM      False
ZN        False
INDUS     False
CHAS      False
NOX       False
RM        False
AGE       False
DIS       False
RAD       False
TAX       False
PTRATIO   False
B         False
LSTAT     False
MEDV     False
dtype: bool
```

4.6 Review the data visually

We can start with the Pandas ‘scatter_matrix’ function we have used before to review all of the features and the correlations between them:

 Your code here

This chart is rather large and complex to read .. therefore it may be better to review only part the data-set.

Use the ‘iloc’ method to select various sub-sets of the overall data-frame.

 Your code here

 Your code here

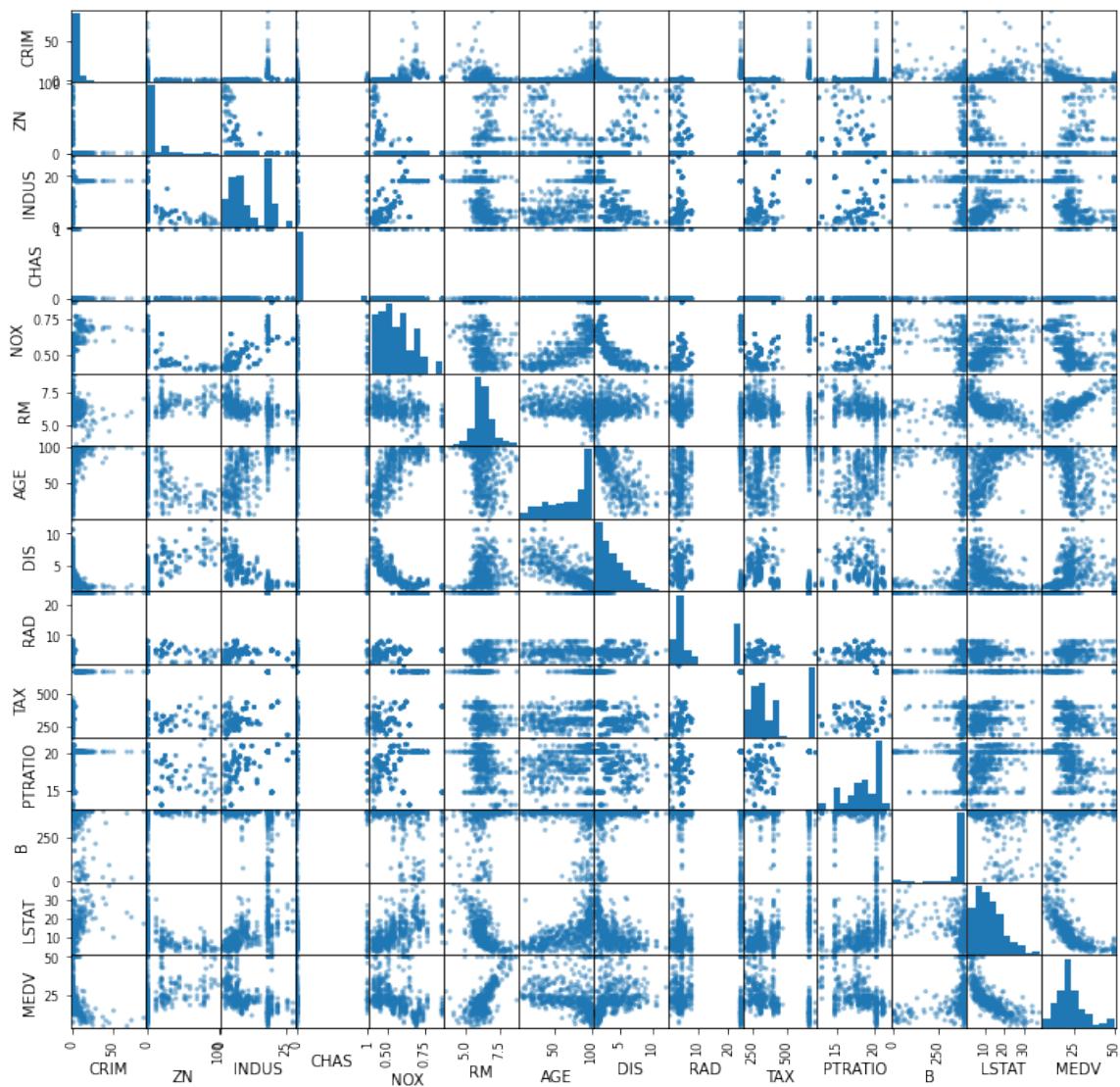


Figure 2: Scatter matrix for the complete Boston data-set

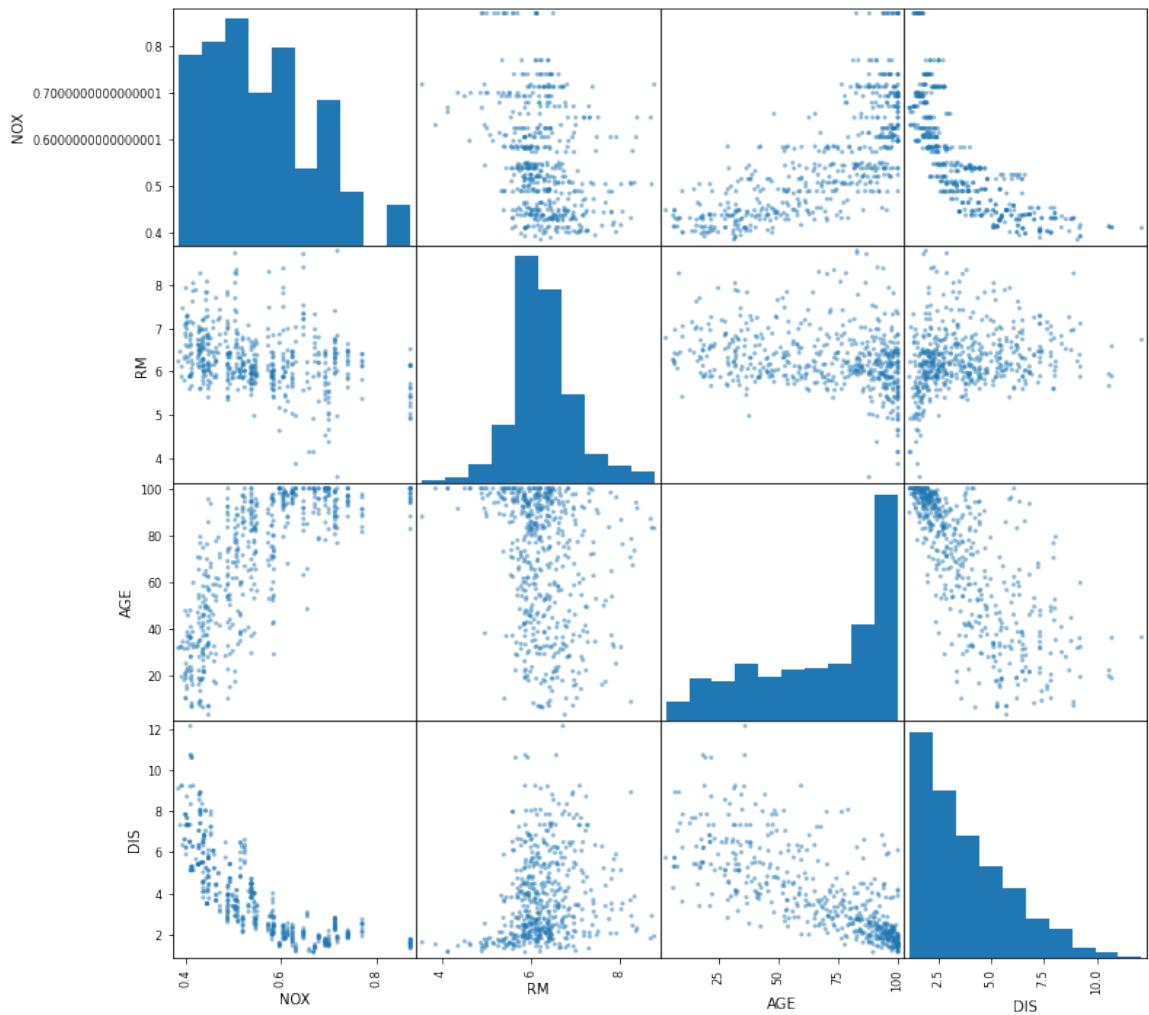


Figure 3: Using .iloc to select columns 4 to 7 to chart as a scatter matrix

This workshop is focussed on regression. We need to determine which of the features can be used to predict house prices. A useful clue the correlation between the variables. In particular, which of the variables is correlated with the MEDV label?

Add code to display a correlation matrix for the data-set. This chart is clearer if the absolute value of each correlation is plotted. This is because, regardless of whether there is a negative or positive correlation .. a stronger correlation will provide a better basis for modelling.

- Compute the correlation of the dataset using the ‘corr()’ function.
- Compute the absolute value for each element in this matrix using the numpy ‘absolute’ function
- Plot this correlation using the seaborn ‘heatmap()’ method.

 Your code here

You can experiment with the way that the heat map is displayed. The ‘cmap’ option in the ‘heatmap’ function defines the colours that are used for the plot. Many different sets of colours are available and these are documented here: Other colour map options at https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html. Experiment with some of the other possible cmap pallets and decide which one you prefer.

From this correlation map, we see that: - MEDV is strongly correlated with LSTAT and RM - RAD and TAX are correlated with each other Therefore, LSTAT and RM are good candidates for building our model, whereas we should not use both RAD and TAX since we will be ‘double counting’ their impact on the result

It is interesting to review those two key features (LSTAT and RM) in a 3D plot. As in previous practical classes, 3D plots can be generated using the same code as was presented in the workshop on clustering customer data.

Set the vertical dimension to represent MEDV.

I have used the command

```
%matplotlib inline
```

So that the chart displays neatly in this workbook. However, you should use the command:

```
%matplotlib auto
```

To create a ‘pop-out’ 3D window that can be rotated and zoomed.

In the previous activities the colour of the points was set based on the cluster number. In this case, base the point colour on the value of MEDV. This provides another visual clue regarding the Mean Value of the Housing.

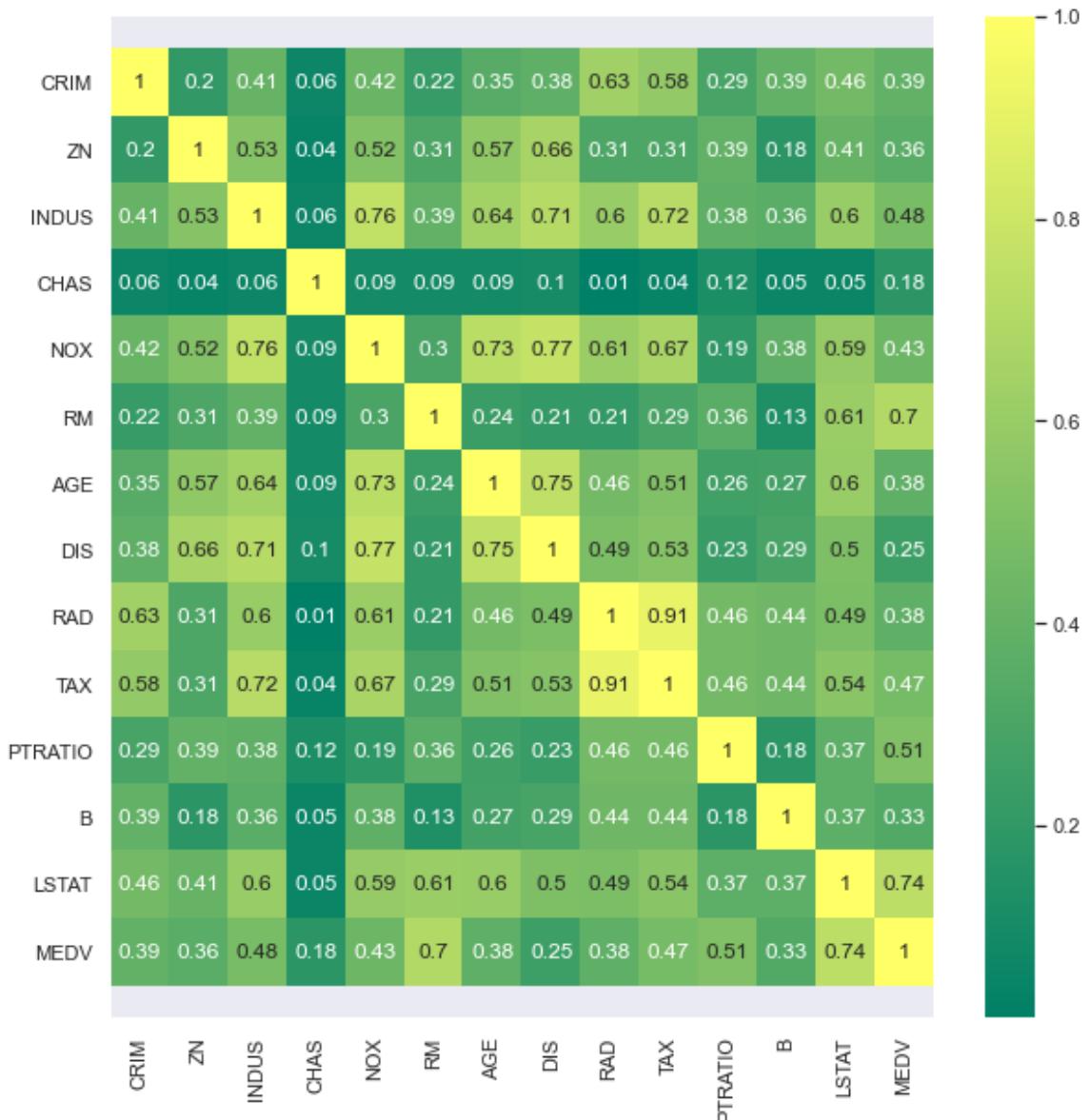
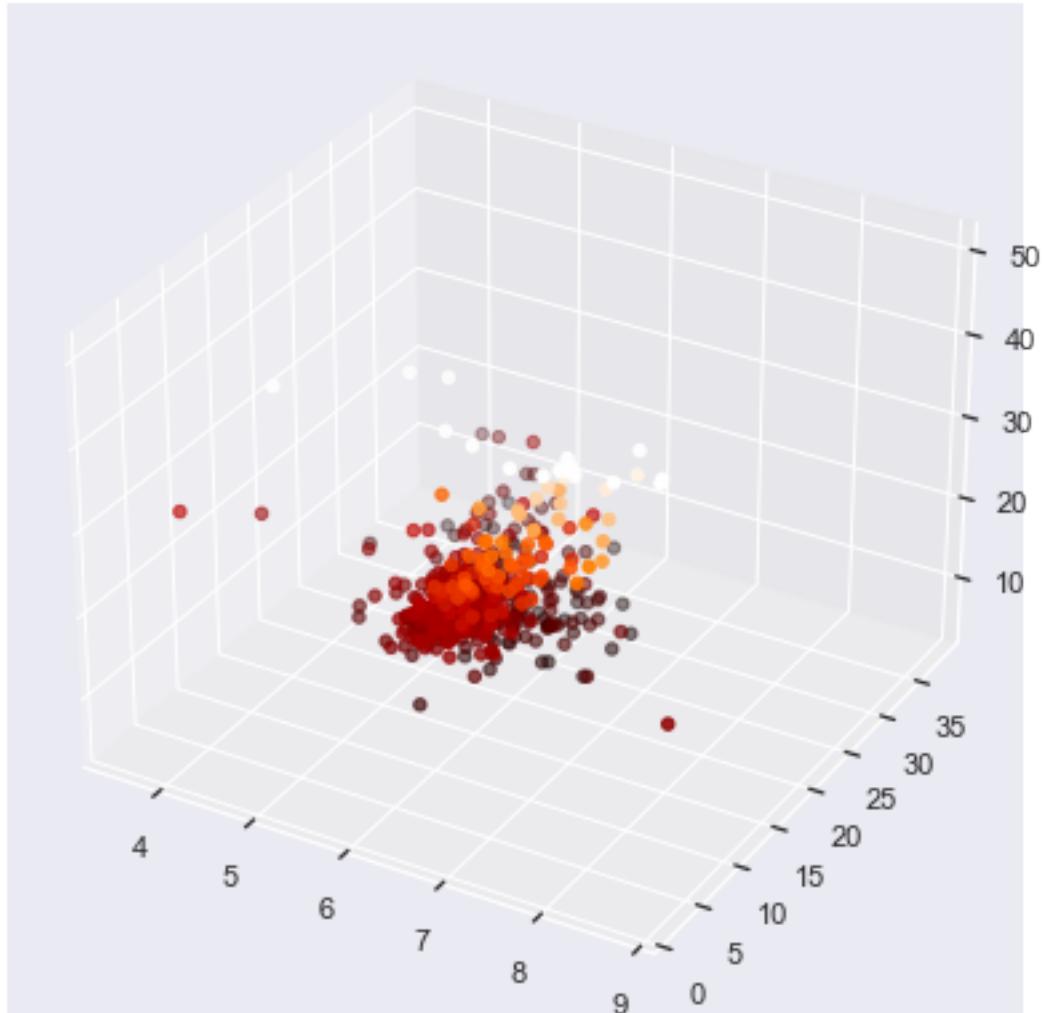


Figure 4: Correlation matrix for the data-set

Rotate the 3D view and look at the ‘shape’ of the data. This is the thing we are going to try to model. It is clearly multi-dimensional and also appears to be non-linear. It will thus require a polynomial model.

Experiment with the cmap values to create a visualisation that you prefer.

💡 Your code here



4.7 Selecting the data we wish to use for modelling

Add code to create a sub-set of the overall dataframe by copying only the features ‘LSTAT’ and ‘RM’ to a new dataframe. Call that dataframe ’X’

💡 Your code here

```
X.iloc[0:20,:] =  
    LSTAT      RM  
0    4.98  6.575  
1    9.14  6.421  
2    4.03  7.185  
3    2.94  6.998  
4    5.33  7.147  
5    5.21  6.430  
6   12.43  6.012  
7   19.15  6.172  
8   29.93  5.631  
9   17.10  6.004  
10  20.45  6.377  
11  13.27  6.009  
12  15.71  5.889  
13  8.26  5.949  
14  10.26  6.096  
15  8.47  5.834  
16  6.58  5.935  
17  14.67  5.990  
18  11.69  5.456  
19  11.28  5.727
```

4.8 Splitting data for training and testing

We want to be able to test how good our model is for prediction. In particular, we need to be assured that it is not over-fitting the data.

Add code to split the model into training and a testing sub-sets using sklearn's 'train_test_split' method.

Note Because this function creates a random selection, you are likely to see different rows selected for your train and test sets.

💡 Your code here

```
X_train[0:5] =  
    LSTAT      RM  
442  16.59  6.219  
314  9.28  6.567
```

```

473  11.66  6.980
350   5.98   6.490
306   6.47   7.420

X_test[0:5] =
    LSTAT      RM
238   6.36   6.481
22    18.72  6.142
200   4.45   7.135
230   11.65  5.981
356   17.60  6.212

Y_train[0:5] =
442    18.4
314    23.8
473    29.8
350    22.9
306    33.4
Name: MEDV, dtype: float64

Y_test[0:5] =
238    23.7
22     15.2
200   32.9
230   24.3
356   17.8
Name: MEDV, dtype: float64

```

4.9 First model - linear regression

To give us a baseline to judge our polynomial regression model we first generate a linear model and score it for model quality

4.9.1 Build the linear regression model

Building a linear model is trivial using sklearn:

- Instantiate a model object based on the ‘LinearRegression()’ class
- Call the ‘.fit’ method of that object, passing as parameters the X_train and Y_train data

 Your code here

```
LinearRegression()
```

4.9.2 Test the linear regression model

The sklearn library provides a range of functions for scoring different types of models:

https://scikit-learn.org/stable/modules/model_evaluation.html

Two useful and common quality measure for regression are:

- The R^2 score or ‘Coefficient of Determination’
 - (See https://en.wikipedia.org/wiki/Coefficient_of_determination)
- The ‘root mean squared error’ or RMSE

Both of these can easily be computed using sklearn functions as follows.

It is interesting to compare how well the model fitted to the training data as compared with the performance of the model on test data (which the model has not ‘seen’). So we will compute these scores first for the training data and then for the test data:

 Tutor provided code

```
y_train_predict = lin_model.predict(X_train)
r2_train = r2_score(Y_train, y_train_predict)
rmse_train = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
```

Now do the same thing for the test data:

 Your code here

And print the results for comparison:

 Your code here

R2:

```
Train = 0.62
Test = 0.69
```

RMSE:

```
Train = 5.53
Test = 5.51
```

As we might expect, the model fits the training data somewhat better than it performs when seeing new data. That is, for the training set the R2 value is closer to 1 (better) and the RMSE is lower (better).

4.10 Second model - Polynomial regression

Now comes the ‘trick’ that enables us to easily generate a polynomial model for given data.

sklearn provides a function that generates an expanded data-set. It creates new features that are combinations of the other features. So that if we have a feature ‘x’ and we use the function to generate Polynomial Features of degree 3, then the function will add new data columns (features) for x^2 and x^3 .

Even better, if we have two features .. say ‘x’ and ‘z’, and we use the function to generate Polynomial Features, for example, order 2, then it will generate new features for x^2 , z^2 and $x.z$ (the last of these being the interaction term described in the first section above).

So let’s experiment with the sklearn.PolynomialFeatures function and generate an expanded data-set for our housing price variables in X_train.

We first generate the features and take a look at the extended set of feature names:

 Tutor provided code

```
poly_features = PolynomialFeatures(degree=5)
X_train_poly = poly_features.fit_transform(X_train)
print(poly_features.get_feature_names_out(["LSTAT","RM"]))
```

```
['1' 'LSTAT' 'RM' 'LSTAT^2' 'LSTAT RM' 'RM^2' 'LSTAT^3' 'LSTAT^2 RM'
'LSTAT RM^2' 'RM^3' 'LSTAT^4' 'LSTAT^3 RM' 'LSTAT^2 RM^2' 'LSTAT RM^3'
'RM^4' 'LSTAT^5' 'LSTAT^4 RM' 'LSTAT^3 RM^2' 'LSTAT^2 RM^3' 'LSTAT RM^4'
'RM^5']
```

We can also look at the actual data that has been produced:

Note In the following I restricted the display of values to using 2 decimal points using the following command:

```
np.set_printoptions(precision=2, suppress=True)
```

 Your code here

```
X_train_poly[0:5] =
[[    1.00      16.59       6.22     275.23      103.17      38.68
   4566.03    1711.64     641.63     240.53    75750.51   28396.17
  10644.71    3990.32   1495.83  1256700.91   471092.40   176595.76
  66199.46   24815.82    9302.57]
 [    1.00      9.28       6.57      86.12      60.94      43.13
```

```

    799.18      565.54      400.20      283.21      7416.38      5248.21
    3713.90     2628.14     1859.81    68824.00     48703.36     34464.97
    24389.17    17259.02    12213.36]
[   1.00      11.66      6.98      135.96      81.39      48.72
  1585.24     948.97      568.08      340.07     18483.93     11064.99
  6623.81     3965.20     2373.68    215522.57    129017.80     77233.64
  46234.20    27677.08    16568.27]
[   1.00      5.98      6.49      35.76      38.81      42.12
  213.85     232.08     251.88      273.36     1278.81     1387.87
  1506.23    1634.69     1774.10     7647.26     8299.45     9007.27
  9775.44    10609.13    11513.93]
[   1.00      6.47      7.42      41.86      48.01      55.06
  270.84     310.61     356.21      408.52     1752.33     2009.63
  2304.71    2643.11    3031.21    11337.61    13002.33    14911.48
 17100.95   19611.91   22491.56]]

```

Notice that, rather conveniently, the `PolynomialFeatures` function has also added a column of '1's to the data. This follows the pattern introduced in the last session for dealing with the constant term.

Also, notice that the coefficients are generally larger for the higher-power parameters. In a later section we will return to this as it is indicative of a problem that can occur with higher-order models. For now, just think about the relative size of the parameters and what issues that might cause.

Having generated an extended set of polynomial features, we can now simply apply a linear regression model to the new extended set! This effectively gives us terms representing polynomial versions of the features and hence is a polynomial model.

The beauty of this is that we can just re-use our linear model fitting tools.

 Tutor provided code

```

polynomial_model = LinearRegression()
polynomial_model.fit(X_train_poly, Y_train)
print("Model coefficients  = ", polynomial_model.coef_)
print("Constant term (bias) = ", polynomial_model.intercept_)

```

```

Model coefficients  =
[   0.00   -96.10  -2585.45      5.56      44.82     840.17     -0.25     -1.16
  -9.17  -135.64       0.01      0.03      0.11      0.87     10.88     -0.00
  -0.00    -0.00     -0.01     -0.03     -0.35]
Constant term (bias) = 3200.27

```

As previously, we can apply this model to both the original training data and to the test data:

 Tutor provided code

```
y_train_predicted = polynomial_model.predict(X_train_poly)
y_test_predicted = polynomial_model.predict(poly_features.fit_transform(X_test))
```

Then measure the model quality for each of these cases:

R2:

```
Train = 0.79
Test = 0.75
```

RMSE:

```
Train = 4.16
Test = 4.93
```

Notice that both the R2 and RMSE values are better in the case of the polynomial model than they were above in the linear model.

But there is also some evidence here that the model is moving towards over-fitting the data. The two scores are very much improved for the training data (they indicate a good fit) but only improved by a small amount for the testing data.

In other words, the second-order polynomial model is better than the original model - but not as much better as we might think without careful testing with new data!

Experiment by generating models of higher-degree polynomials. That is, try substituting different values for the ‘degree’ in the line:

```
poly_features = PolynomialFeatures(degree=2)
```

Observe the behaviour and determine at which degree the model starts to over-fit the data. That is, the polynomial degree at which the model actually starts to perform less well with new test data.

Chapter 5 - matrices and Linear Algebra

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2024-10-12

Table of contents

5 matrices and Linear Algebra	5 - 3
5.1 Introduction	5 - 3
5.2 Instructions for Students	5 - 3
5.3 Two Libraries for Linear Algebra - Sympy and Numpy	5 - 5
5.4 Algebraic Equations represented in matrices	5 - 5
5.5 matrices using the Sympy Library	5 - 5
5.5.1 Defining matrices in Sympy	5 - 5
5.5.2 Matrix Addition and Subtraction	5 - 6
5.5.3 Matrix multiplication by a constant	5 - 6
5.5.4 Multiplying pairs of matrices	5 - 7
5.5.5 Dot-Product (Inner Product) of Vectors	5 - 8
5.5.6 Cross-product of Vectors	5 - 8
5.5.7 Vector Norm - Length of a Vector	5 - 9
5.5.8 Determinant and Inverse of a Square Matrix	5 - 9
5.5.9 Other Matrix Operations from Linear Algebra	5 - 11
5.6 matrices in Numpy	5 - 13
5.6.1 Matrix Addition	5 - 14
5.6.2 Matrix Subtraction	5 - 15
5.6.3 Matrix multiplication by a Constant	5 - 15
5.6.4 Dot-product (Sometimes called ‘inner product’)	5 - 16
5.6.5 Matrix Division by a constant	5 - 17
5.6.6 Determinant of a matrix	5 - 18
5.6.7 Inverse of a matrix	5 - 18
5.6.8 Solutions to Linear Equations	5 - 19
5.6.9 Matrix (or vector) Norm	5 - 20

List of Figures

- 1 The 1853 attempt to solve ‘the matrix problem’. The so called ‘Steam Rivulet Engine’ was capable of solving large-scale, matrix calculations but unfortunately suffered from frequent gasket blow-outs and ultimately proved to be too dangerous for common use 5 - 4

5 matrices and Linear Algebra

5.1 Introduction

This Machine Learning course is intended to be generally non-mathematical. Nevertheless, no advanced study of Machine Learning can be realistically considered complete without a good grounding in Probability and Linear Algebra.

For students who are less familiar with these subjects I recommend the following as useful references:

Strang, G. (2005) "Linear Algebra and its Applications", TBS

Walpole, R.E., Myers, R.H., Myers, S.L. and Ye, K. (2012) "Probability and Statistics for Engineers and Scientists", Prentice Hall

Deisenroth, M.P., Faisal, A.A. and Ong, C.S (2021) "Mathematics for Machine Learning", Cambridge University Press

For many years Prof. Gilbert Strang presented a 'famous' course in Linear Algebra at MIT. The course is available free as part of MITs 'Open Courseware' programme. If you wish to become proficient in Machine Learning, that course is a strong recommendation: <https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/>. My own experience was that both Strang's book and the lecture course videos are easier to follow if you have access to a computer-tool to experiment with some of the Algebra. The SymPy tool used in this section provides exactly such a tool and may be a significant aid to study.

The following tutorial provides what I regard as an minimum for required for a first course in Machine Learning. The material in this tutorial will be a requirement in order to read and understand most text-books, papers and even websites in the field of Machine Learning.

5.2 Instructions for Students

This workbook is different to some of the others in this series. Unlike the other workbooks, I have provided all of the code you need. This is because the best way of describing the operations in each case is to simply show the code.

However, I would still encourage you to type in all of the code snippets and experiment with them. Modifying and experimenting with this code is an excellent way of becoming familiar with operations on matrices and Linear Algebra. Having completed this workbook you may find that the recommended course text book on Linear Algebra (Strang) is rather easier to follow.

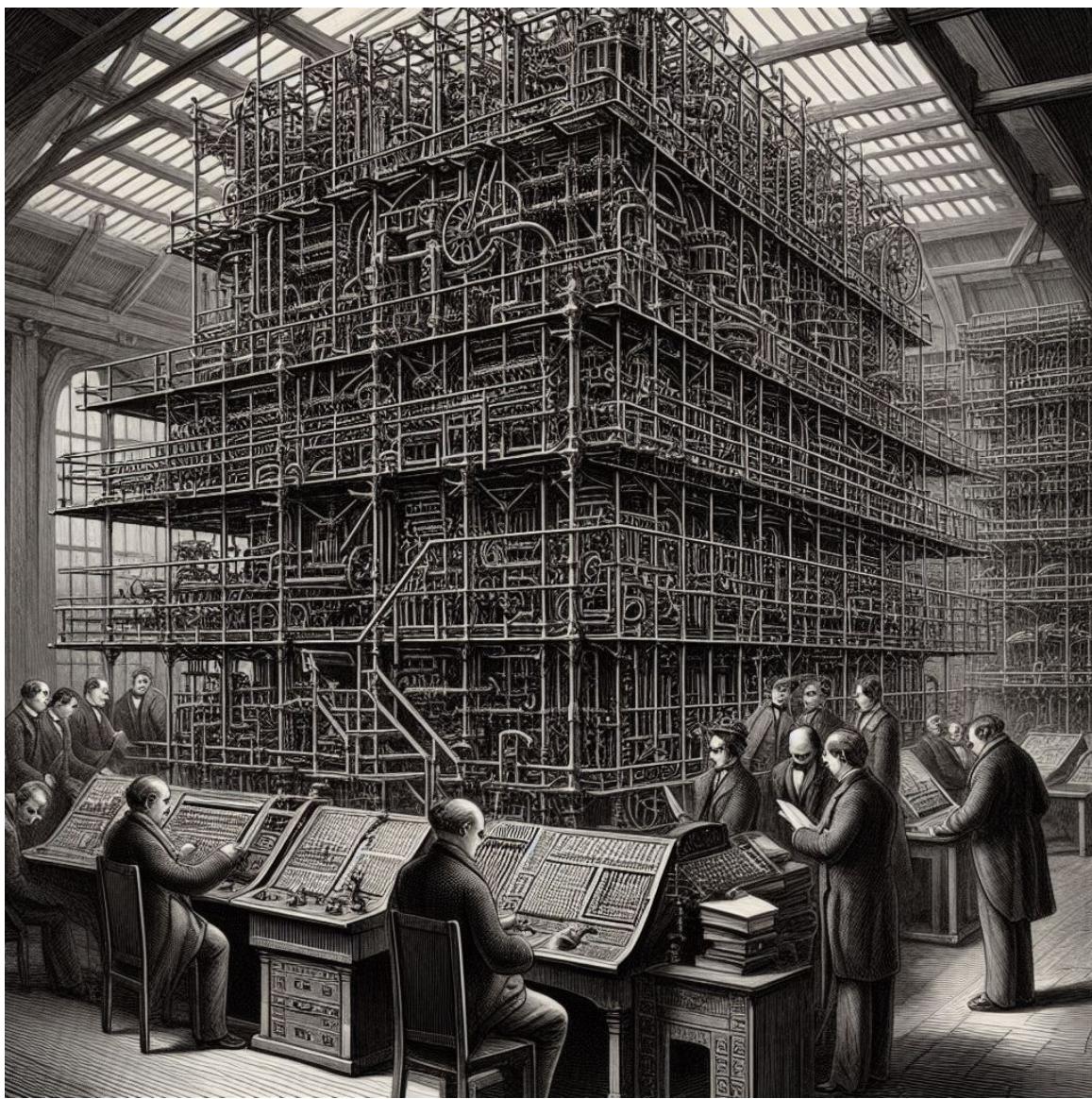


Figure 1: The 1853 attempt to solve ‘the matrix problem’. The so called ‘Steam Rivulet Engine’ was capable of solving large-scale, matrix calculations but unfortunately suffered from frequent gasket blow-outs and ultimately proved to be too dangerous for common use

5.3 Two Libraries for Linear Algebra - Sympy and Numpy

This tutorial introduces two libraries for linear algebra + Sympy + Numpy

The first is a *symbolic* mathematics library - and this make it easier to read and understand linear algebra as it might appear in a mathematics text-book (e.g. Strang).

The second is a very fast and efficient library used for numerical computation. It implements many more linear algebra operations than Sympy and is much more widely used in Machine Learning.

So Sympy is the best tool to help you learn linear algebra wheras you are much more likely to use Numpy for real Machine Learning projects.

5.4 Algebraic Equations represented in matrices

Linear Algebra is concerned with the use of equations such as:

$$1x + 2y = 3$$

$$4x + 5y = 6$$

In the context of Machine Learning you will often see such equations represented in the form of matrices:

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

At this point you should check your understanding of how the above equations are represented in matrix form

5.5 matrices using the Sympy Library

5.5.1 Defining matrices in Sympy

For example, we can represent the following two matrices:

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

.. and

$$y = \begin{bmatrix} 7 & 8 \\ 9 & 3 \\ 2 & 1 \end{bmatrix}$$

Let's start by using Sympy to define some matrices which we can use in later examples

```

1 from sympy import *
2
3 x = Matrix([[1,2,3], [4,5,6]])
4 x

```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```

1 y = Matrix([[7,8],[9,3],[2,1]])
2 y

```

$$\begin{bmatrix} 7 & 8 \\ 9 & 3 \\ 2 & 1 \end{bmatrix}$$

```

1 p = Matrix([[4,7,1],[4,3,2]])
2 p

```

$$\begin{bmatrix} 4 & 7 & 1 \\ 4 & 3 & 2 \end{bmatrix}$$

5.5.2 Matrix Addition and Subtraction

Two matrices can be added only if they are the same shape. Addition of matrices, simply means adding each element in one array to the corresponding element in the second array:

```

1 x + p

```

$$\begin{bmatrix} 5 & 9 & 4 \\ 8 & 8 & 8 \end{bmatrix}$$

The same goes for subtraction. matrices have to be the same shape, and they are subtracted on an element-by-element basis:

```

1 x - p

```

$$\begin{bmatrix} -3 & -5 & 2 \\ 0 & 2 & 4 \end{bmatrix}$$

5.5.3 Matrix multiplication by a constant

matrices can be multiplied by a constant. Each element is multiplied by the constant:

```
1 2 * x
```

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

5.5.4 Multiplying pairs of matrices

matrices can be multiplied together but they have to be the correct shape. The number of columns in x has to match the number of rows in y. Notice also the order of multiplication and the shape of the result

Remember that x has a shape of 2x3:

```
1 x
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

y has a shape of 3x2

```
1 y
```

$$\begin{bmatrix} 7 & 8 \\ 9 & 3 \\ 2 & 1 \end{bmatrix}$$

So the result will be a 2x2 matrix:

```
1 x * y
```

$$\begin{bmatrix} 31 & 17 \\ 85 & 53 \end{bmatrix}$$

And the multiplication won't work with if the matrices are of the wrong shape:

```
1 k = Matrix([[1,2],[3,4]])
2 k
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The next cell will give an error:

```
1 x * k
```

```
ShapeError: Matrix size mismatch: (2, 3) * (2, 2).
```

5.5.5 Dot-Product (Inner Product) of Vectors

It is frequently useful in Machine Learning to compute the ‘dot-product’ between two vectors. The dot-product measures the projection (shadow) of one vector onto another. The result is a scalar quantity.

```
1 v1 = Matrix([1,2,3,4,5])
2 v2 = Matrix([3,2,1,7,6])
3 v1
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

```
1 v2
```

$$\begin{bmatrix} 3 \\ 2 \\ 1 \\ 7 \\ 6 \end{bmatrix}$$

```
1 v1.dot(v2)
```

68

Which, because of symmetry is exactly the same as $v2.v1$:

```
1 v2.dot(v1)
```

68

5.5.6 Cross-product of Vectors

The other, common, vector operation is the cross-product. This essentially provides a measure of the area or volume ‘enclosed’ or ‘marked out’ by the vectors. It returns a vector result such that the magnitude of the vector is the area / volume and the direction of the vector is normal to the volume / area defined.

Sympy will only perform this operation with vectors of maximum length 3

```
1 v3 = Matrix([1,2,3])
2 v4 = Matrix([3,2,1])
3 v3
```

```
[1]  
[2]  
[3]
```

```
1 v4
```

```
[3]  
[2]  
[1]
```

```
1 v3.cross(v4)
```

```
[-4]  
[ 8]  
[-4]
```

5.5.7 Vector Norm - Length of a Vector

In machine learning we will often want to compute the length of a vector. There are actually many (an infinite number!) of ways of doing this .. but the most common is computed in exactly the same way as in normal geometry: the square-root of the sum of the squares of the coordinates.

Interestingly, Sympy shows the result symbolically rather than as a real number .. but then again, that is exactly what Sympy was designed to do!

```
1 v1.norm()
```

$\sqrt{55}$

```
1 v2.norm()
```

$3\sqrt{11}$

5.5.8 Determinant and Inverse of a Square Matrix

If you are not familiar with the definition, use and computation of determinants then you should check out one of the references (such as Strang) for an explanation. Essentially, square matrices can represent a transformation in m-dimensional space. The determinant indicates the scaling of an object (represented by a matrix / vector) in that space from before to after the transformation. If the determinant is zero, that means that the object is squished to zero volume (length) by the transformation. One implication of this is that there is no inverse defined for a matrix with determinant of zero : once an object is squished flat we can't un-squish it again.

Sympy enables easy computation of the determinant of a square matrix.

```
1 A = Matrix([[1, 2, 3], [3, 6, 2], [2, 0, 1]])  
2 A
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 6 & 2 \\ 2 & 0 & 1 \end{bmatrix}$$

```
1 A.det()
```

-28

A has a non-zero determinant and therefore the matrix has an inverse:

```
1 A.inv()
```

$$\begin{bmatrix} -\frac{3}{14} & \frac{1}{14} & \frac{1}{2} \\ -\frac{1}{28} & \frac{5}{28} & -\frac{1}{4} \\ \frac{3}{7} & -\frac{1}{7} & 0 \end{bmatrix}$$

However, if we started with a matrix with a determinant of zero:

```
1 Z = Matrix([[1,2,3],[4,5,6],[7,8,9]])  
2 Z
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1 Z.det()
```

0

.. the inverse of Z is not defined (the following will show an error)

```
1 Z.inv()
```

NonInvertibleMatrixError: Matrix det == 0; not invertible.

5.5.9 Other Matrix Operations from Linear Algebra

We will often want to think about matrices as concise representations of a system of linear equations. In that case there are several operations that will help us think about and compute properties of that system:

- Determinant - (as above) - is the matrix invertible / system of equations ‘soluble’)
- Rank - How many rows / columns in the matrix are independent
- Column space - The sub-space which is ‘accessible’ (‘marked out’) by the vectors in the matrix (which may not be a full m-dimensional space, even though the matrix has dimension $m \times n$)
- Null space - For a matrix ‘A’ - the set of all vectors ‘x’ that satisfy $Ax = 0$

The last of these are somewhat advanced concepts. If you are not familiar with these, then you are directed towards Gilbert Strangs “Linear Algebra” for an excellent explanation.

Consider this matrix, representing three equations with 4 unknowns:

```
1 A = Matrix([[1,3,3,2], [2,6,9,7], [-1,-3,3,4]])  
2 A
```

$$\begin{bmatrix} 1 & 3 & 3 & 2 \\ 2 & 6 & 9 & 7 \\ -1 & -3 & 3 & 4 \end{bmatrix}$$

But there are only actually 2 independent rows / columns here:

```
1 A.rank()
```

2

That is illustrated by using Gaussian Elimination to transform the matrix into reduced row echelon form:

```
1 U = A.rref()  
2 U[0]
```

$$\begin{bmatrix} 1 & 3 & 0 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(The slightly annoying ‘[0]’ here is because the sympy ‘rref’ function actually returns a vector result. The zeroth element of that vector is the reduced echelon form. The second element indicates the ‘pivot’ columns. The pivot columns are those with that mark out the column space of the matrix. Another way of saying this, is that they are the columns that have non-zero numbers along the ‘stair-case diagonal’ running from top-left to bottom-right of the reduced matrix.

```
1 U[1]
```

$$(0, 2)$$

And here is that column space (again, I will show each vector in the result separately):

```
1 C = A.columnspace()
2 C[0]
```

$$\begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

```
1 C[1]
```

$$\begin{bmatrix} 3 \\ 9 \\ 3 \end{bmatrix}$$

In this case, all solutions to the original system of equations exist on a 2-dimensional (flat) plane that cuts across 3-dimensional space. The two vectors above define that plane. Every point on the plane is some linear combination of the above two vectors.

The ‘nullspace’ of the matrix is the set of vectors defining the sub-space that would be mapped onto zero by the matrix. All linear combinations of vectors in this space will be transformed onto zero by the matrix.

```
1 N = A.nullspace()
2 N[0]
```

$$\begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
1 N[1]
```

$$\begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

Which can be demonstrated with some examples:

```
1 A * N[0]
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
1 A * (2 * N[0])
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
1 A * (5 * N[0] + 7 * N[1])
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Whereas all other vectors do not have this property:

```
1 NZ = Matrix([-4,1,0,0])
2 NZ
```

$$\begin{bmatrix} -4 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
1 A * NZ
```

$$\begin{bmatrix} -1 \\ -2 \\ 1 \end{bmatrix}$$

5.6 matrices in Numpy

The above should help you understand some of the operations of matrices and Linear Algebra. In practice however, sympy is much less frequently used in Machine Learning than Numpy. In general, in ML we need to be doing fast, numerical computation rather than slower, symbolic manipulation.

In this section I review the use of ‘Numpy’ - a very common library for numeric calculations in the field of Machine Learning.

```

1 import numpy as np
2
3 x = np.array([[1,2,2],[4,5,6]])
4 y = np.array([[7,8],[9,3], [2,1]])
5
6 print(x, "\n")
7 print(y)

```

$\begin{bmatrix} 1 & 2 & 2 \\ 4 & 5 & 6 \end{bmatrix}$

$\begin{bmatrix} 7 & 8 \\ 9 & 3 \\ 2 & 1 \end{bmatrix}$

Note that numpy's data type we are using to represent Matrices is actually called an 'array'
Numpy provides a variety of functions for manipulating matrices, including:

5.6.1 Matrix Addition

Here I will define a new matrix 'p' to make the addition more interesting.

Note that addition is only define for matrices of the same 'shape'. I.e. with equivalent numbers of rows and columns.

Addition of matrices is particularly easy to read and write due to the re-use of the '+' operator:

```

1 p = np.array([[4,7,1],[4,3,2]])
2 print("x =\n", x)
3 print("\np =\n", p)
4 print("\nx + p =\n", x + p)

```

x =
 $\begin{bmatrix} 1 & 2 & 2 \\ 4 & 5 & 6 \end{bmatrix}$

p =
 $\begin{bmatrix} 4 & 7 & 1 \\ 4 & 3 & 2 \end{bmatrix}$

x + p =
 $\begin{bmatrix} 5 & 9 & 3 \\ 8 & 8 & 8 \end{bmatrix}$

5.6.2 Matrix Subtraction

Analogously, the ‘-’ operator performs element-by-element matrix (array) subtraction:

```
1 print("x =\n", x)
2 print("\np =\n", p)
3 print("\nx - p =\n", x - p)
```

```
x =
[[1 2 2]
 [4 5 6]]
```

```
p =
[[4 7 1]
 [4 3 2]]
```

```
x - p =
[[-3 -5  1]
 [ 0  2  4]]
```

5.6.3 Matrix multiplication by a Constant

If you simply want to multiply each element of a matrix by a constant, then that can be achieved using the “*” operator:

```
1 print (5 * x)
```

```
[[ 5 10 10]
 [20 25 30]]
```

Otherwise, if you want to multiple two matrices together, this can be achieved with the following. Notice, as the convention with matrix multiplication that the rank of the matrices has to match: the number of columns in x has to match the number of rows in y. Notice also the order of multiplication and the shape of the result:

```
1 print("x =\n", x)
2 print("\ny =\n", y)
3 print("\nmatmul(x,y) =")
4 print(np.matmul(x,y))
```

```

x =
[[1 2 2]
 [4 5 6]]

y =
[[7 8]
 [9 3]
 [2 1]]

matmul(x,y) =
[[29 16]
 [85 53]]

```

or .. using the '@' operator rather than the more familiar '*' operator for matrix multiplication

```
1 print(x @ y)
```

```
[[29 16]
 [85 53]]
```

5.6.4 Dot-product (Sometimes called ‘inner product’)

Numpy also provides a function to calculate the ‘dot-product’ between two 1-dimensional arrays (also frequently referred to as ‘vectors’). The result of the numpy dot-product between two vectors is a number (scaler).

The dot-product measures the ‘projection’ of one vector onto another. That is, it provides a measure of similarity in the ‘direction’ of vectors. If vectors are orthogonal (at right-angles) to each other, then the scalar product is defined as zero. Otherwise, the scalar product indicates the size of the ‘shadow’ one vector would cast on the other.

In text books this operation is often shown as $x.y^T$

When thinking about vectors in space it is more frequently indicated as $|x|.|y|\cos(\theta)$, where theta is the angle between the lines.

Both of these provide the same result.

The scalar product can be calculated in numpy using ‘dot’:

```
1 v1 = np.array([1,2,3])
2 v2 = np.array([4,5,6])
3 print(np.dot(v1,v2))
```

Finally, numpy also provides an ‘inner’ function. For 1-D vectors the result of ‘inner’ is identical to the ‘dot’ product. In higher dimensions these operators perform different operations - however those are beyond the scope of this course.

```
1 print (np.inner(v1,v2))
```

32

5.6.5 Matrix Division by a constant

We divide each element of a matrix by a constant like this:

```
1 print(np.divide(x,2))
```

```
[[0.5 1. 1. ]
 [2. 2.5 3. ]]
```

By analogy with the above, you can also use the ‘/’ operator to divide a matrix by a constant:

```
1 print(x/2)
```

```
[[0.5 1. 1. ]
 [2. 2.5 3. ]]
```

We can do an element-by-element divide between two matrices of the same shape:

```
1 print (np.divide(x,p))
```

```
[[0.25 0.28571429 2.
 [1. 1.66666667 3. ]]
```

And again, use the ‘/’ operator if more convenient:

```
1 print (x/p)
```

```
[[0.25 0.28571429 2.
 [1. 1.66666667 3. ]]
```

5.6.6 Determinant of a matrix

The Determinant of a matrix is a measure of the ‘scaling’ that the matrix generates when it is applied to a series of vectors. This is very well explained here:

<https://towardsdatascience.com/what-really-is-a-matrix-determinant-89c09884164c>

Determinants can be calculated for square matrices.

Numpy provides an easy-to-use function to calculate determinants:

```
1 s = np.array([[1,2,3], [14,25,36], [74,38,29]])
2 print(np.linalg.det(s))
```

-81.0000000000024

5.6.7 Inverse of a matrix

Finally, numpy provides a function to determine the inverse of a matrix:

```
1 s_inv = np.linalg.inv(s)
2 print(s_inv)
```

```
[[ 7.9382716 -0.69135802  0.03703704]
 [-27.87654321  2.38271605 -0.07407407]
 [ 16.27160494 -1.35802469  0.03703704]]
```

matrices, of course, have the property that if they are multiplied by their inverse the result is the Identity matrix (1’s on the diagonal, 0’s elsewhere):

```
1 print( np.matmul(s,s_inv))
```

```
[[ 1.0000000e+00  6.66133815e-16  6.93889390e-18]
 [-2.84217094e-14  1.00000000e+00  1.38777878e-16]
 [-7.10542736e-15  7.32747196e-15  1.00000000e+00]]
```

Which is a close approximation to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5.6.8 Solutions to Linear Equations

As mentioned in the first section, series of linear equations can be represented as matrices.

$$1x + 2y = 1$$

$$3x + 5y = 2$$

In the context of Machine Learning you will often see such equations represented in the form of matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Such systems of linear equations may be soluble using numpy. By ‘soluble’ I mean that we can find values for x and y for which the system of equations is consistent.

```
1 M1 = np.array([[1, 2], [3, 5]])
2
3 M2 = np.array([1, 2])
4
5 A = np.linalg.solve(M1, M2)
6
7 print(A)
```

`[-1. 1.]`

In this case, substituting a value of ‘-1’ for x, and ‘1’ for y, produces a constant set of equations:

```
1 print(1 * -1 + 2 * 1)
```

`1`

```
1 print (3 * -1 + 5 * 1)
```

`2`

Of course, in some cases there will be no solution to a set of linear equation. For example:

$$1x + 2y = 1$$

$$2x + 4y = 3$$

represented as:

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

The following should produce a ‘Singular Matrix’ error, since there are no values for which the equations hold true:

```
1 M1 = np.array([[1, 2], [2, 4]])
2
3 M2 = np.array([1, 3])
4
5 A = np.linalg.solve(M1, M2)
6
7 print(A)
```

```
LinAlgError: Singular matrix
```

5.6.9 Matrix (or vector) Norm

It is often useful to be able to compute the ‘length’ of a vector. There are many ways to do this (technically there are an infinite number of ways of doing it!). However, the most common methods for computing lengths (distances) in Machine learning are the L1 and L2 norms.

The L2 norm is what, in standard geometry we call a “Euclidean distance” (or “Euclidean norm”) and is simply the square root or the sum of the squares of the dimensions. Here is the simplest example .. a 3,4,5 triangle:

```
1 myVector = [3,4]
2 print(np.linalg.norm(myVector))
```

```
5.0
```

Numpy extends this into higher dimensions, although calculated in an exactly analogous manner:

```
1 myVector = [1,2,3]
2 print(np.linalg.norm(myVector))
```

```
3.7416573867739413
```

Numpy can also calculate the ‘L1’ norm as follows. This is sometimes referred to as the ‘Manhattan distance’. The L1 is simply the sum of each vector dimension:

```
1 print(np.linalg.norm(myVector, ord = 1))
```

```
6.0
```

Chapter 6 - Clustering

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2024-07-12

Table of contents

6 Clustering	6 - 4
6.1 Introduction	6 - 4
6.2 Instructions for Students	6 - 4
6.3 Load the required libraries	6 - 5
6.4 Load the data into a Pandas Dataframe	6 - 5
6.5 Clean and tidy the Data	6 - 6
6.5.1 Deal with Missing Data	6 - 6
6.5.2 Improve Layout by Renaming Features and Reducing decimal places .	6 - 8
6.5.3 Deal with Categorical Data	6 - 8
6.6 Visualise the data as a chart to help understand it better	6 - 9
6.7 Scale (standardise) the data into a standard size	6 - 11
6.8 Decide the number of clusters	6 - 11
6.8.1 Silhouette Method	6 - 12
6.8.2 Elbow Method	6 - 14
6.9 Cluster the data (Build the model)	6 - 15
6.10 Review the Results	6 - 16
6.10.1 Identifying cluster centres	6 - 17
6.10.2 Visualising Clusters	6 - 18
6.11 Another method for Visualising Clusters	6 - 21
6.12 Bonus Material - Experimenting with Other Clustering Algorithms	6 - 23
6.12.1 DBSCAN	6 - 24
6.12.2 Optics	6 - 25

List of Figures

- 1 Scholars of the Kensington Ladies Mathematical Society enjoying the game of 'Kluster-Puc' : A passtime reserved for the mathematically gifted and intended to enhance reasoning and inference skills. (1842) 6 - 3
- 2 missingno matrix of the dataframe to identify missing data 6 - 7

3	Bar chart showing volume of missing data for each feature	6 - 7
4	Scatter Matrix of Numerical features	6 - 10
5	Silhouette score for various numbers of clusters	6 - 13
6	Elbow method plot	6 - 15
7	Scatter plot of ‘Age’ vs ‘Salary’ with each data point colour coded by cluster	6 - 19
8	Scatter plot of ‘Annual Spend’ vs ‘Salary’ with each data point colour coded by cluster	6 - 22
9	sns Pairplot of the dataset with each data point colour coded by cluster . . .	6 - 23
10	sns Pairplot for DBSCAN clustering	6 - 26
11	sns Pairplot for OPTICS clustering	6 - 28



Figure 1: Scholars of the Kensington Ladies Mathematical Society enjoying the game of 'Kluster-Puc' : A passtime reserved for the mathematically gifted and intended to enhance reasoning and inference skills. (1842)

6 Clustering

6.1 Introduction

In this workshop we will be introducing an un-supervised Machine Learning task - that of ‘Clustering’. Clustering is intended to help organise data into groups based on the similarity of data records.

This particular activity uses the ‘k-means’ algorithm - which is one of the simpler un-supervised Machine Learning algorithms. In the last part of the practical these is ‘bonus material’ covering other clustering algorithms.

6.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

6.3 Load the required libraries

For this practical session we will need the following libraries:

- pandas : Conventionally given the reference name ‘pd’
- ‘preprocessing’ from sklearn
- ‘metrics’ from sklearn
- numpy : Conventionally given the reference name ‘np’
- ‘kMeans’ from sklearn.cluster
- ‘DBSCAN’ from sklearn.cluster
- ‘OPTICS’ from sklearn.cluster
- ‘cluster_optics_dbSCAN’ from sklearn.cluster
- matplotlib.pyplot : Conventionally given the reference name ‘plt’
- seaborn : Conventionally given the reference name ‘sns’
- missingno : Conventionally given the reference name ‘msno’
- ‘scatter_matrix’ from pandas.plotting
- ‘Axes3D’ from mpl_toolkits.mplot3d

💡 Your code here

6.4 Load the data into a Pandas Dataframe

For the first part of this practical activity we will be a ‘synthetic’ data set .. that is, one that I created for this activity rather than being real customer data from a business. We are using this data set because it provides a useful illustration of the techniques whilst also removing any issues of privacy or security.

The data is contained within a file called ‘Retail Data v4 - unclean.csv’. Load this data into a pandas dataframe called ‘customer_data’

💡 Your code here

Display the first 20 rows of the ‘customer_data’ dataframe.

💡 Your code here

	Age	Gender	Married	Salary	Annual Spend
0	28.2	Male	Single	26908.95	331.56
1	43.5	Female	Married	39366.44	3071.18
2	27.7	Female	Single	NaN	1357.19
3	18.9	Male	Single	26235.55	769.78

	Age	Gender	Married	Salary	Annual Spend
4	18.0	Male	Single	30822.14	100.00
5	22.9	Female	Single	18334.52	2854.59
6	43.5	Female	Married	36642.04	1926.37
7	28.0	Male	Single	34612.04	853.10
8	52.3	Female	Married	34779.79	NaN
9	27.2	Female	Single	21901.08	566.21
10	47.3	Female	Married	33535.09	2344.79
11	31.0	Female	Single	24944.10	1661.97
12	34.9	Female	Single	23136.51	1343.36
13	45.8	Female	Married	27398.68	3250.98
14	32.8	Female	Single	21680.78	2600.72
15	28.7	Female	Single	27930.64	3717.44
16	70.1	Female	Married	38683.06	3965.79
17	28.0	Female	Single	25404.68	786.29
18	NaN	Male	Married	48905.96	3105.87
19	30.4	Female	Single	30677.96	1120.23

6.5 Clean and tidy the Data

6.5.1 Deal with Missing Data

By now you should be familiar with the process of data cleaning. Perform the following data cleaning operations on the ‘customer_data’ dataframe.

First, display a total count of the missing data.

💡 Your code here

```
Count of missing data = 29
```

Visualise missing data using ‘missingno’

💡 Your code here

Draw a bar-chart to indicate the amount of missing data in each feature:

💡 Your code here

Impute missing data using the mean of other data from the same feature

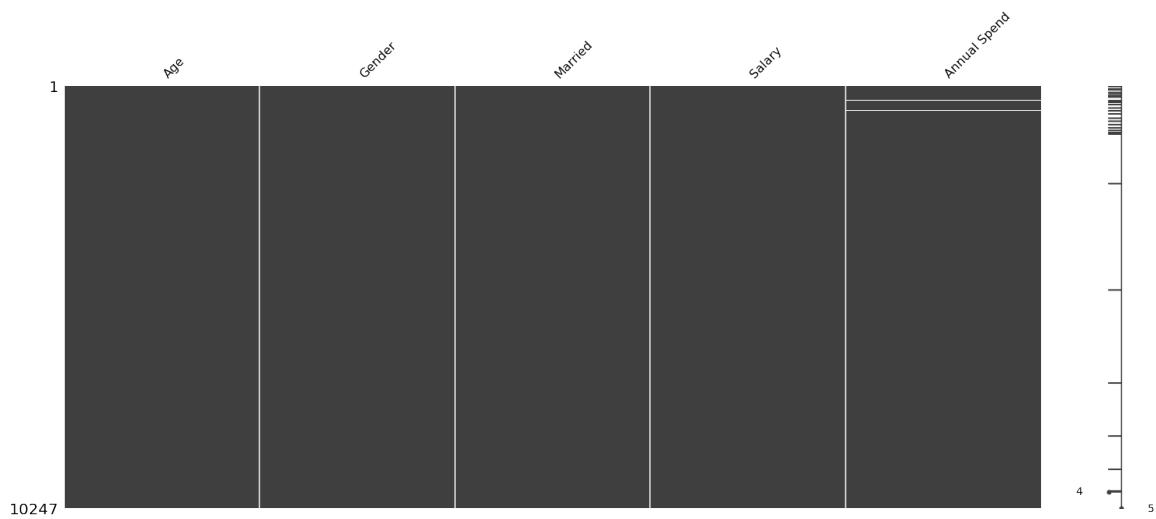


Figure 2: missingno matrix of the data frame to identify missing data

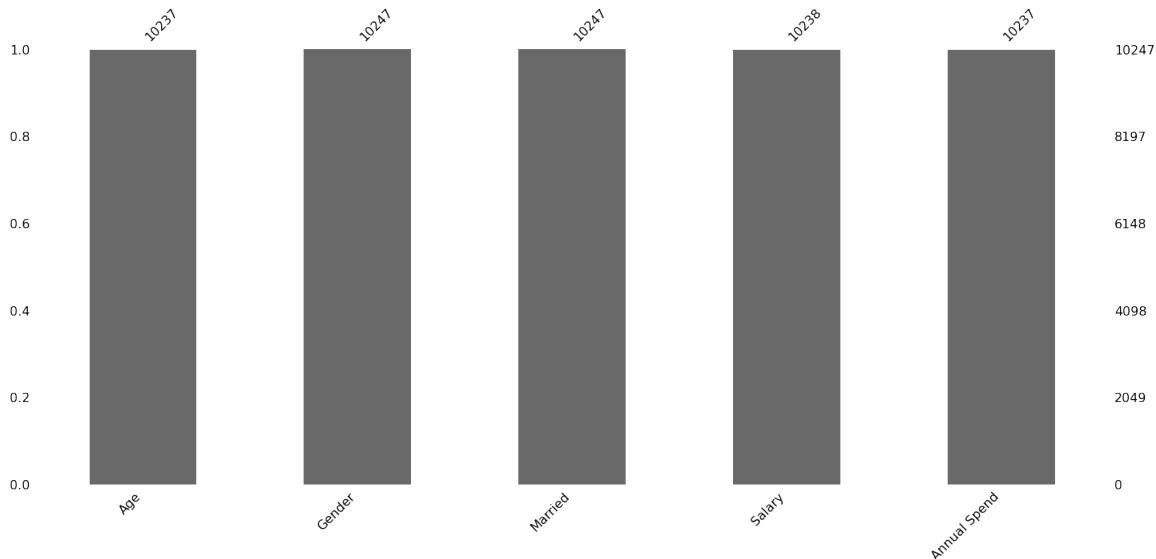


Figure 3: Bar chart showing volume of missing data for each feature

 Your code here

Do a final check by re-counting the amount of missing data in the ‘customer_data’ data set

 Your code here

```
Count of missing data = 0
```

6.5.2 Improve Layout by Renaming Features and Reducing decimal places

I want to rename the two categorical features at this point simply to make them shorter and improve page layout for this workbook. This is not strictly necessary, but is a useful thing to know how to do when using Pandas.

Fist, add code that changes the category feature names as follows:

- ‘Gender’ into ‘G’
- ‘Married’ into ‘M’

Then set Pandas to display only a single decimal place for float values.

 Your code here

	Age	G	M	Salary	Annual Spend
0	28.2	Male	Single	26909.0	331.6
1	43.5	Female	Married	39366.4	3071.2
2	27.7	Female	Single	36629.1	1357.2
3	18.9	Male	Single	26235.5	769.8
4	18.0	Male	Single	30822.1	100.0
...
10242	23.8	Female	Single	24218.5	1779.7
10243	26.6	Female	Single	26793.3	2352.4
10244	22.1	Male	Single	29096.5	410.2
10245	23.0	Female	Single	23438.0	1945.9
10246	29.8	Male	Single	28754.0	2673.3

6.5.3 Deal with Categorical Data

Two of the features are category data (strings). Convert these into a one-hot-encoded form

💡 Your code here

Review the final data set by again displaying the first 20 rows.

💡 Your code here

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
0	28.2	26909.0	331.6	0	1	0	1
1	43.5	39366.4	3071.2	1	0	1	0
2	27.7	36629.1	1357.2	1	0	0	1
3	18.9	26235.5	769.8	0	1	0	1
4	18.0	30822.1	100.0	0	1	0	1
5	22.9	18334.5	2854.6	1	0	0	1
6	43.5	36642.0	1926.4	1	0	1	0
7	28.0	34612.0	853.1	0	1	0	1
8	52.3	34779.8	2159.1	1	0	1	0
9	27.2	21901.1	566.2	1	0	0	1
10	47.3	33535.1	2344.8	1	0	1	0
11	31.0	24944.1	1662.0	1	0	0	1
12	34.9	23136.5	1343.4	1	0	0	1
13	45.8	27398.7	3251.0	1	0	1	0
14	32.8	21680.8	2600.7	1	0	0	1
15	28.7	27930.6	3717.4	1	0	0	1
16	70.1	38683.1	3965.8	1	0	1	0
17	28.0	25404.7	786.3	1	0	0	1
18	39.6	48906.0	3105.9	0	1	1	0
19	30.4	30678.0	1120.2	1	0	0	1

6.6 Visualise the data as a chart to help understand it better

Add code to help you visualise the data. The task is to visualise the numerical columns ('Age', 'Salary' and 'Annual Spend'). We want to see a grid that shows the relationships between each of these three variables and also provides a histogram that shows the distribution for each variable.

💡 Your code here

It is already fairly easy to see that there is some natural clustering of data here.

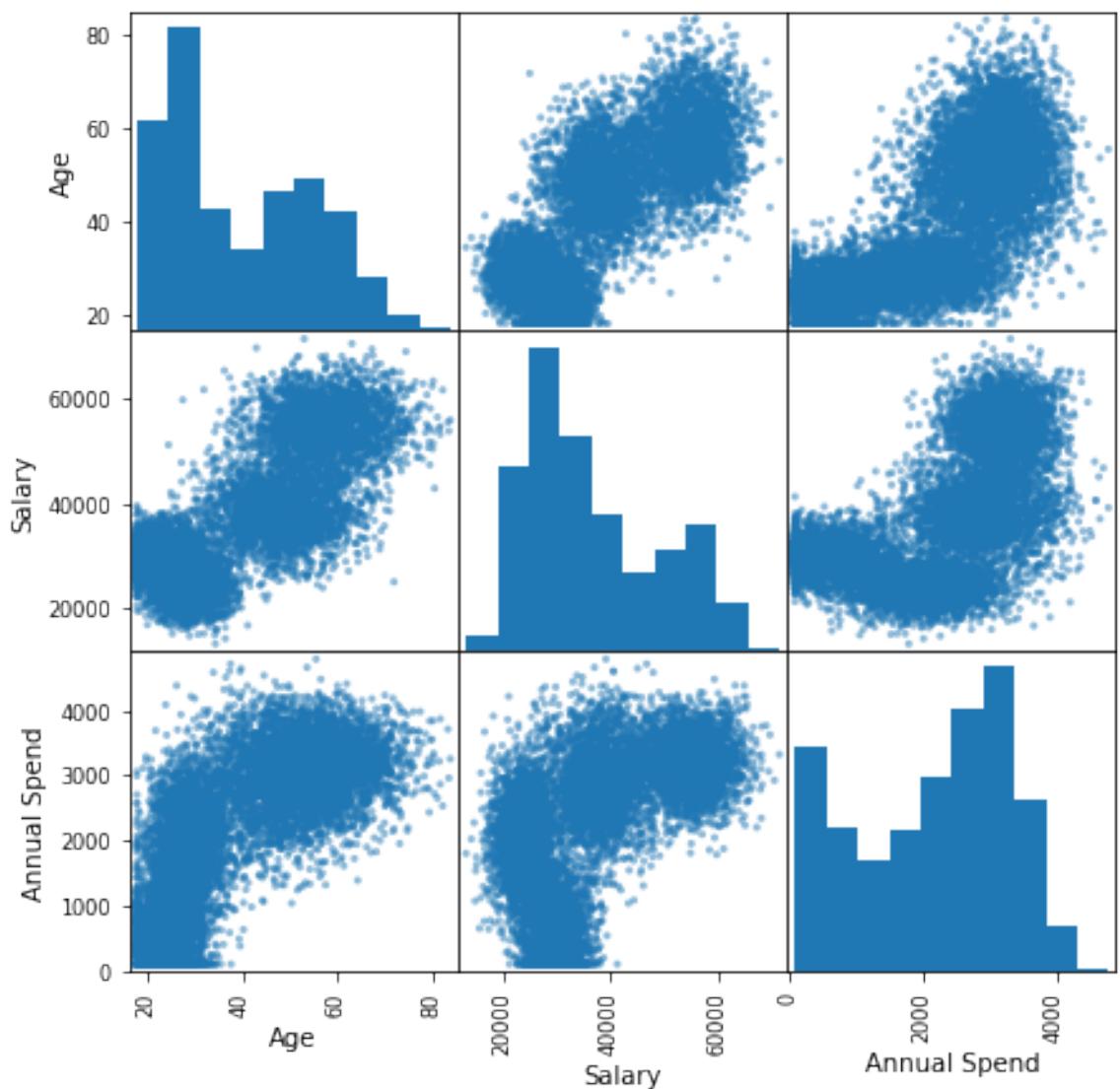


Figure 4: Scatter Matrix of Numerical features

6.7 Scale (standardise) the data into a standard size

Scaling is often important pre-processing step for Machine Learning. In particular, standardisation can often make algorithms run more quickly. There are several approaches to scaling. In this case we are going to shift the data so that its mean is around zero. The data is also scaled to a width which matches a ‘Normal’ (Gaussian) distribution.

This concept will be explored in one of the later lectures in the course. However, for now, we will just go ahead and re-scale the data.

Create code that uses the `sklearn preprocessing.scale` method to standardise the dataframe. Store the standardised data into a new dataframe called ‘`standardized_customer_data`’

💡 Your code here

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
1	0.3	0.2	0.8	1.0	-1.0	1.0	-1.0
2	-0.8	0.0	-0.7	1.0	-1.0	-1.0	1.0
3	-1.4	-0.8	-1.2	-1.0	1.0	-1.0	1.0
4	-1.4	-0.5	-1.8	-1.0	1.0	-1.0	1.0
5	-1.1	-1.5	0.6	1.0	-1.0	-1.0	1.0
6	0.3	0.0	-0.2	1.0	-1.0	1.0	-1.0
7	-0.8	-0.2	-1.1	-1.0	1.0	-1.0	1.0
8	0.8	-0.2	0.0	1.0	-1.0	1.0	-1.0
9	-0.8	-1.2	-1.4	1.0	-1.0	-1.0	1.0
10	0.5	-0.3	0.2	1.0	-1.0	1.0	-1.0
11	-0.6	-0.9	-0.4	1.0	-1.0	-1.0	1.0
12	-0.3	-1.1	-0.7	1.0	-1.0	-1.0	1.0
13	0.4	-0.7	1.0	1.0	-1.0	1.0	-1.0
14	-0.4	-1.2	0.4	1.0	-1.0	-1.0	1.0
15	-0.7	-0.7	1.4	1.0	-1.0	-1.0	1.0
16	2.0	0.2	1.6	1.0	-1.0	1.0	-1.0
17	-0.8	-0.9	-1.2	1.0	-1.0	-1.0	1.0
18	0.0	1.0	0.8	-1.0	1.0	1.0	-1.0
19	-0.6	-0.5	-0.9	1.0	-1.0	-1.0	1.0

6.8 Decide the number of clusters

There are several approaches to determining the number of clusters. It may be that we determine the number of clusters arbitrarily. For example, this might be because we wish to manage a certain number of clusters for business operational reasons. For example, we may simple decide that 4 different customer groups is all we can practically managed.

In some cases, scatter plots will show that there are visibly distinct clusters in the data.

Alternatively, we can use statistics from the clustering to help determine a ‘natural’ number of clusters. Two methods for doing this are presented in the following sections.

6.8.1 Silhouette Method

One example of a metric that with help in this regard is a ‘Silhouette’ chart, as shown below.

The silhouette function returns a value between -1 and 1 and measures the extent to which clusters overlap. The higher the score the better separated are the clusters.

The silhouette library function is documented at:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

Add the following code to your notebook to display the silhouette score for various numbers of clusters.

 Tutor provided code

```
def k_silhouette(X, clusters):
    K = range(2, clusters+1)
    score = []
    for k in K:
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        labels = kmeans.labels_
        score.append(metrics.silhouette_score(X, labels,
                                               metric='euclidean'))

    plt.plot(K, score, 'b*-')
    plt.xlabel('k')
    plt.ylabel('silhouette_score')

    plt.show();

k_silhouette(standardized_customer_data_df, 15)
```

If you wish, you can annotate diagrams like this for reports and presentations

 Tutor provided code

```
def k_silhouette_annotated(X, clusters):
    K = range(2, clusters+1)
    scores = []
    for k in K:
```

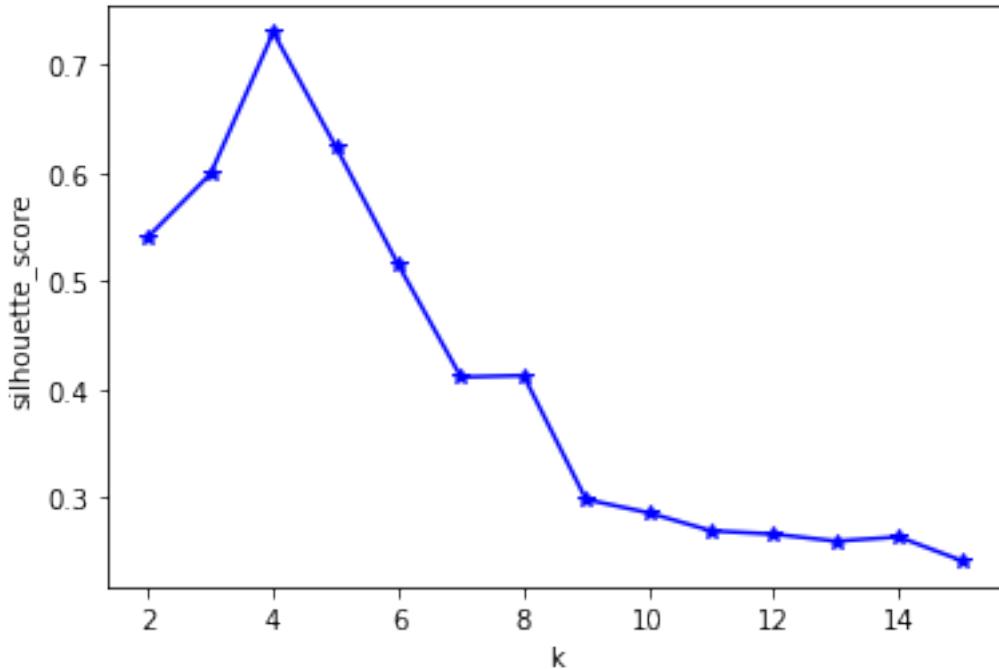


Figure 5: Silhouette score for various numbers of clusters

```

kmeans = KMeans(n_clusters=k)
kmeans.fit(X)
labels = kmeans.labels_
scores.append(metrics.silhouette_score(X, labels,
                                         metric='euclidean'))

fig = plt.figure()
ax = fig.add_subplot(111)

line = ax.plot(K, scores)

ymax = max(scores)
i = scores.index(ymax)
xmax = K[i]

ax.annotate(("{} clusters score: {}".format(xmax, round(ymax,4))),
            xy=(xmax, ymax), xytext=(xmax, ymax+0.12),
            arrowprops=dict(facecolor='red',
                            shrink=0.0005,
                            width=0.5,
                            headwidth=8,

```

```

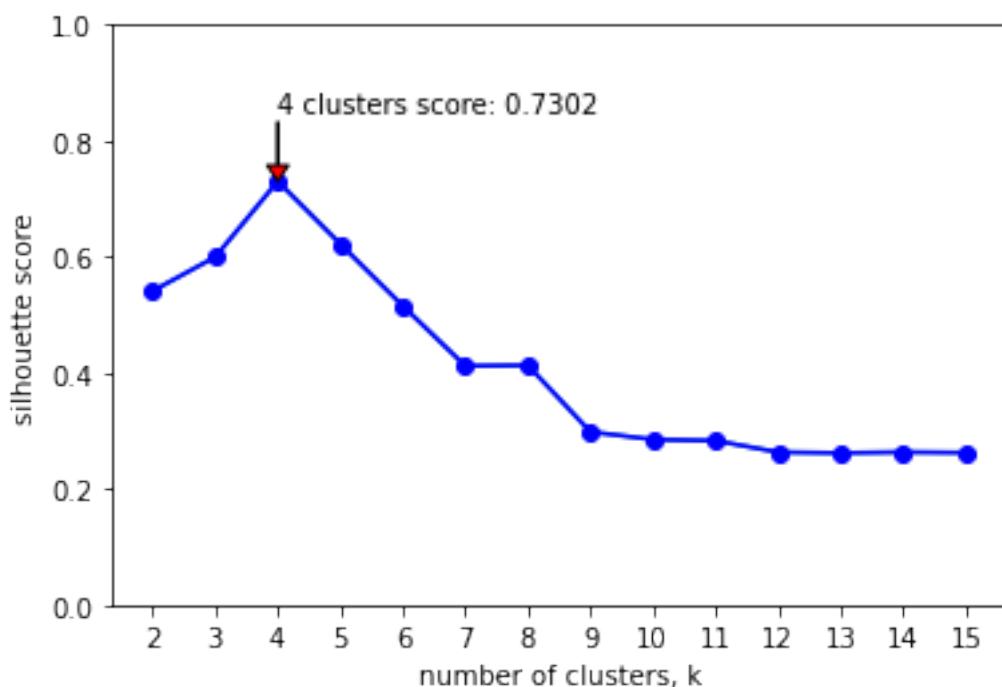
        headlength=6),
    )

ax.set_ylim(0,1)

plt.plot(K, scores, '-o', color='blue');
plt.xlabel('number of clusters, k');
plt.ylabel('silhouette score');
plt.xticks(K);
plt.show();

k_silhouette_annotated(standardized_customer_data_df, 15)

```



6.8.2 Elbow Method

A second, commonly applied way of determining an ‘optimum’ number of clusters is the ‘Elbow’ method.

To calculate the Elbow Method, we use the ‘.inertia_’ property of the sklearn kmeans function. This gives us the sum of the squared distances of each point from the cluster centre. As you will see from the graph below, as the number of clusters increases, the Standard Square Error (SSE) rapidly falls to a value. After this, there is very little improvement in

terms of moving points closer to cluster centres. The ‘elbow’ value on the graph indicates the optimum number of clusters.

 Tutor provided code

```
sse = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k,)
    kmeans.fit(standardized_customer_data_df)
    sse.append(kmeans.inertia_)
x = range(1, 11)
plt.xlabel('K')
plt.ylabel('SSE')
plt.plot(x, sse, 'o-')
plt.show()
```

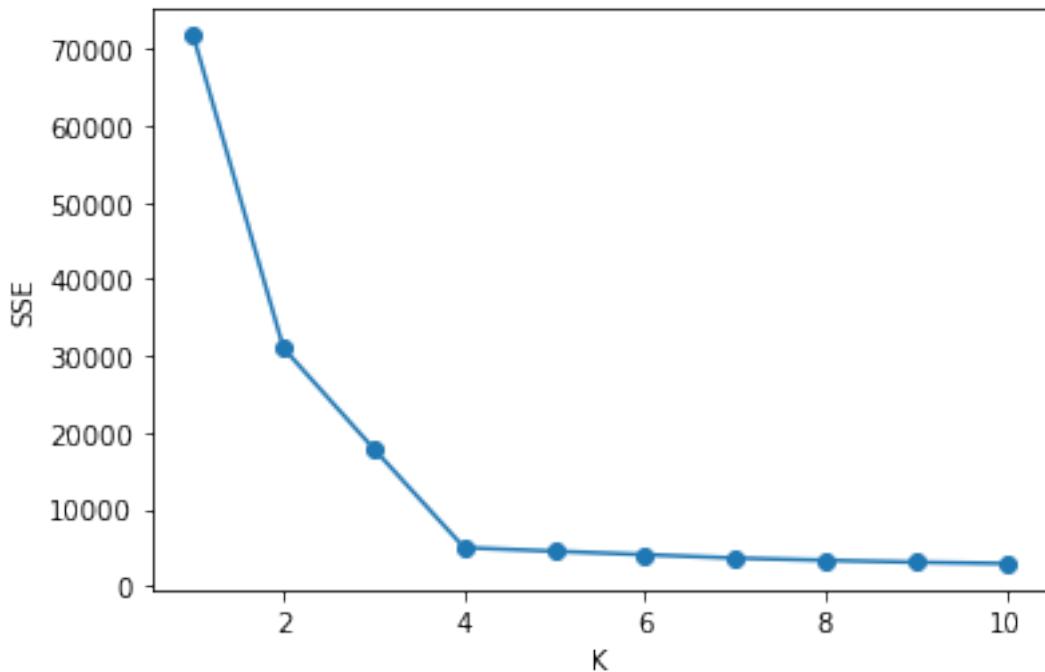


Figure 6: Elbow method plot

In this case we see good correspondence between the Silhouette and Elbow methods .. the number of clusters here is 4.

6.9 Cluster the data (Build the model)

In this section we will use the sklean ‘kmeans’ algorithm to cluster the (now standardised) customer data into clusters.

First we create the object ('machine') that we will use to build the model. In this case I arbitrarily decided that I want the data grouped into 4 clusters. One of the limitations of the KMeans algorithm is that the user has to decide how many clusters that the data should be arranged into.

Create an object called 'kmeans' as in instance of the sklearn 'KMeans' class. Set the number of cluster ('n_clusters') to 4.

 Your code here

Apply the .fit() method of the KMeans class to the standardised data frame.

 Your code here

```
KMeans(n_clusters=4)
```

The 'fit' method determines the centres of each cluster using the k-means algorithm.

Having identified the centres of each cluster, we now assign each member of the data-set to one of the clusters. Members are assigned to the cluster which has its centre closest to it.

Add code to apply the 'fit_predict' method to the standardised data frame. Store the result in a variable 'y_km'

 Your code here

6.10 Review the Results

The 'y_km' variable stores a list of numbers. Each number represents the cluster assignment for the corresponding row of data in the standardised data frame.

Add code to print the first twenty rows of y_km

 Your code here

```
y_km[0:20] = [3 2 1 3 3 1 2 3 2 1 2 1 1 2 1 1 2 1 0 1]
```

Alternatively, we could display the members of particular clusters .. starting with those in group 2

 Your code here

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
1	43.5	39366.4	3071.2	1	0	1	0
6	43.5	36642.0	1926.4	1	0	1	0
8	52.3	34779.8	2159.1	1	0	1	0
10	47.3	33535.1	2344.8	1	0	1	0
13	45.8	27398.7	3251.0	1	0	1	0
16	70.1	38683.1	3965.8	1	0	1	0
23	26.3	38994.8	4377.9	1	0	1	0
24	46.5	34270.0	3361.1	1	0	1	0
27	43.3	37264.7	2919.6	1	0	1	0
30	37.4	34761.2	1954.9	1	0	1	0

6.10.1 Identifying cluster centres

It is often useful to identify the centre of clusters as they are in some way ‘representative’ of each cluster. They can also be used to measure the distance of any individual from the ‘centre’ or ‘ideal’ case.

First, create a new dataframe called ‘customer_clusters’ which is a copy of ’standardized_customer_data_df’.

To this dataframe, append a new column called ‘y_km’ which is a copy of the y_km cluster numbers from the above activities.

 Your code here

	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single	y_km
0	-0.8	-0.8	-1.6	-1.0	1.0	-1.0	1.0	3
1	0.3	0.2	0.8	1.0	-1.0	1.0	-1.0	2
2	-0.8	0.0	-0.7	1.0	-1.0	-1.0	1.0	1
3	-1.4	-0.8	-1.2	-1.0	1.0	-1.0	1.0	3
4	-1.4	-0.5	-1.8	-1.0	1.0	-1.0	1.0	3
...
10242	-1.0	-1.0	-0.3	1.0	-1.0	-1.0	1.0	1
10243	-0.9	-0.8	0.2	1.0	-1.0	-1.0	1.0	1
10244	-1.2	-0.6	-1.5	-1.0	1.0	-1.0	1.0	3
10245	-1.1	-1.1	-0.2	1.0	-1.0	-1.0	1.0	1
10246	-0.6	-0.6	0.4	-1.0	1.0	-1.0	1.0	3

💡 Your code here

We can then use the ‘groupby’ method of pandas to collect together customers in each cluster and find the mean values for each feature.

💡 Tutor provided code

```
Cluster_Means = customer_clusters.groupby(['y_km']).mean().round(1)  
Cluster_Means
```

y_km	Age	Salary	Annual Spend	G_Female	G_Male	M_Married	M_Single
0	1.2	1.5	0.9	-1.0	1.0	1.0	-1.0
1	-0.7	-1.0	-0.1	1.0	-1.0	-1.0	1.0
2	0.5	0.1	0.6	1.0	-1.0	1.0	-1.0
3	-1.0	-0.5	-1.4	-1.0	1.0	-1.0	1.0

6.10.2 Visualising Clusters

We can also review the clusters in chart form.

In the following charts, the cluster that a particular point belongs to is indicated by a colour.

.. First plotting Age Vs Salary

💡 Tutor provided code

```
plt.figure(figsize=(7,7))  
plt.scatter(customer_data[y_km ==0] ['Age'],  
           customer_data[y_km == 0] ['Salary'],  
           s=15, c='red', alpha=.5)  
plt.scatter(customer_data[y_km ==1] ['Age'],  
           customer_data[y_km == 1] ['Salary'],  
           s=15, c='black', alpha=.5)  
plt.scatter(customer_data[y_km ==2] ['Age'],  
           customer_data[y_km == 2] ['Salary'],  
           s=15, c='blue', alpha=.5)  
plt.scatter(customer_data[y_km ==3] ['Age'],  
           customer_data[y_km == 3] ['Salary'],  
           s=15, c='cyan', alpha=.5)
```

Then plot Annual Spend against Salary

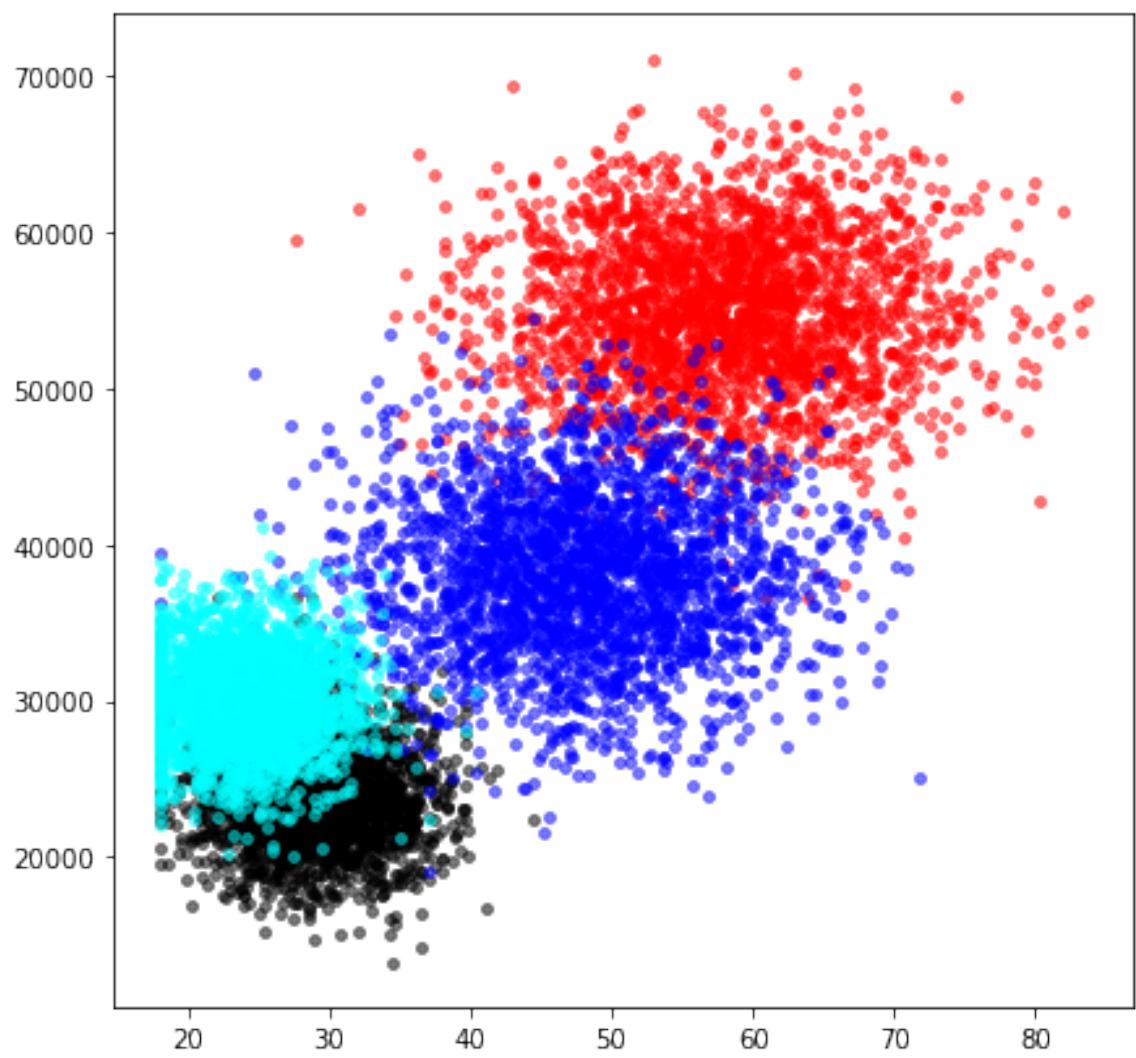
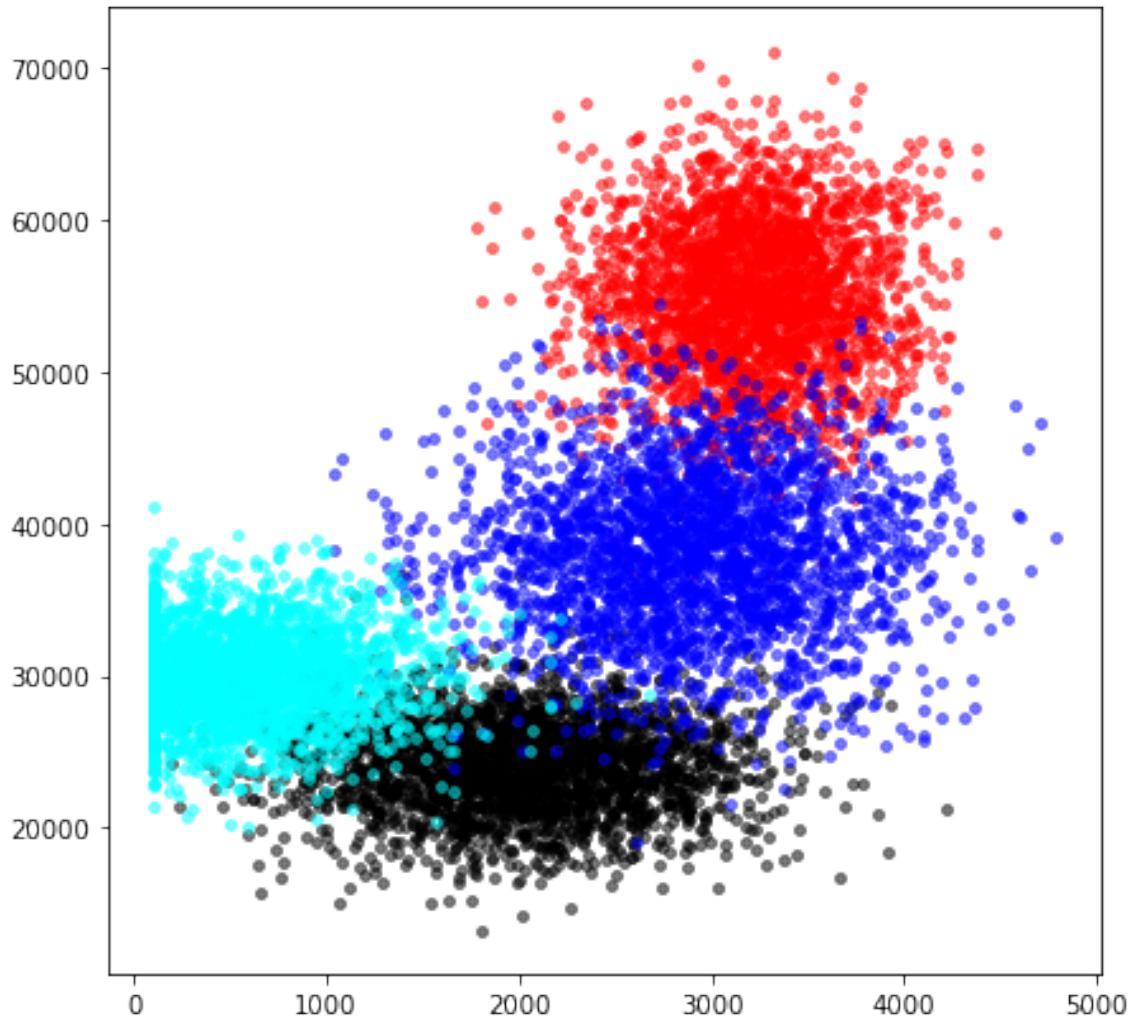


Figure 7: Scatter plot of ‘Age’ vs ‘Salary’ with each data point colour coded by cluster

💡 Your code here

```
<function matplotlib.pyplot.show(close=None, block=None)>
```

Scatter plot of ‘Annual Spend’ vs ‘Salary’ with each data point colour coded by cluster



We can even take a ‘3D’ view of the data

💡 Tutor provided code

```
%matplotlib  
  
fig = plt.figure(figsize=(10,10))  
ax = fig.add_subplot(111, projection='3d')
```

```

ax.view_init(20, 20)
ax.set_xlabel('Annual Spend')
ax.set_ylabel('Salary')
ax.set_zlabel('Age')
ax.scatter(customer_data[y_km ==0] ['Annual Spend'],
           customer_data[y_km == 0] ['Salary'],
           customer_data[y_km == 0] ['Age'],
           s=15, c='red', alpha=.3)
ax.scatter(customer_data[y_km ==1] ['Annual Spend'],
           customer_data[y_km == 1] ['Salary'],
           customer_data[y_km == 1] ['Age'],
           s=15, c='black', alpha=.3)
ax.scatter(customer_data[y_km ==2] ['Annual Spend'],
           customer_data[y_km == 2] ['Salary'],
           customer_data[y_km == 2] ['Age'],
           s=15, c='blue', alpha=.3)
ax.scatter(customer_data[y_km ==3] ['Annual Spend'],
           customer_data[y_km == 3] ['Salary'],
           customer_data[y_km == 3] ['Age'],
           s=15, c='cyan', alpha=.3)

```

In the above code I have included a line:

```
%matplotlib
```

This should cause a 3D graph to display in a pop-out window.

If you prefer, you can also display the chart as a static imagine ‘in line’ using the following command:

```
%matplotlib inline
```

6.11 Another method for Visualising Clusters

Another attractive method for displaying discovered clusters is provided by using sns pairplot
..

 Tutor provided code

```

# The 'X' on the following lines is simply to make
# this code easier to follow

X = standardized_customer_data_df[['Age',
                                    'Salary',

```

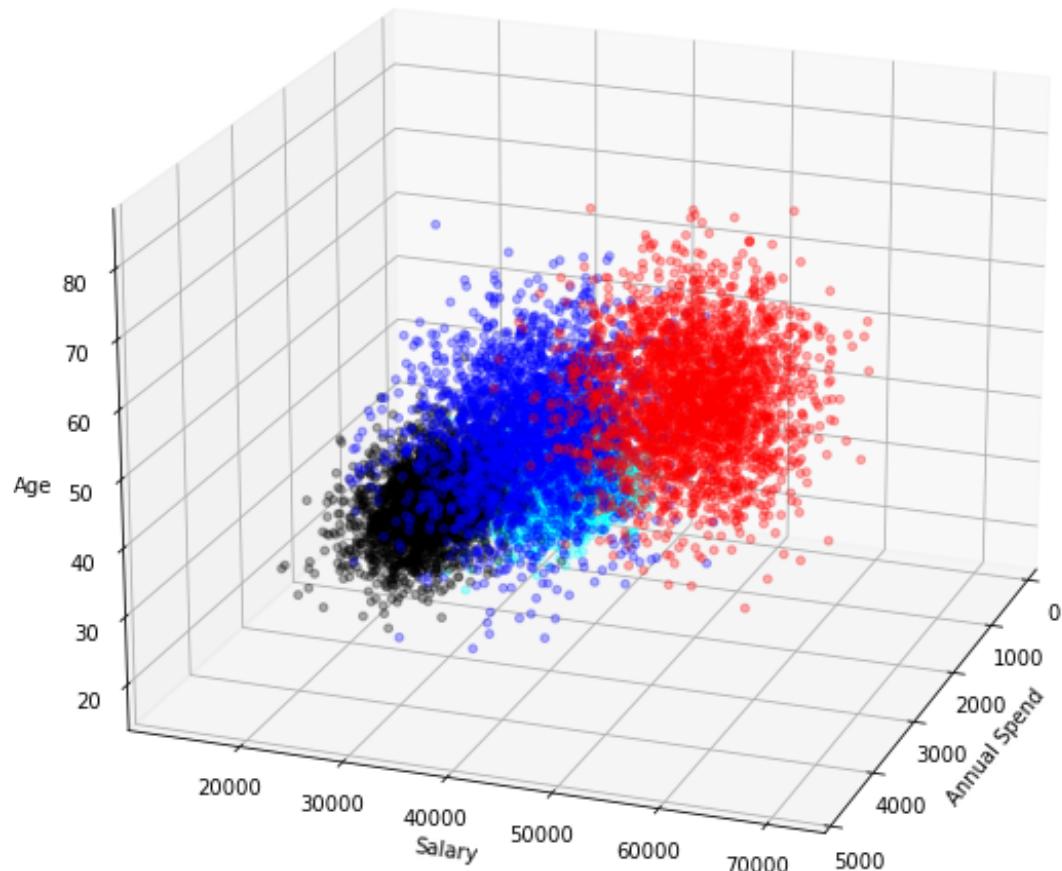


Figure 8: Scatter plot of 'Annual Spend' vs 'Salary' with each data point colour coded by cluster

```

'Annual Spend']].copy()

X["y_km"] = y_km.astype(str)
sns.pairplot(X, hue='y_km')
plt.show()

```

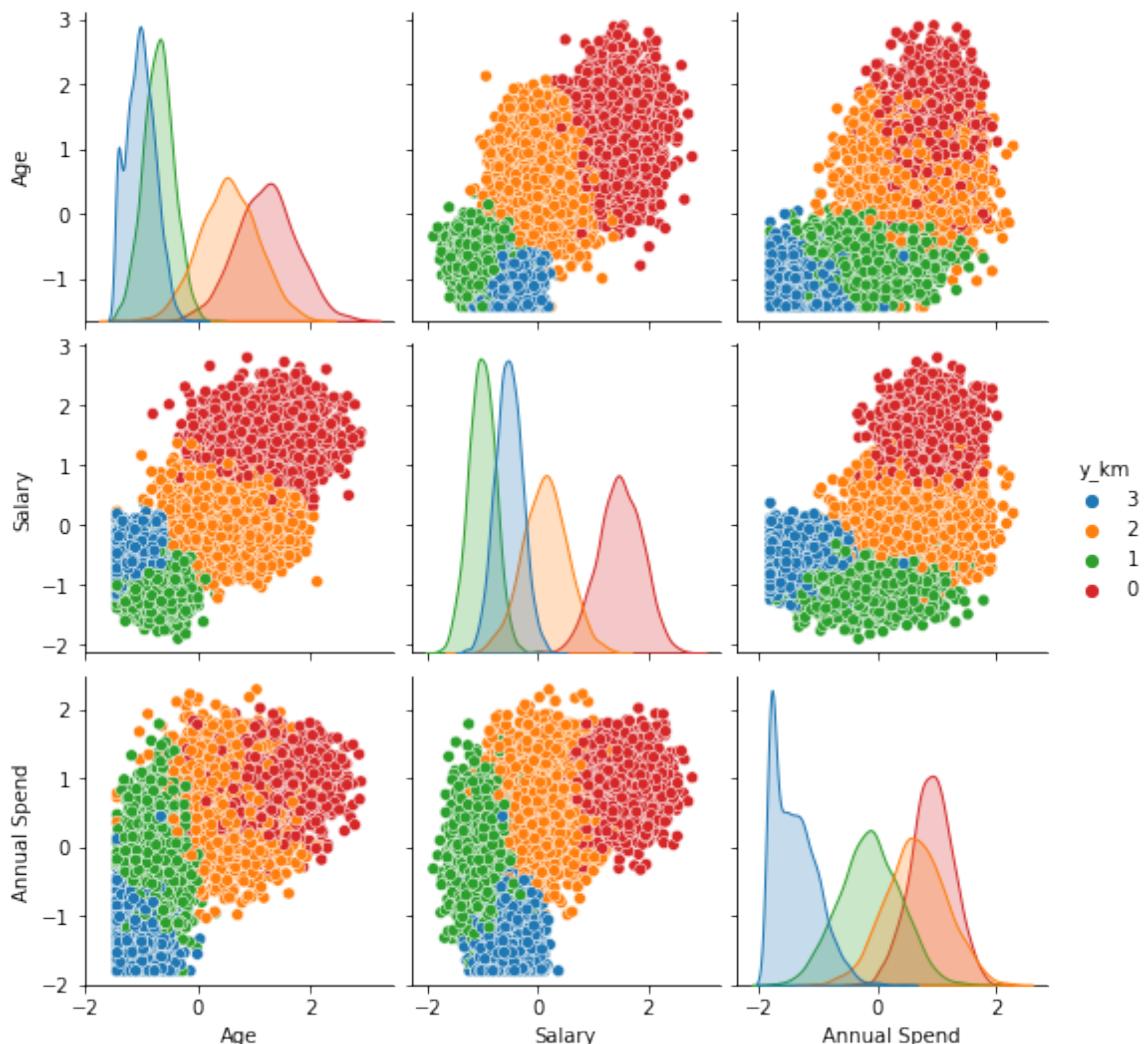


Figure 9: sns Pairplot of the dataset with each data point colour coded by cluster

6.12 Bonus Material - Experimenting with Other Clustering Algorithms

sklearn provides a large set of different clustering methods. If you have time available, then you could use your time usefully to explore some of those other methods. The main sklearn URL for clustering is here:

<https://scikit-learn.org/stable/modules/clustering.html>

And below are some of the examples I selected to experiment with.

6.12.1 DBSCAN

DBSCAN is documented here:

<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html>

DBSCAN has a number of difference to kmeans: The algorithm ..

- Determines the optimum number of clusters - you are not required to pass this as a parameter
- Identifies ‘noise points’ that do not fit the model clustering.

First, create an object called ‘db’ which is an instance of the sklearn ‘DBSCAN’ class. For now, do not pass any parameters to this instantiation.

 Your code here

Then, as is consistent throughout sklearn, cluster the data using the ‘fit’ method, passing ‘standardized_customer_data_df’ as a parameter.

 Your code here

`DBSCAN()`

To make the following code slightly easier to read, I copied the result labels from the ‘db.labels_’ attribute into a variable called ‘labels’. ‘labels’ is an array of dtype int64

 Your code here

```
labels = [ 0  1  2 ...  0  2 -1]
```

It is useful to know how many different clusters have been identified. That can be obtained by reviewing the number of unique label values. An easy way to do this in Python is to create a set (using the ‘set()’ function). A set created from the list of labels will include unique values of each label value.

 Your code here

```
label_set = {0, 1, 2, 3, -1}
```

An interesting feature of DBSCAN is that it identifies ‘noise points’ - those that don’t fit into the cluster model. These are labelled ‘-1’. We can count the number of points that have been identified as noise by first converting the labels into a list ..

💡 Your code here

```
label_list[0:10] = [0, 1, 2, 0, 0, 2, 1, 0, 1, 2]
```

Then counting the number of items in that list that are labelled ‘-1’. This can be achieved using the ‘.count(-1)’ function.

💡 Your code here

```
noise_point_count = 6
```

After that, we can create charts of the clusters as previously.

💡 Your code here

Note that the documentation points out that there is an important parameter for DBSCAN called ‘eps’. This represents the maximum distance between two points for them to be considered as part of the same neighborhood. So an instantiation of DBSCAN would normally look something like:

```
db = DBSCAN(eps=0.3, min_samples=10)
```

6.12.2 Optics

This is the best article I have found on the OPTICS algo: <https://saturncloud.io/blog/python-implementation-of-optics-clustering-algorithm/>

This article provides some more information: <https://www.geeksforgeeks.org/ml-optics-clustering-explanation/>

There are two another articles here .. although unfortunately they are behind the Medium paywall:

<https://towardsdatascience.com/understanding-optics-and-implementation-with-python-143572abdfb6>

<https://pub.towardsai.net/fully-explained-optics-clustering-with-python-example-4553108fa04b>

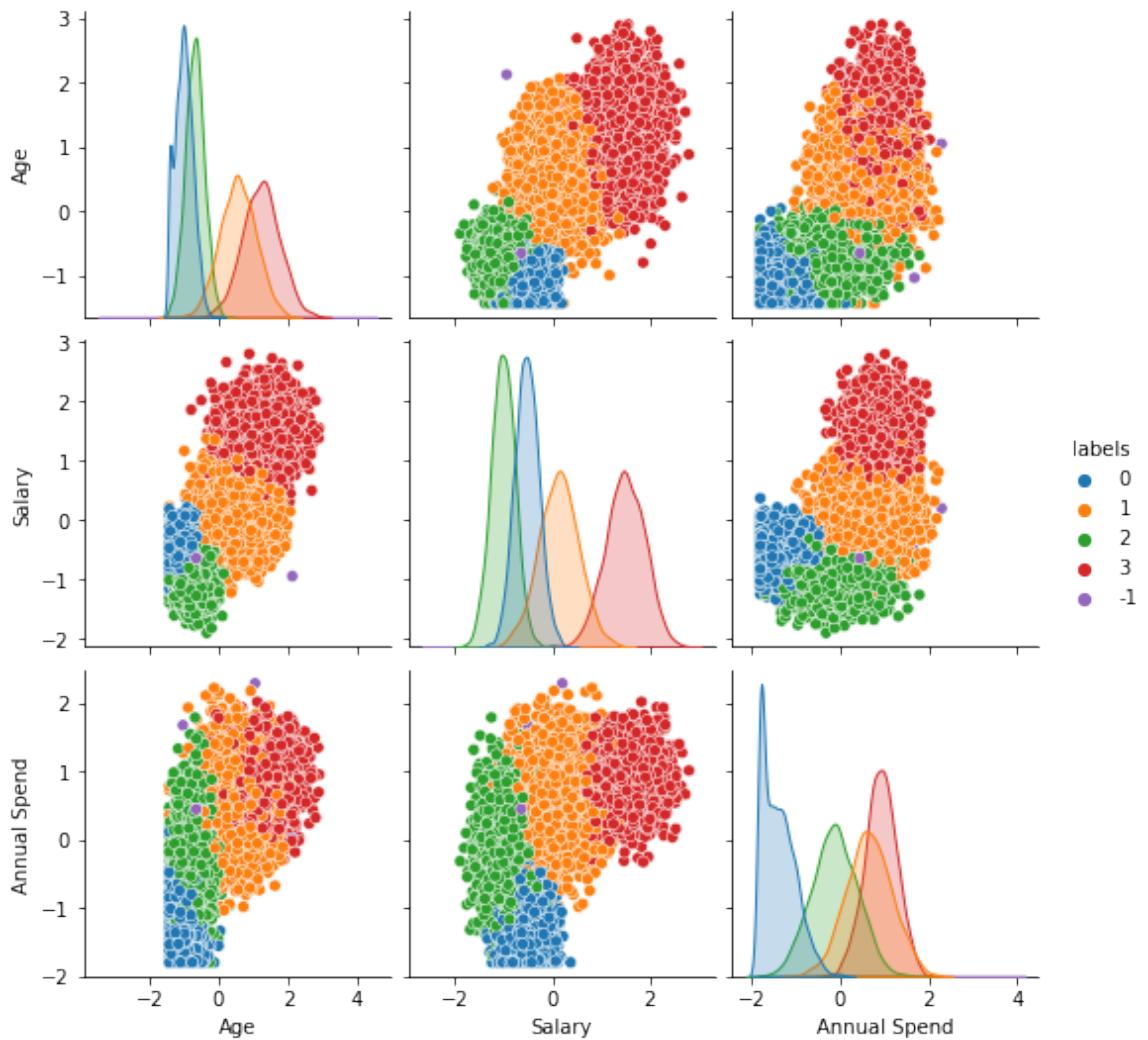


Figure 10: sns Pairplot for DBSCAN clustering

 Tutor provided code

```
opt = OPTICS(min_samples=50)
opt.fit(standardized_customer_data_df)
```

```
OPTICS(min_samples=50)
```

 Tutor provided code

```
labels = opt.labels_
print(f"labels = {labels}")

label_set = set(labels)
print(f"label_set = {label_set}")

label_list= list(labels)
print(f"label_list[0:10] = {label_list[0:10]}")

noise_point_count = label_list.count(-1)
print(f"noise_point_count = {noise_point_count}")
```

```
labels = [0 2 1 ... 0 1 0]
label_set = {0, 1, 2, 3}
label_list[0:10] = [0, 2, 1, 0, 0, 1, 2, 0, 2, 1]
noise_point_count = 0
```

To quote from <https://www.geeksforgeeks.org/ml-optics-clustering-explanation/>

“OPTICS (Ordering Points To Identify the Clustering Structure) is a density-based clustering algorithm, similar to DBSCAN (Density-Based Spatial Clustering of Applications with Noise), but it can extract clusters of varying densities and shapes. It is useful for identifying clusters of different densities in large, high-dimensional datasets.

The main idea behind OPTICS is to extract the clustering structure of a dataset by identifying the density-connected points. The algorithm builds a density-based representation of the data by creating an ordered list of points called the reachability plot. Each point in the list is associated with a reachability distance, which is a measure of how easy it is to reach that point from other points in the dataset. Points with similar reachability distances are likely to be in the same cluster.”

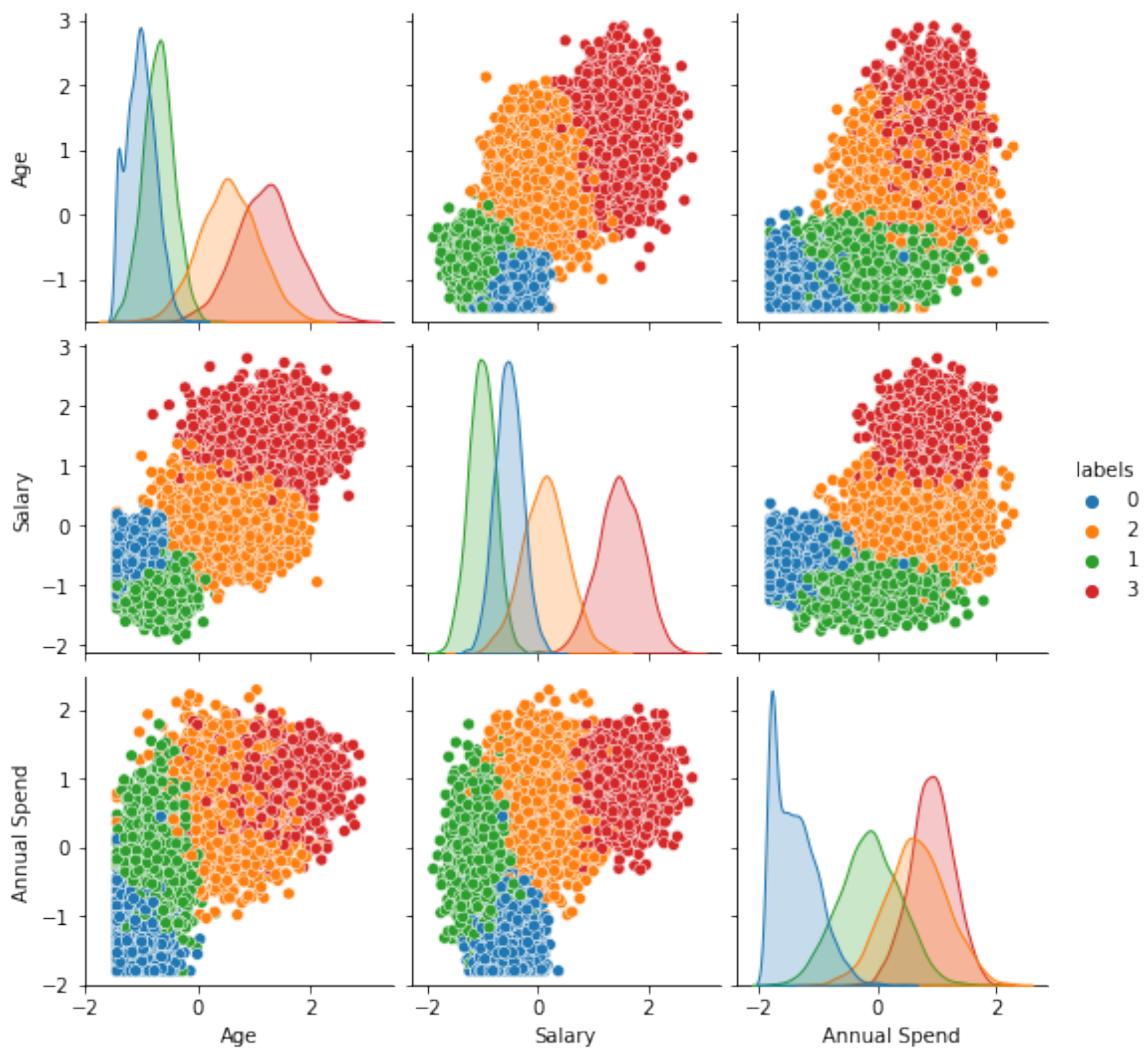


Figure 11: sns Pairplot for OPTICS clustering

Chapter 7 - Decision Trees, Ensemble Methods and Hyper-parameter Tuning

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2023-04-11

Table of contents

7 Decision Trees, Ensemble Methods and Hyper-parameter Tuning	7 - 3
7.1 Introduction	7 - 3
7.2 Instructions for Students	7 - 3
7.3 Loading library functions	7 - 5
7.4 Read the Data	7 - 5
7.5 Building the Decision Tree Model	7 - 9
7.6 Displaying the Tree as a Graphic	7 - 12
7.6.1 Pruning decision trees	7 - 12
7.7 3 Random Forrest Classifier	7 - 14
7.8 Random Forrest Classifier Performance	7 - 16
7.9 Hyper-parameter Tuning	7 - 19
7.9.1 Manually searching for the best hyper-parameters	7 - 19
7.9.2 Automated search of optimum hyper-parameters	7 - 21

List of Figures

1 The young Edward Deshion - Considered one of the great mathematical geniuses of his age demonstrating his own invention of the ‘Simulmathic Symphony’: “A grandiose assembly of mathematical endeavors, wherein the artful coordination of numbers takes center stage”	7 - 4
2 Bar-chart showing relative numbers within each category of outcome	7 - 7
3 Bar-chart showing number of pregnancies	7 - 7
4 Scatter Matrix for Glucose, BloodPressure, BMI and Age	7 - 10
5 Correlation Matrix for the Diabetes Data-set	7 - 11
6 Decision tree for the Diabetes Data-set	7 - 13
7 Pruned decision tree for the Diabetes Data-set	7 - 14

8	Confusion Matrix for a Random Forrest Classier applied to the Diabetes data-set	7 - 17
9	Confusion Matrix for a Decision Tree Classier applied to the Diabetes data-set	7 - 18
10	Confusion Matrix for a Random Forrest Classier applied to the Diabetes data-set	7 - 20

7 Decision Trees, Ensemble Methods and Hyper-parameter Tuning

7.1 Introduction

This tutorial starts with a treatment of ‘Decision Trees’. Decision Trees are an example of a supervised-learning, classification algorithm.

Decision Trees have a number of advantages and disadvantages. In terms of advantages, they are easy to understand and are ‘explainable’: A human being reviewing the model can understand how decisions are being made. The core algorithm is rather simple to explain and write: In class we use a spread-sheet implementation which demonstrates the algorithm step-by-step. Finally, the algorithm not only operates on numerical data - but can also be programmed to operate on category data directly (having said that, the specific library function we will be using from sklearn is restricted to numerical data only. But this is a limitation of the library function - not of the fundamental algorithm).

In terms of disadvantages, Decision Tree models can grow rather large. They also have a tendency to ‘over-fit’ data. That is, whilst the fit a given data-set they may not generalise so well.

To deal with the disadvantage of the tendency of decision trees to over-fit, we will look at a number of methods for dealing with that.

In this workshop we will work with a data-set relating to Diabetes. The data-set contains a number of descriptive features for individuals and a single output variable (label) that indicate if they acquired Diabetes. We will use that data to build a Decision-Tree model of the type that might be used to help predict the onset of Diabetes for individuals.

7.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding



Figure 1: The young Edward Deshion - Considered one of the great mathematical geniuses of his age demonstrating his own invention of the ‘Simulmathic Symphony’: “A grandiose assembly of mathematical endeavors, wherein the artful coordination of numbers takes center stage”

4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

7.3 Loading library functions

As always we start by loading key library functions:

- numpy : Conventionally named as ‘np’
- panda : Conventionally named as ‘pd’
- scatter_matrix : From pandas.plotting
- matplotlib.pyplot: Conventionally named as ‘plt’
- seaborn : Conventionally named as ‘sns’
- train_test_split : From sklearn.model_selection
- DecisionTreeClassifier : From sklearn.tree
- RandomForestClassifier : From sklearn.ensemble
- confusion_matrix : From sklearn.metrics
- ConfusionMatrixDisplay : From sklearn.metrics
- plot_confusion_matrix : From sklearn.metrics
- GridSearchCV : From sklearn.model_selection
- plot_tree : From sklearn.tree
- tree : From sklearn

 Your code here

7.4 Read the Data

Add code to load then print the data-set into a dataframe called ‘df’. The name of the data file for this workshop is ‘diabetes.csv’

 Your code here

```
df.iloc[0:5,0:5] =
Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin
```

```

0      6    148      72      35      0
1      1     85       66      29      0
2      8    183      64       0      0
3      1     89       66      23     94
4      0    137      40      35    168
df.iloc[0:5,6:] =
   DiabetesPedigreeFunction  Age  Outcome
0                  0.627    50       1
1                  0.351    31       0
2                  0.672    32       1
3                  0.167    21       0
4                 2.288    33       1

```

In this data-set the label is the ‘Outcome’ feature. The label is set to ‘1’ if the person developed diabetes and ‘0’ otherwise.

When building classification models we will want to check if the data is reasonably ‘balanced’. That is, are there sufficient numbers of each class to build a reasonable statistical model. It is therefore useful at this point to produce a bar-chart that counts the number of observations in each class.

Such a bar-chart is easily drawn using the ‘sns.countplot()’ function. Set the ‘x’ parameter to “Outcome” and the ‘data’ parameter to ‘df’

 Your code here

There are a smaller number of people who developed diabetes than people who are well. But still, there are sufficient number of positive cases to build a reasonably predictive model.

We can review other features in the same way.

Substitute various features from the data-set into the sns.countplot() function. That is, replace ‘Pregnancies’ with ‘Age’ etc. Think about which of these you find useful charts and which you do not.

 Your code here

As in previous workshops we can obtain an overall ‘statistical’ view of the data using the Pandas ‘describe’ function:

 Your code here

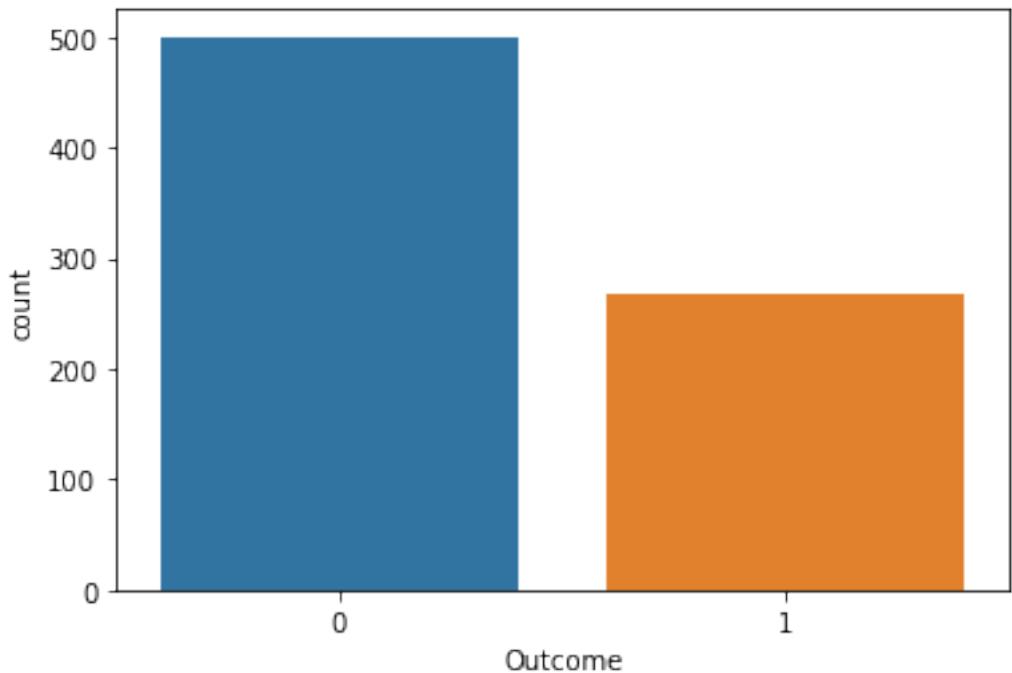


Figure 2: Bar-chart showing relative numbers within each category of outcome

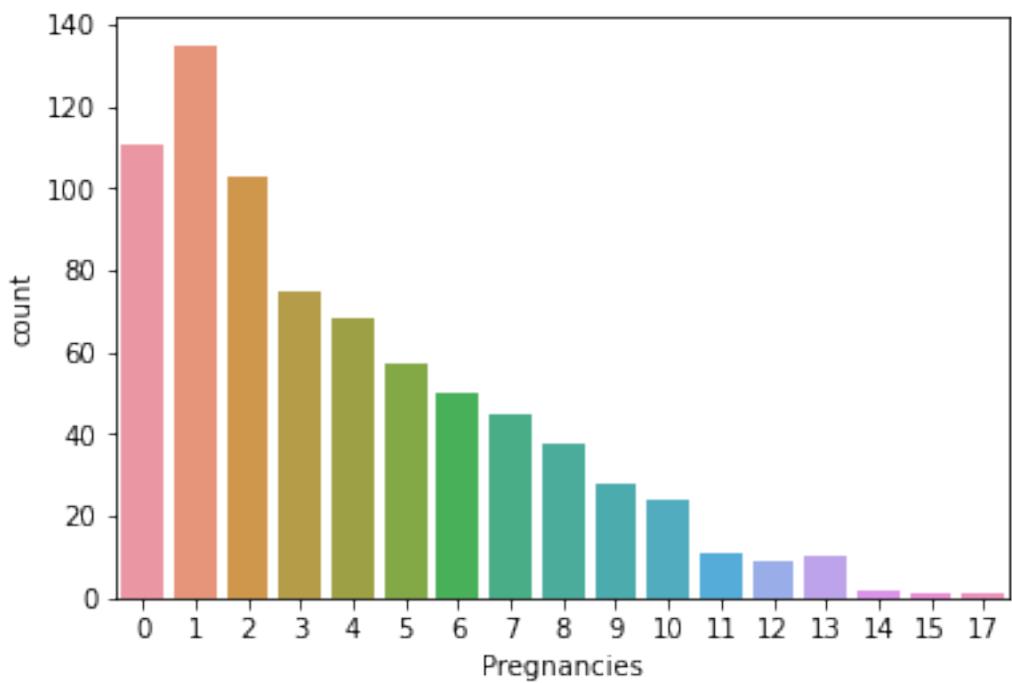


Figure 3: Bar-chart showing number of pregnancies

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin
count	768.0	768.0	768.0	768.0	768.0
mean	3.8	120.9	69.1	20.5	79.8
std	3.4	32.0	19.4	16.0	115.2
min	0.0	0.0	0.0	0.0	0.0
25%	1.0	99.0	62.0	0.0	0.0
50%	3.0	117.0	72.0	23.0	30.5
75%	6.0	140.2	80.0	32.0	127.2
max	17.0	199.0	122.0	99.0	846.0

	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.0	768.0	768.0	768.0
mean	32.0	0.5	33.2	0.3
std	7.9	0.3	11.8	0.5
min	0.0	0.1	21.0	0.0
25%	27.3	0.2	24.0	0.0
50%	32.0	0.4	29.0	0.0
75%	36.6	0.6	41.0	1.0
max	67.1	2.4	81.0	1.0

On reviewing this data we can see that certain values seem to be missing. For example the lower ‘Blood Pressure’ measurement is zero! There are also zero values for ‘SkinThickness’, ‘Insulin’ and ‘BMI’ - none of which can be correct.

We will fix this as a two stage process; First replace all of the zero values with ‘NaN’ values:

💡 Your code here

```
df.loc[0:5,'SkinThickness':'BMI'] =
   SkinThickness  Insulin  BMI
0          35.0      nan  33.6
1          29.0      nan  26.6
2          nan       nan  23.3
3          23.0     94.0  28.1
4          35.0    168.0  43.1
5          nan       nan  25.6
```

then use the built-in Pandas function ‘fillna’ to replace NaN values with imputed Mean values.

💡 Your code here

```
df.isnull().sum() =  
Pregnancies          0  
Glucose              0  
BloodPressure         0  
SkinThickness         0  
Insulin              0  
BMI                  0  
DiabetesPedigreeFunction 0  
Age                  0  
Outcome              0  
dtype: int64
```

As previously, review the data distributions using ‘scatter_matrix’:

💡 Your code here

And review the correlation matrix:

💡 Your code here

7.5 Building the Decision Tree Model

The first model we will create is a basic ‘Decision Tree’ using the sklearn library.

Add code to split the data into inputs (x’s) and label (y). The ‘y’ values are the label column ‘Outcome’. All other columns should be copied into a dataframe called ‘x’.

💡 Your code here

```
x.iloc[0:5,0:4] =  
    Pregnancies  Glucose  BloodPressure  SkinThickness  
0            6     148.0           72.0        35.0  
1            1      85.0           66.0        29.0  
2            8     183.0           64.0        29.2  
3            1      89.0           66.0        23.0  
4            0     137.0           40.0        35.0  
x.iloc[0:5,5:] =
```

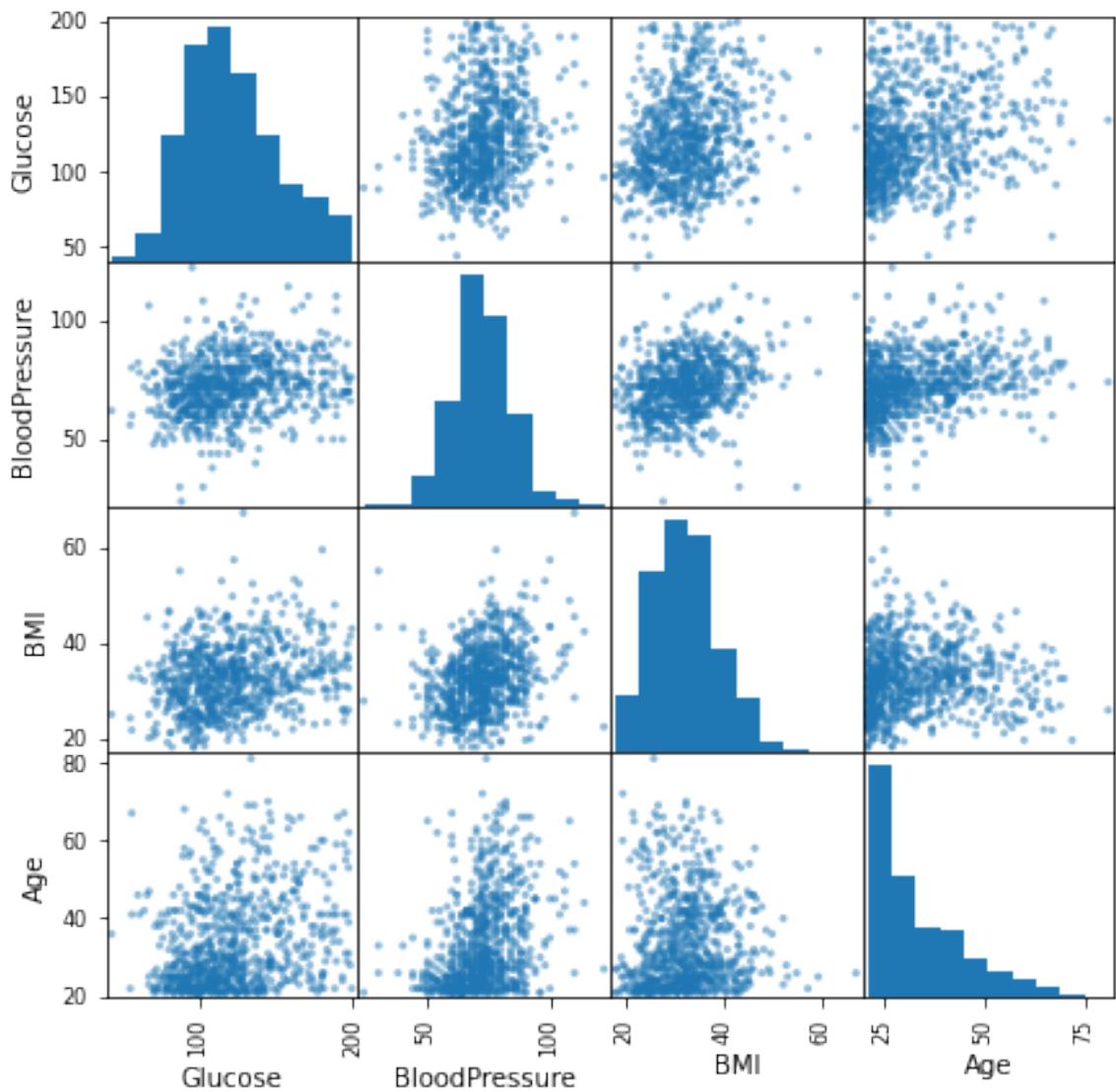


Figure 4: Scatter Matrix for Glucose, BloodPressure, BMI and Age

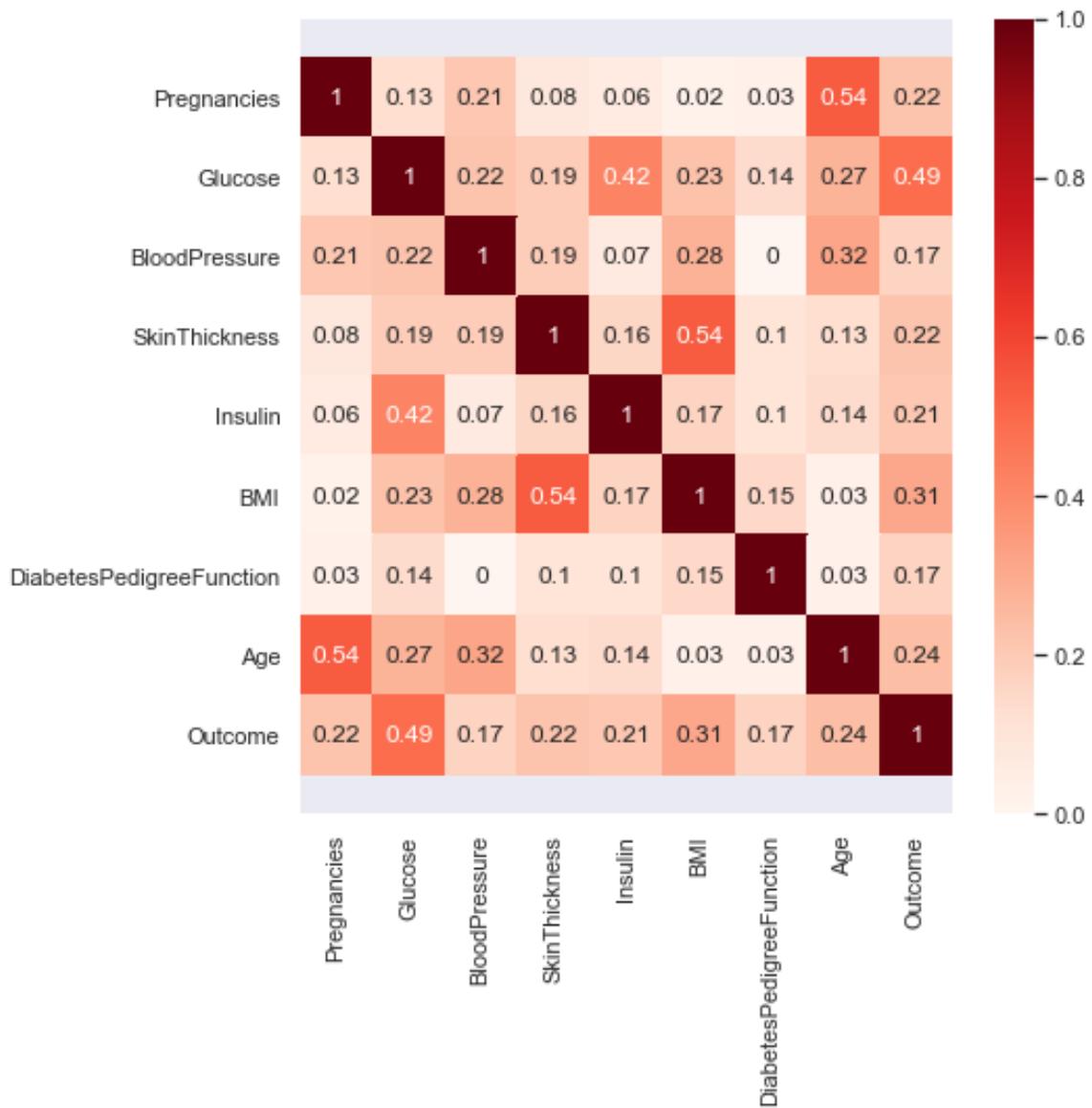


Figure 5: Correlation Matrix for the Diabetes Data-set

	BMI	DiabetesPedigreeFunction	Age
0	33.6	0.6	50
1	26.6	0.4	31
2	23.3	0.7	32
3	28.1	0.2	21
4	43.1	2.3	33

Instantiate a model object called ‘dtc’ as an instance of the sklearn ‘DecisionTreeClassifier’ class

 Your code here

Fit the data using the model:

 Your code here

```
DecisionTreeClassifier(max_depth=3, random_state=0)
```

7.6 Displaying the Tree as a Graphic

For smaller trees it can be informative to view the decision tree as a graphic. In practice, I have only found this useful for explanatory and tutorial purposes; most trees are too large to display easily and rather complicated to navigate.

 Tutor provided code

```
plt.figure(figsize=(20, 10))
tree.plot_tree(dtc,
               filled=True,
               fontsize=12,
               feature_names = df.columns)
plt.show()
```

Experiment by changing the ‘max_depth = 3’ to some other values.

Note : If the max depth gets too large then the graphic may too large to display. Also, it may take a long time to generate the graphic.

7.6.1 Pruning decision trees

The ‘max_depth’ parameter provides one method for preventing over-fitting in Decision Trees. It simply prevents the tree from growing below a certain depth. It divides the data

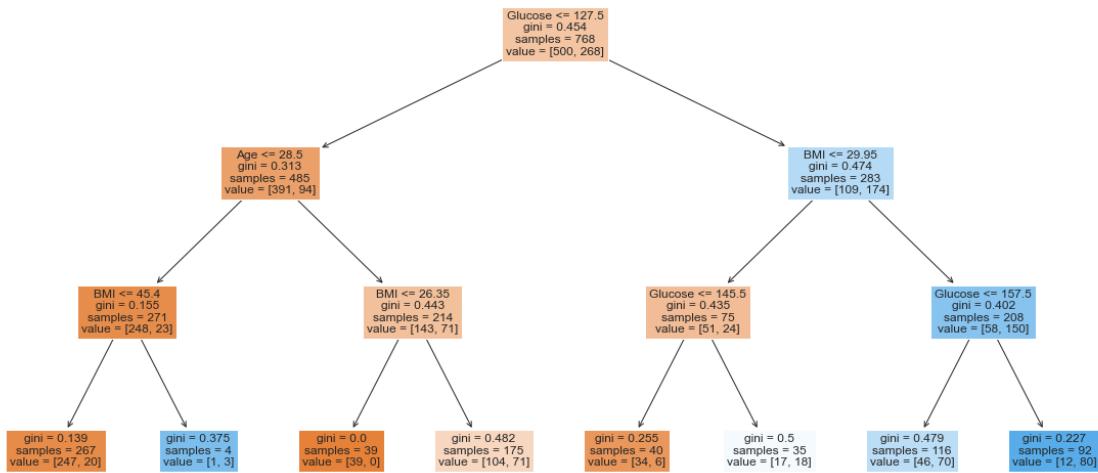


Figure 6: Decision tree for the Diabetes Data-set

set to a certain degree, choosing the most effective decision at each point .. but when the tree gets to a defined depth it just stops.

A more sophisticated way of preventing over-fitting is to ‘prune’ the tree. This means that nodes (branches) on tree are successively removed after the tree is built. The nodes are removed which causes the smallest loss of information and decision making power.

Describing the pruning algorithm in detail is beyond the scope of this course. However, if you are interested in this, after the workshop you may wish to review:

https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html#sphx-glr-auto-examples-tree-plot-cost-complexity-pruning-py

Create a new code block that replicates the model-building processes in the previous cells. However, this time add a parameter ‘ccp_alpha = 0.01’ to the instantiation of dtc (the decision tree classifier model). ‘CCP_alpha’ is a ‘regularisation parameter’ and it controls the level of pruning.

You should increase the value of ‘max_depth’ to 6.

💡 Your code here

You may see that the tree is no longer a simple binary tree. The tree is deeper on one side as compared with the other. This is simply because ‘weak’ nodes have been removed in the pruning.

Now experiment with different values for ‘max_depth’ and ‘ccp_alpha’.

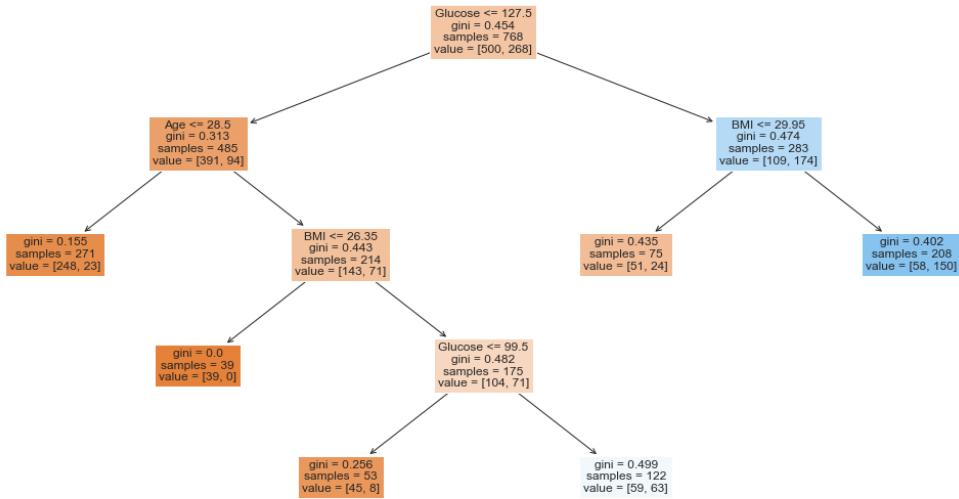


Figure 7: Pruned decision tree for the Diabetes Data-set

7.7 3 Random Forrest Classifier

One way of extending the basic Decision Tree model is to build a ‘forrest’. That is, build a collection of Decision Trees, each based on a slightly different sub-set of the data.

In this section, we will build just such a ‘Random Forrest Classifier’ and then compares its performance with the basic Decision Tree.

In order to do this, you should split the data-set into a training and a test set as explained in previous workshops. As previously, your results will not match those here as the training and test sets are selected randomly.

Your code here

```

x_train.iloc[0:5, 0:4] =
    Pregnancies  Glucose  BloodPressure  SkinThickness
728          2      175.0           88.0        29.2
713          0      134.0           58.0        20.0
625          4      90.0            88.0        47.0
597          1      89.0            24.0        19.0
520          2      68.0            70.0        32.0
x_train.iloc[0:5, 5:] =
    DiabetesPedigreeFunction  Age
728                  0.3   22
713                  0.4   21
625                  0.4   29

```

```

597          0.6   21
520          0.2   25
x_test.iloc[0:5, 0:4] =
    Pregnancies  Glucose  BloodPressure  SkinThickness
266           0     138.0            72.4        29.2
446           1     100.0            72.0        12.0
535           4     132.0            72.4        29.2
590          11     111.0            84.0        40.0
756           7     137.0            90.0        41.0
x_test.iloc[0:5, 5:] =
    DiabetesPedigreeFunction  Age
266                  0.9   25
446                  0.7   28
535                  0.3   23
590                  0.9   45
756                  0.4   39
y_train.iloc[0:5,] =
728      0
713      0
625      0
597      0
520      0
Name: Outcome, dtype: int64
y_train.iloc[0:5,] =
728      0
713      0
625      0
597      0
520      0
Name: Outcome, dtype: int64
y_test.iloc[0:5,] =
266      1
446      0
535      1
590      1
756      0
Name: Outcome, dtype: int64
y_test.iloc[0:5,] =
266      1
446      0
535      1
590      1
756      0
Name: Outcome, dtype: int64

```

 Your code here

7.8 Random Forrest Classifier Performance

Now do the following:

- Add code to instantiate an object called ‘rfc’ based on the sklearn ‘RandomForestClassifier()’ class.
- Fit the model using the training data
- Score the model using the sklearn method ‘score’ passing the ‘x_test’ and ‘y_test’ data as parameters.

 Your code here

Random Forrest Classifier Score = 0.792

Or for a better view of performance, plot the whole Confusion Matrix:

 Tutor provided code

```
predictions = rfc.predict(x_test)
cm = confusion_matrix(y_test, predictions, labels=rfc.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=rfc.classes_)
disp.plot()
plt.show()
```

Add a code block to build a Decision Tree Classifier model on the same training data and test it using the same test data.

dtc score = 0.72

 Your code here

And again, plot the confusion matrix.

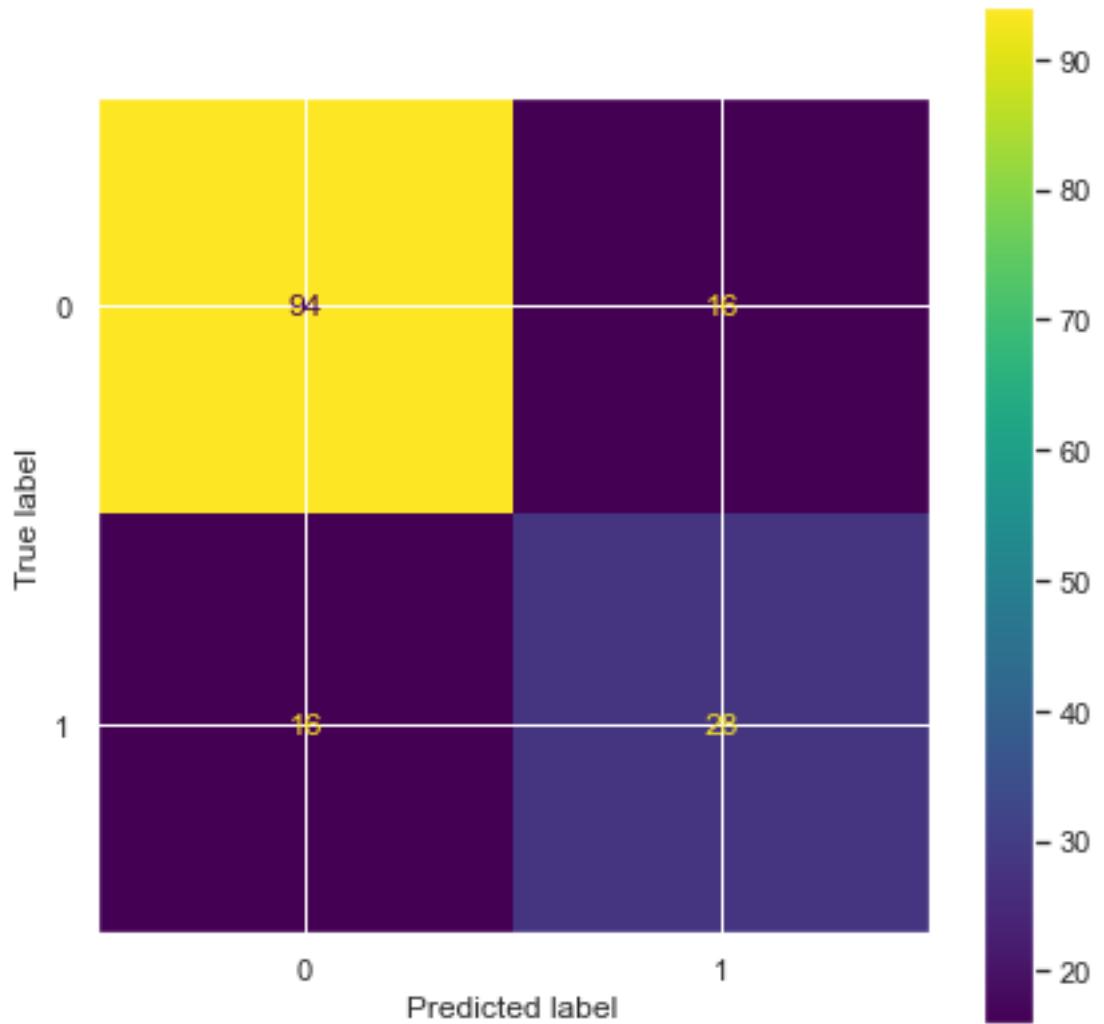


Figure 8: Confusion Matrix for a Random Forrest Classifier applied to the Diabetes data-set

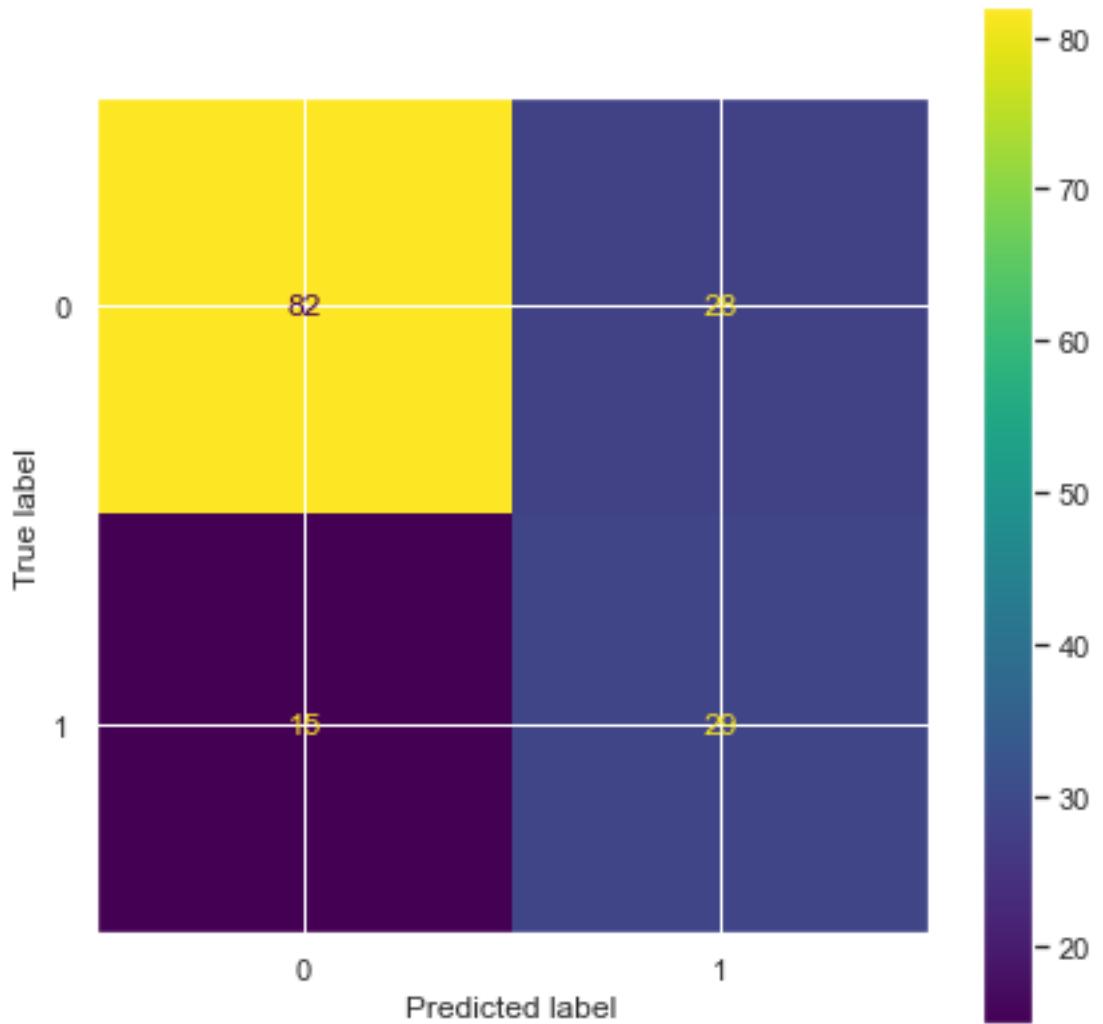


Figure 9: Confusion Matrix for a Decision Tree Classifier applied to the Diabetes data-set

7.9 Hyper-parameter Tuning

Many Machine Learning model algorithms have a large set of ‘Hyper-parameters’. That is, there are settings that change the way that the model is built. You saw an example of this in the description of the Decision Tree Classifier above. In that case the hyper-parameter ‘ccp_alpha’ controlled the amount of pruning.

An indicator of the number of hyper-parameters available for a model can be obtained using the ‘get_params()’ function which is available for some sklearn algorithms. For example, for our Random Forrest Classifier:

💡 Your code here

```
rfc.get_params() =  
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth':
```

7.9.1 Manually searching for the best hyper-parameters

Add a code block there with another version of the RandomForestClassifier. In this case add some of the other available hyper-parameters.

Experiment with various settings of the Hyper-parameters of ‘RandomForestClassifier’ below to see if you can produce a better model. Typical values for the hyper-parameters are as follows:

- ccp_alpha : a number between 0 and 1
- criterion : ‘entropy’ or ‘gini’
- max_depth : A number greater than 0
- max_leaf_nodes : A number greater than 1

💡 Your code here

An important insight can be gained here by computing the ‘score’ for the above model:

💡 Your code here

```
rfc.score(x_test, y_test) = 0.73
```

When I ran this code I obtained a score of 0.66 .. not bad! However, it is clear from the confusion matrix that the model has taken the simple expedient of labelling every case as ‘0’. The result just reflects original imbalance in classes in the data-set (see above). This is

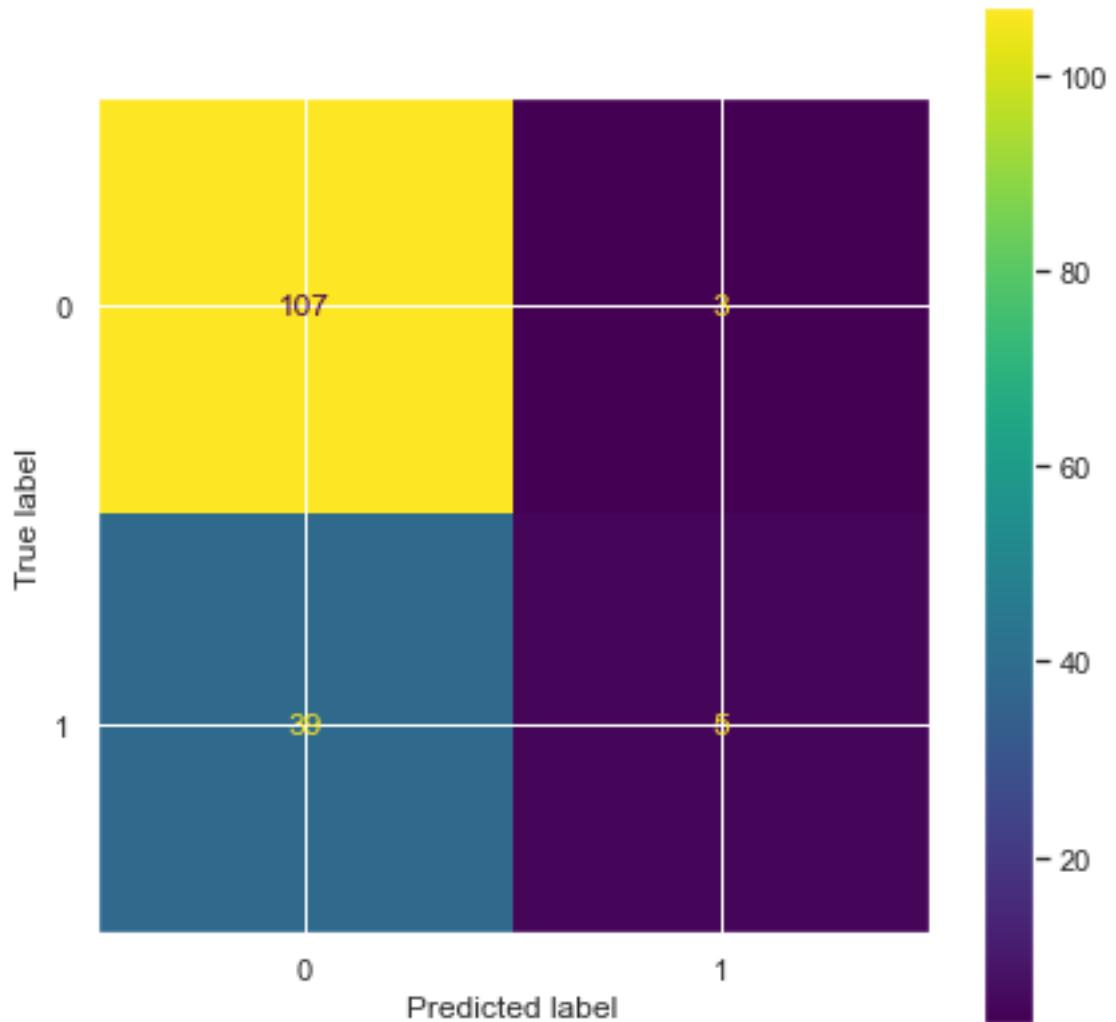


Figure 10: Confusion Matrix for a Random Forrest Classier applied to the Diabetes data-set

one of the dangers of an un-balanced data-set .. a ‘score’ may produce a rather misleading result.

The lessons to be learnt here are:

- Wherever possible obtain a data-set that is well balanced between different classifications
- At the very least *be aware* of any imbalance and the impact it may have
- Don’t rely only on a ‘score’ quality metric - it can be misleading

7.9.2 Automated search of optimum hyper-parameters

Experimenting with many parameters, each with many potential settings is time consuming. Additionally, there are further risks of over-fitting on the hyper-parameters since the same data is being used repeatedly for testing.

Alternatively, there is an automated method within sklearn for searching for optimum hyper-parameters.

First, a list is created that identifies the hyper-parameters and the range of values that should be applied to each:

 Tutor supplied code

```
param_grid = [  
    {'ccp_alpha':[0.0, 0.005, 0.007, 0.01, 0.015, 0.02, 0.03, 0.05, 0.1],  
     'criterion':['gini', 'entropy'],  
     'max_depth': [2,3,4,5,6,7,8,9],  
     'max_leaf_nodes' : [2,3,4,5,6,7,8,9]}  
]
```

Then use the ‘GridSearchCV’ function to try each permutation of hyper-parameter until the optimum settings are discovered.

WARNING This operation may take a considerable amount of time, since the model has to be built for each combination of parameter settings. On my CORE i7-7700HQ laptop it takes about 20 minutes to run! So if you have a slower machine - you may be waiting a long time!

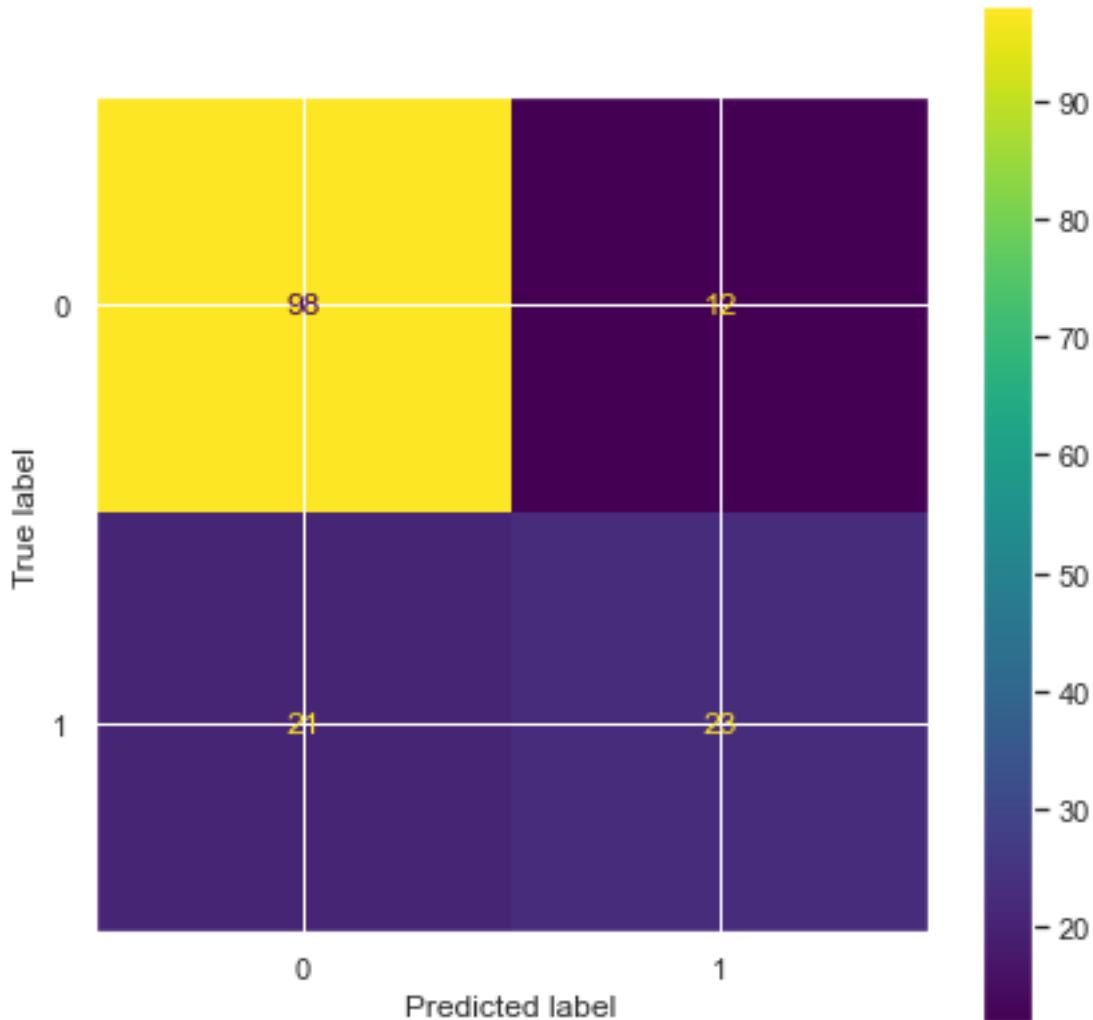
 Tutor supplied code

```
clf = GridSearchCV(estimator=rfc,  
                    param_grid=param_grid)  
best_model= clf.fit(x_train,y_train)
```

 Tutor supplied code

```
predictions = clf.predict(x_test)
cm = confusion_matrix(y_test, predictions, labels=rfc.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=clf.classes_)
disp.plot()
plt.show()
```

Review the confusion matrix for the model with optimum settings:



Useful though the confusion Matrix is, you will often see other specific metrics for classification models. A useful method within sklearn called ‘classification_report’ provides an easy-to-use and clearly presented display of the most useful metrics.

You can learn more about ‘classification_report’ at the following websites:

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
- <https://www.statology.org/sklearn-classification-report/>

Add a code block to display the classification_report for the optimised model generated above.

💡 Your code here

```
precision    recall   f1-score   support\n\n          0       0.82      0.89      0.86      110\n          1       0.66      0.52      0.58       44\n\n   accuracy                           0.79      154\n  macro avg       0.74      0.71      0.72      154\nweighted avg       0.78      0.79      0.78      154\n\nbest_model.score(x_test, y_test) = 0.79
```

Chapter 8 - Dimensionality Reduction

No-Code Version for Student Practical Classes

(c) Dr Rob Collins 2023

2023-04-11

Table of contents

8 Dimensionality Reduction	8 - 3
8.1 Introduction	8 - 3
8.2 Instructions for Students	8 - 3
8.3 Demonstration of low-dimensional data being embedded in a higher-dimensional representation	8 - 4
8.4 Principle Component Analysis (PCA) for MNIST	8 - 7
8.5 Choosing the number of components	8 - 12
8.6 PCA as Noise Filtering	8 - 13
8.7 PCA on Facial Image Data	8 - 16
8.8 References:	8 - 19

List of Figures

1 “One of the many attempts to build a true ‘Machinum Reductum Dimesionitas’ - one of the 23 ‘Ultimum Problema’ established as a requirement for the true path to enlightenment by the 11th-Century monk Hilbertus”	8 - 2
2 3D Plot of 2-Dimensional Data with Random Gaussian Noise	8 - 6
3 3D Plot of MNIST data-set after PCA reduction to 3 dimensions	8 - 10
4 2D Plot of MNIST data-set after PCA reduction to 2 dimensions	8 - 11
5 Cumulative Explained Variance of each component in the MNIST data-set after PCA	8 - 12
6 Set of original MNIST digits before addition of noise	8 - 14
7 Set of MNIST digits with the addition of Gaussian Noise	8 - 15
8 Noisy version of MNIST digits after filtering using PCA	8 - 15
9 Eigen Faces for the the Labeled Faces in the Wild database	8 - 17
10 Cumulative variance plot for Eigen Faces	8 - 18
11 Comparing faces before and after dimensionality reduction	8 - 18

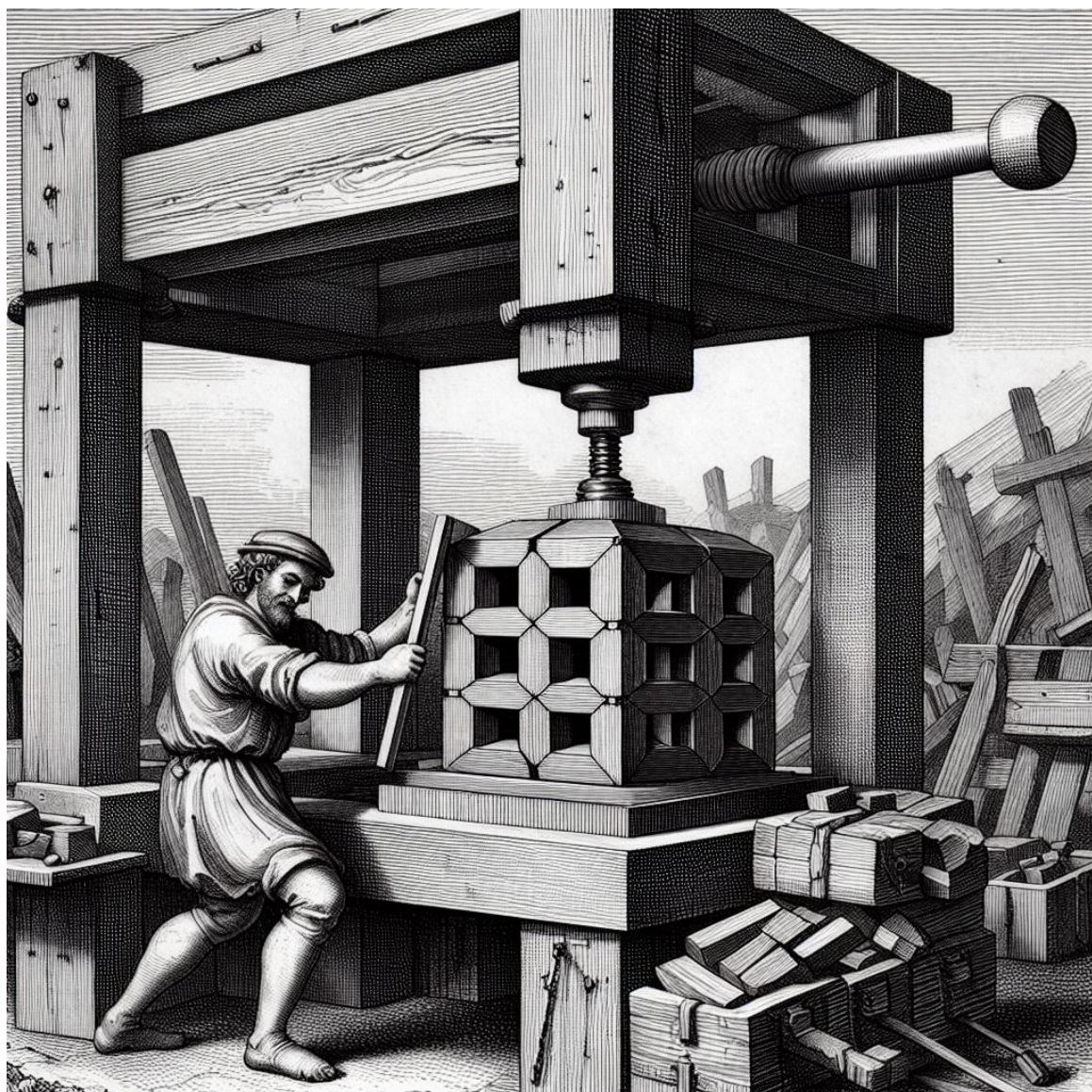


Figure 1: “One of the many attempts to build a true ‘Machinum Reductum Dimesionitas’ - one of the 23 ‘Ultimum Problema’ established as a requirement for the true path to enlightenment by the 11th-Century monk Hilbertus”

8 Dimensionality Reduction

8.1 Introduction

This tutorial is focussed on ‘Dimensionality Reduction’. It is quite common in data-sets that although the data set may have many features (‘dimensions’), some of these features may be highly correlated. Another way of thinking about this is that using both features (‘dimensions’) provides little more information than using one of them alone.

Data may often appear to occupy many dimensions because of the way it is represented and stored. However, the data is in-fact of much lower dimensionality.

In this case it is useful to reduce the dimensionality of the data-set. That is to ‘squeeze’ the information from the data-set and represent it as data that has far fewer dimensions. The lower dimensional representation will be smaller to store and will often require significantly lower resources to process - since there is less data.

Sometimes, some of the dimensions in data really only represent random noise in the original data. Thus, dimensionality reduction can also achieve noise reduced. This will be illustrated in one of the examples below.

8.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class

8. After class, enter and run any remaining code blocks that you have not been able to complete in class

8.3 Demonstration of low-dimensional data being embedded in a higher-dimensional representation

This part of the workshop provides a visualisation that helps show how low-dimensional data might be ‘embedded’ in a higher dimensional representation.

First, as always, import the libraries needed for this part of the workshop:

- numpy as np
- matplotlib.pyplot as plt
- random as rand
- Axes3D : From mpl_toolkits.mplot3d
- import seaborn as sns
- PCA : From sklearn.decomposition
- load_digits : From sklearn.datasets
- fetch_lfw_people: From sklearn.datasets

 Your code here

 Your code here

Create a two-dimensional data set. Then add data in 3rd (z) dimension which is a simple linear function of the original 2-D data-set with a small amount of Gaussian noise.

Note : I have provided the code to achieve this below. However, by this point in the course you should be able to do this without the provided code. Challenge yourself to write the code yourself before reviewing my solution.

 Tutor supplied code

```
xGrad = 5  
yGrad = 7  
noiseSD = .05  
  
x = []  
y = []  
z=[]  
  
for i in range(100):  
    xItem = rand.randint(1,200)
```

```
yItem = rand.randint(1,200)
x.append(xItem)
y.append(yItem)
noise = np.random.normal(0, scale=50.0)
z.append((xItem * xGrad) + (yItem * yGrad) + noise)
```

The first part of each list will look something like this (although, of course, you will see different random numbers when you run the code):

💡 Your code here

```
x[0:5] = [119, 139, 183, 79, 113]
y[0:5] = [103, 195, 73, 140, 44]
z[0:5] = [1374.99, 2107.72, 1436.65, 1441.73, 863.60, ]
```

Now plot the data as a 3D chart. Note that you can **rotate the view of the chart by clicking and dragging** on it. As you do so, notice that whilst the data is represented in a 3D space - the points generally lie in a flat plane.

Any variation above and below the plane is just random ‘noise’ (error). This three dimensional representation provides no useful information that could not be represented in two dimensions.

Note : In my code I have included the directive:

```
%matplotlib inline
```

To force the chart to appear ‘inline’ in the notebook for display purposes. Your chart should appear as a ‘pop-out’ 3D window.

💡 Your code here

The interested reader may care to review the material in Chapter 5 on ‘Matrices and Linear Algebra’. In that chapter I describe the situation of low dimensional data existing within a higher-dimensional space. This is described in terms of the ‘rank’ of the matrix - the number of independent variables.

Because of the noise component in the above data-set we do truly have a matrix of rank 3. However, if there was some way we could remove that noise (or indeed if we had not injected it in the first place) then the data could be represented as in a matrix of rank 2 .. it would actually only be 2-dimensional.

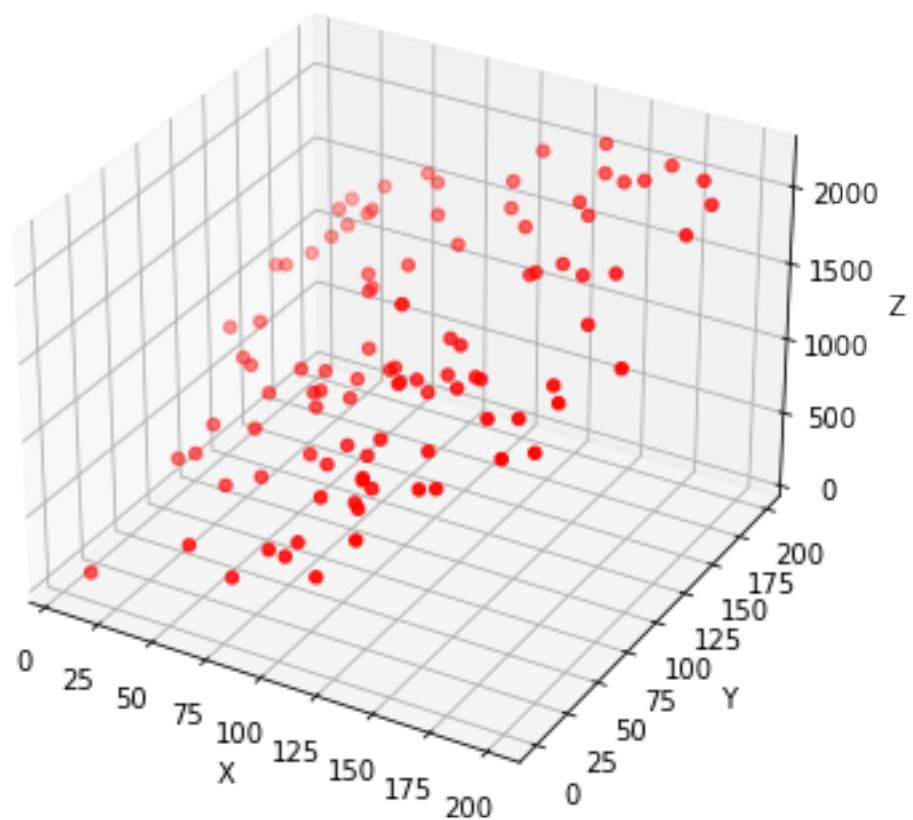


Figure 2: 3D Plot of 2-Dimensional Data with Random Gaussian Noise

8.4 Principle Component Analysis (PCA) for MNIST

In this section we will look at an algorithm that is used to reduce the dimensions of data sets.

In this case we will be using a series of images of handwritten digits. This data set is called MNIST and is frequently used in Machine Learning tutorials for such tasks as building tools for handwriting analysis.

An important thing to remember when working through the following section is that we are using an **unsupervised learning** algorythm. When we process the images of the digits we are not giving the algorithm examples of ‘the correct answer’. Rather, we are purely squeezing information from the raw data.

In this case, the data set we want to use is built into sklearn. That means that we can access it easily and directly.

 Tutor provided code

```
digits = load_digits()
```

Review the shape of the data ..

 Your code here

```
digits.data.shape = (1797, 64)
```

That is, there are 1797 images, each with **64 dimensions**. The dimensions represent an image which is 8x8 pixels in size. Each pixel represents a grey-scale.

Add a code block to print the first element of the data-set.

You can see that it consists of 64 numbers. Each number is in the range 0..255 and represents a ‘grey-scale’.

Experiment by changing the index in the code below to look at other digits in their raw form.

 Your code here

```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
      12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
      0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
     10., 12.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

Sklean provides a function that enables you to look at an image of the image represented by the data. The code to do this is included in the code-block below.

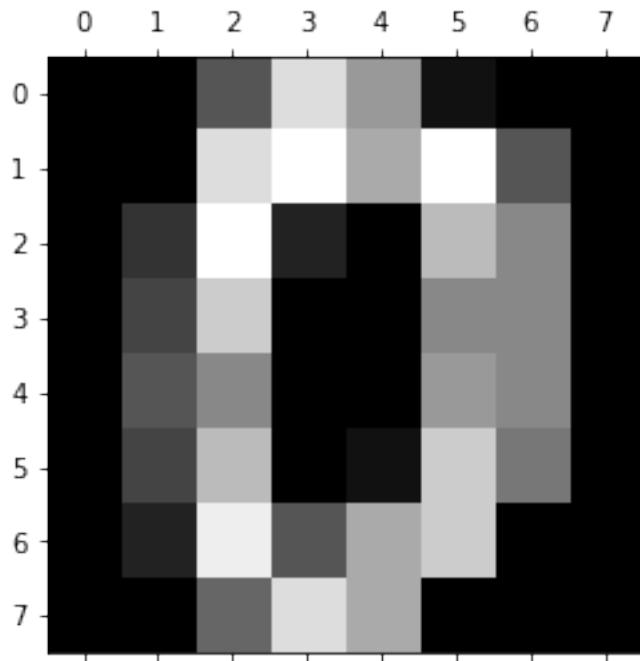
Experiment by changing the index in the code below to look at other digits as images.

 Tutor supplied code

```
plt.gray()  
plt.matshow(digits.images[0])  
plt.show()
```

<Figure size 432x288 with 0 Axes>

Visualisation of a single image from the MNIST data-set



Now, whilst the digits are represented in a 64 dimensional data structure .. there are not really 64 dimensions of real ‘information’ here. Hand-written digits are very constrained - they fall into a rather simple graphical pattern.

A 64 dimensional array of number in the range 0-255 can represent 10^{154} different data values! This means that the image representation of a digit is highly redundant.

We can reduce the number of dimensions for the data-set whilst still retaining much of the useful information.

The sklean ‘pca’ function can be used to perform a ‘Principle Component Analysis’. This transforms the data into a lower dimensional space. Dimensions are ordered so that the first

dimension contains the more information than any other dimension. The second dimension contains more information than the third .. and so on. Dimensions are thus ordered by the amount of information they contain.

We can reduce the number of dimensions in the MNIST data set from 64 to just 3 dimensions.

 Tutor supplied code

```
pca = PCA(3) #  
projected = pca.fit_transform(digits.data)
```

The resulting data is no longer 1797 rows with 64 dimensions .. but 1797 rows each with 3 dimensions...

 Your code here

```
digits.data.shape = (1797, 64)  
projected.shape = (1797, 3)
```

The raw data looks like this ..

 Your code here

```
array([[ -1.25946626,   21.27488552,  -9.46304764] ,  
       [  7.95761282,  -20.76869703,   4.43954015] ,  
       [  6.99192144,  -9.95598949,   2.95852974] ,  
       ... ,  
       [ 10.80128228,  -6.9602539 ,   5.59953785] ,  
       [-4.87209964,   12.42395769, -10.17084683] ,  
       [-0.34439144,   6.36554644,  10.77366824]])
```

Even in just 3 dimensions there is still enough information to do quite a good job of separating the image data into clusters for each digit. We can see this by plotting a 3D chart. In this chart, each point has a colour depending on the value of the original digit 0, 1 .. 9.

 Tutor provided code

```
x = projected[:,0]  
y = projected[:,1]  
z = projected[:,2]
```

```

fig2 = plt.figure(figsize=(6,6))
ax2 = fig2.add_subplot(111, projection='3d')

ax2.scatter(x, y,z,
            c=digits.target,
            cmap=plt.cm.get_cmap('Spectral', 10))

ax2.set_xlabel('X Label')
ax2.set_ylabel('Y Label')
ax2.set_zlabel('Z Label')

fig2.show()

```

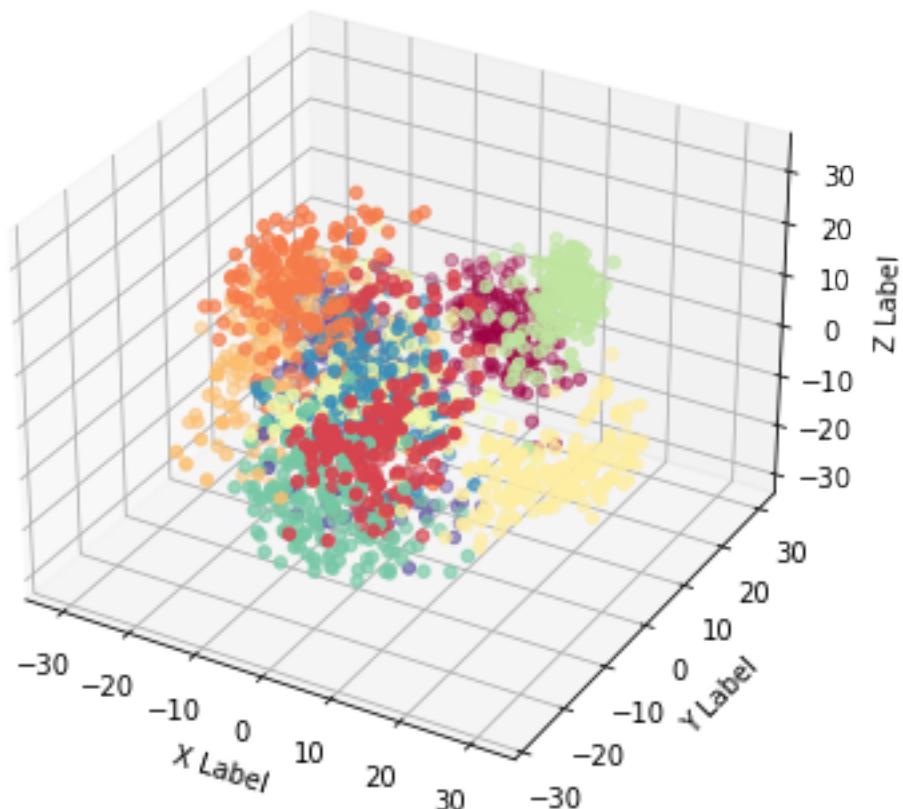


Figure 3: 3D Plot of MNIST data-set after PCA reduction to 3 dimensions

We can even do the same thing with just 2 dimensions:

💡 Your code here

And again, plot the results:

💡 Your code here

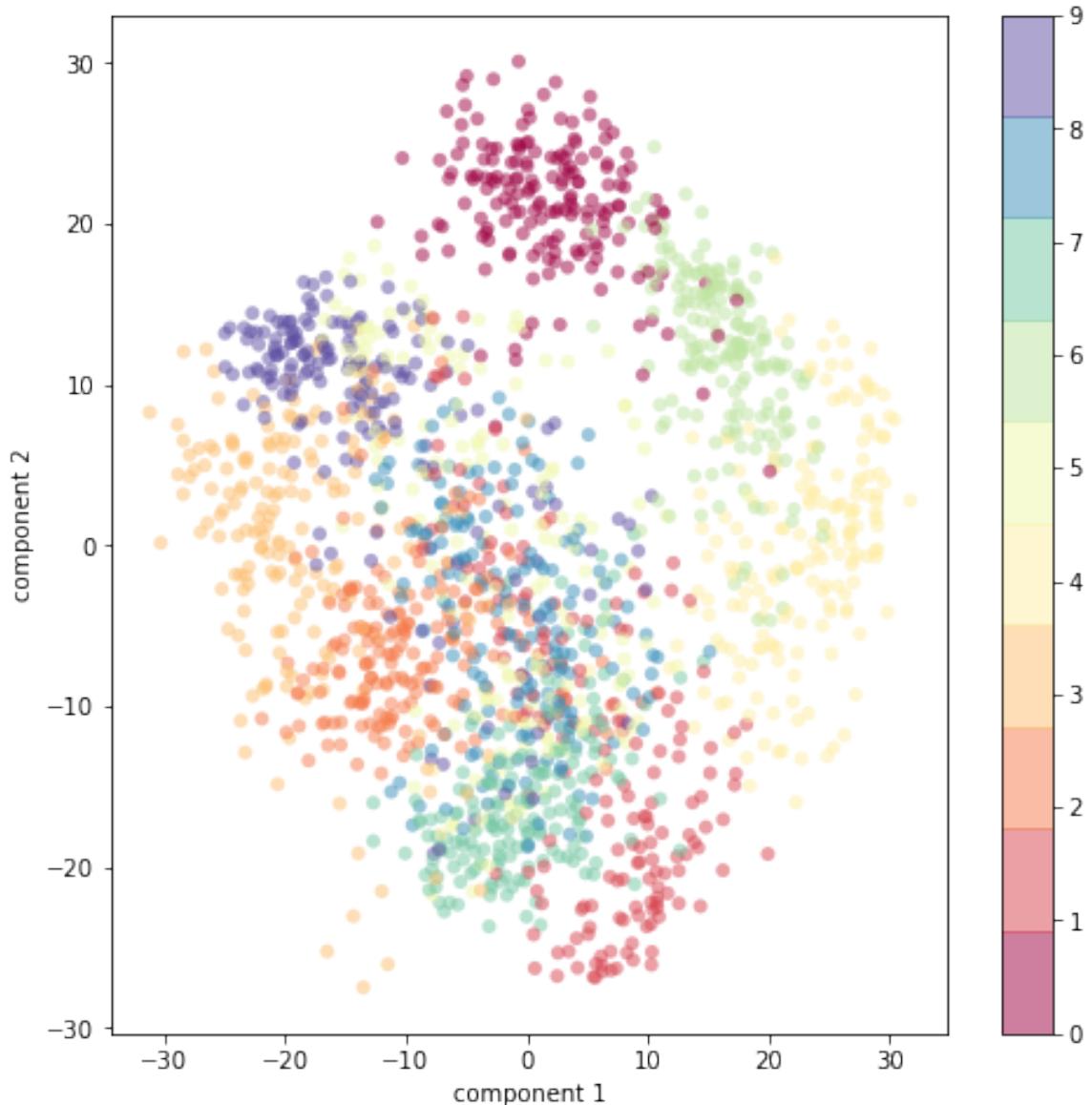


Figure 4: 2D Plot of MNIST data-set after PCA reduction to 2 dimensions

So even with just 2 dimensions, there seems to be enough information to separate the digits into clusters.

8.5 Choosing the number of components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. This can be determined by looking at the cumulative explained variance ratio as a function of the number of components:

It is interesting to see how much information is contained in each of the dimensions generated through PCA. The following chart shows the cumulative total of information provided by each dimension ('component'). From the graph you will see that over 70% of the variation is explained by just 10 components ('dimensions').

 Tutor provided code

```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```

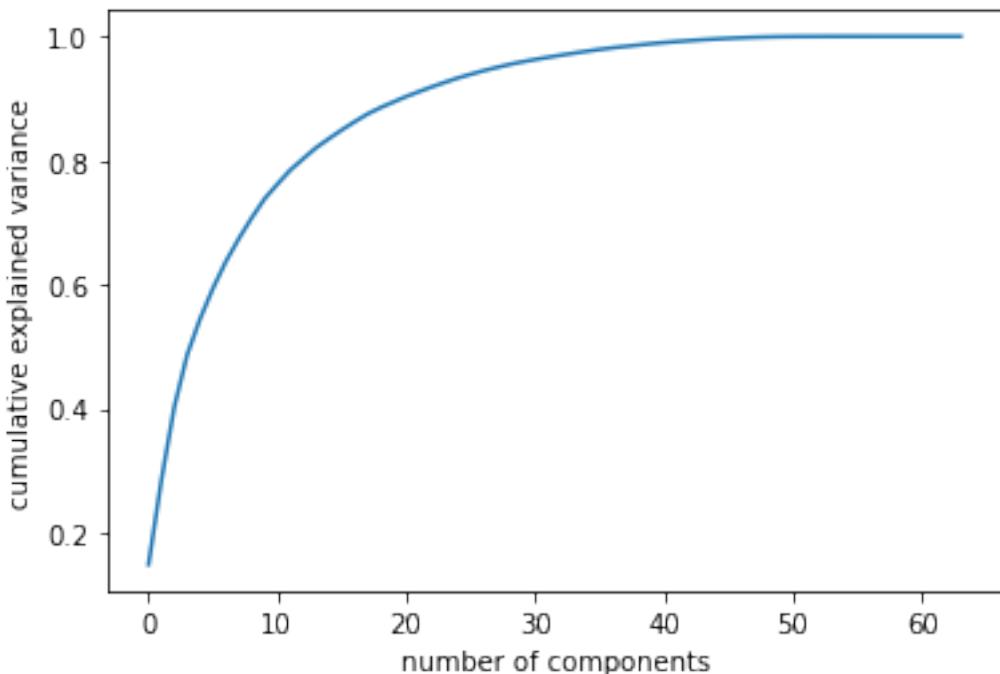


Figure 5: Cumulative Explained Variance of each component in the MNIST data-set after PCA

The numerical values for the Cumulative Explained Variance can be obtained using the 'np.cumsum()' method:

💡 Your code here

```
np.cumsum(pca.explained_variance_ratio_) =  
[0.14890594 0.28509365 0.40303959 0.48713938 0.54496353 0.59413263  
0.6372925 0.67390623 0.70743871 0.73822677 0.76195018 0.78467714  
0.80289578 0.82063433 0.83530534 0.84940249 0.86258838 0.87506976  
0.88524694 0.89430312 0.9031985 0.91116973 0.91884467 0.9260737  
0.93303259 0.9389934 0.94474955 0.94990113 0.95479652 0.9590854  
0.96282146 0.96635421 0.96972105 0.97300135 0.97608455 0.97902234  
0.98158823 0.98386565 0.98608843 0.98820273 0.99010182 0.99168835  
0.99319995 0.99460574 0.99577196 0.99684689 0.99781094 0.99858557  
0.99914278 0.99954711 0.99975703 0.99983951 0.99989203 0.99994255  
0.99997555 0.99998798 0.99999503 0.99999804 0.99999911 0.99999966  
1. 1. 1. 1. ]
```

The plot and the numerical result can help you understand the level of redundancy present in higher dimensional data-sets.

With just 30 dimensions (less than half the total) you still obtain over 95% of the ‘real’ information (or ‘variance’ as it is properly called in the context of PCA).

It is also worth considering that the remaining 5% of ‘information’ may also actually be ‘noise’ in the data. That is .. you may need all of those extra dimensions simply to capture random errors in the data! It is not just that you have squeezed the amount of information into a smaller form, but you have also made a step towards extracting **useful** information.

This idea is explored in the next section.

8.6 PCA as Noise Filtering

As mentioned above, Principle Component Analysis (PCA) can also be a way to reduce ‘noise’ in data. The idea is that most of the information is contained within the first few dimensions (principle components). Noise is technically speaking ‘information’ but since it is random it will have a high degree of variance from the principle components. Thus, removing some of the dimensions with low information content will have the effect of reducing the noise.

We can test this idea as an experiment using the MNIST data-set again.

The following code displays some of the MNIST digits.

💡 Tutor provided code

```
def plot_digits(data):  
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
```

```

        subplot_kw={'xticks':[], 'yticks':[]},
        gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(data[i].reshape(8, 8),
              cmap='binary', interpolation='nearest',
              clim=(0, 16))
plot_digits(digits.data)

```

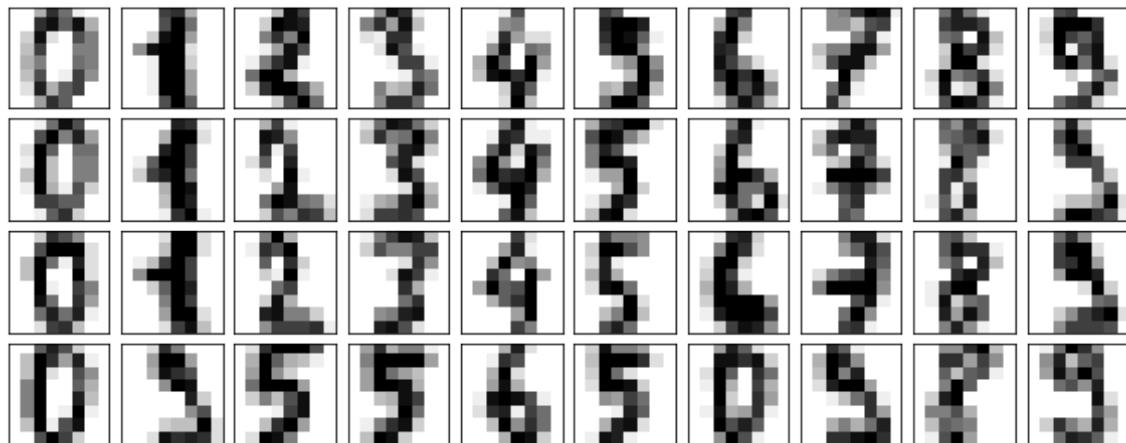


Figure 6: Set of original MNIST digits before addition of noise

Now we add some random noise to create a noisy dataset, and re-plot it:

Tutor provided code

```

noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)

```

It's clear by eye that the images are noisy, and contain spurious pixels.

Now apply to the noisy data, specifying that the projection preserve 50% of the variance:

Tutor provided code

```

pca = PCA(0.50).fit(noisy)
print(f"Number of components (pca.n_components_) = {pca.n_components_}")

```

Number of components (pca.n_components_) = 12

Here 50% of the variance amounts to 12 principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits:

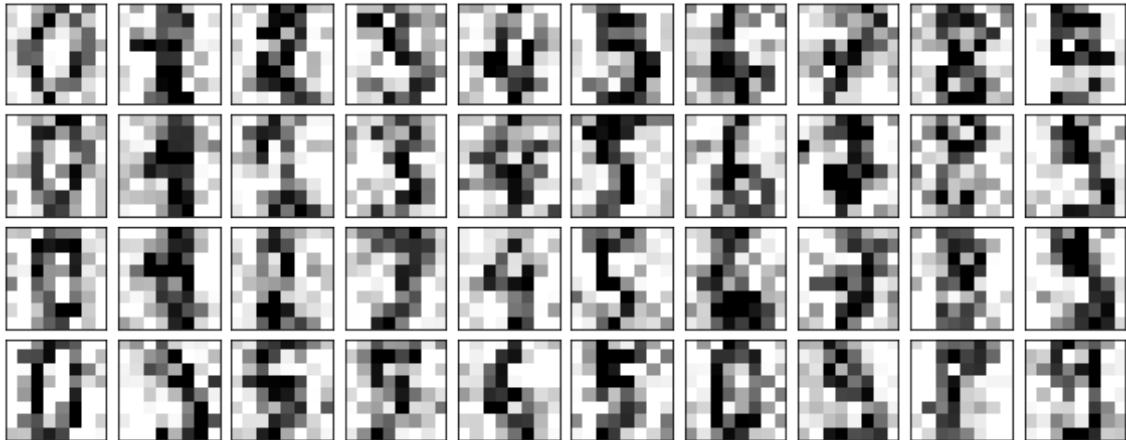


Figure 7: Set of MNIST digits with the addition of Gaussian Noise

💡 Tutor provided code

```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```

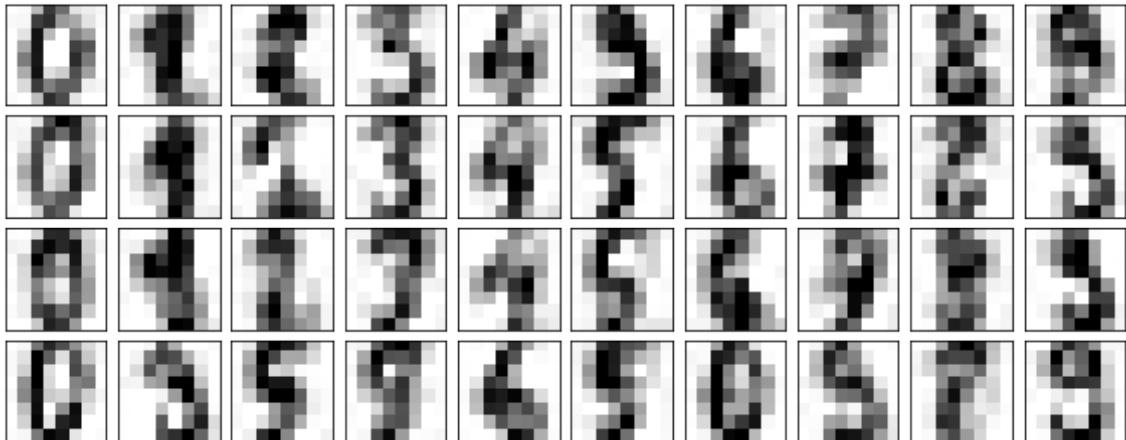


Figure 8: Noisy version of MNIST digits after filtering using PCA

This signal preserving/noise filtering property makes PCA a very useful feature selection routine — for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional representation. This will achieve two things:

- Filtering out random noise
- Reducing the computational effort required for model fitting

8.7 PCA on Facial Image Data

In this part of the workshop we will apply the same un-supervised learning algorythm (Principle Component Analysis) to images of famous people's faces. The images are available through sklearn and are sourced from the 'Labeled Faces in the Wild' database:

<http://vis-www.cs.umass.edu/lfw/#explore>

In this case, we select a set of people for which there are more than 60 images per person:

 Tutor provided code

```
faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use RandomizedPCA—it contains a randomized method to approximate the first N principal components much more quickly than the standard PCA estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

 Tutor provided code

```
pca = PCA(150, svd_solver='randomized')
pca.fit(faces.data)
```

```
PCA(n_components=150, svd_solver='randomized')
```

In this case, it can be interesting to visualize the images associated with the first several principal components. These components are technically known as “eigenvectors,” so these types of images are often called “eigenfaces”).

 Tutor provided code

```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```



Figure 9: Eigen Faces for the the Labeled Faces in the Wild database

Review the cumulative variance of these components to see how much of the data information the projection is preserving:

Your code here

We see that these 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components:

Compute the components and projected faces

Tutor provided code

```
pca = PCA(90).fit(faces.data)
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)
```

Plot the results

Your code here

The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features.

So although the PCA algorythm reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize

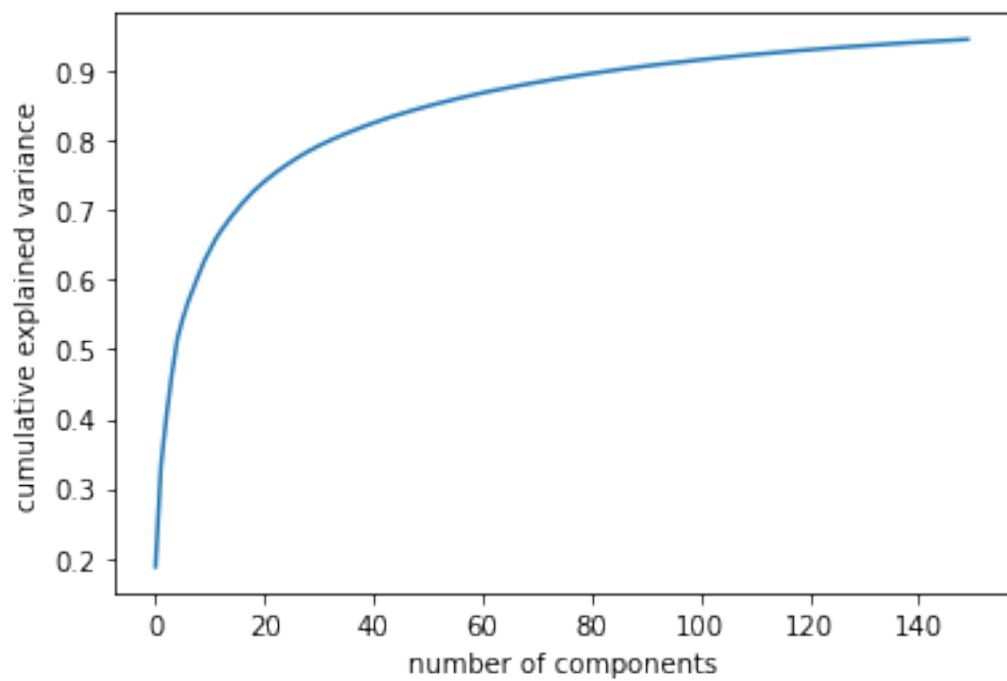


Figure 10: Cumulative variance plot for Eigen Faces



Figure 11: Comparing faces before and after dimensionality reduction

the individuals in the image.

In practice, PCA may often be used as a ‘pre-processor’ for other algorythms. It significantly reduces the amount of data in the data-set - retaining only the most important information and reducing noise. This means that any subsequent algorythm run on the data is likely to be much more efficient.

8.8 References:

Sections of code in this tutorial has been drawn from:

- https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html
- <https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html>

Chapter 9 - Reinforcement Learning

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2023-07-10

Table of contents

9 Reinforcement Learning	9 - 2
9.1 Introduction	9 - 2
9.2 Instructions for Students	9 - 3
9.3 Creation of the Learning Environment (World)	9 - 4
9.4 A simple attempt at controlling the cart-pole (Does not work!)	9 - 8
9.5 Tabular Q-Learning	9 - 10
9.5.1 Defining the Q-table	9 - 10
9.5.2 Defining controlling variables	9 - 13
9.5.3 Translating System State from Continuous to Discrete values	9 - 13
9.5.4 Selecting an Action based on the Q-Table	9 - 16
9.5.5 Learning! .. Updating the Q-Table based on the reward	9 - 17
9.5.6 Adaptive rates for Learning and Exploration	9 - 19
9.5.7 The main learning loop	9 - 20

List of Figures

1 Automating the training of <i>Canis lupus familiaris</i>	9 - 2
2 Line chart of Reward over 500 iterations for a naive control strategy	9 - 9
3 Bellman Equation	9 - 19
4 Line chart demonstrating epsilon decay	9 - 20
5 Line Chart showing Growth in Learning though multiple episodes	9 - 22

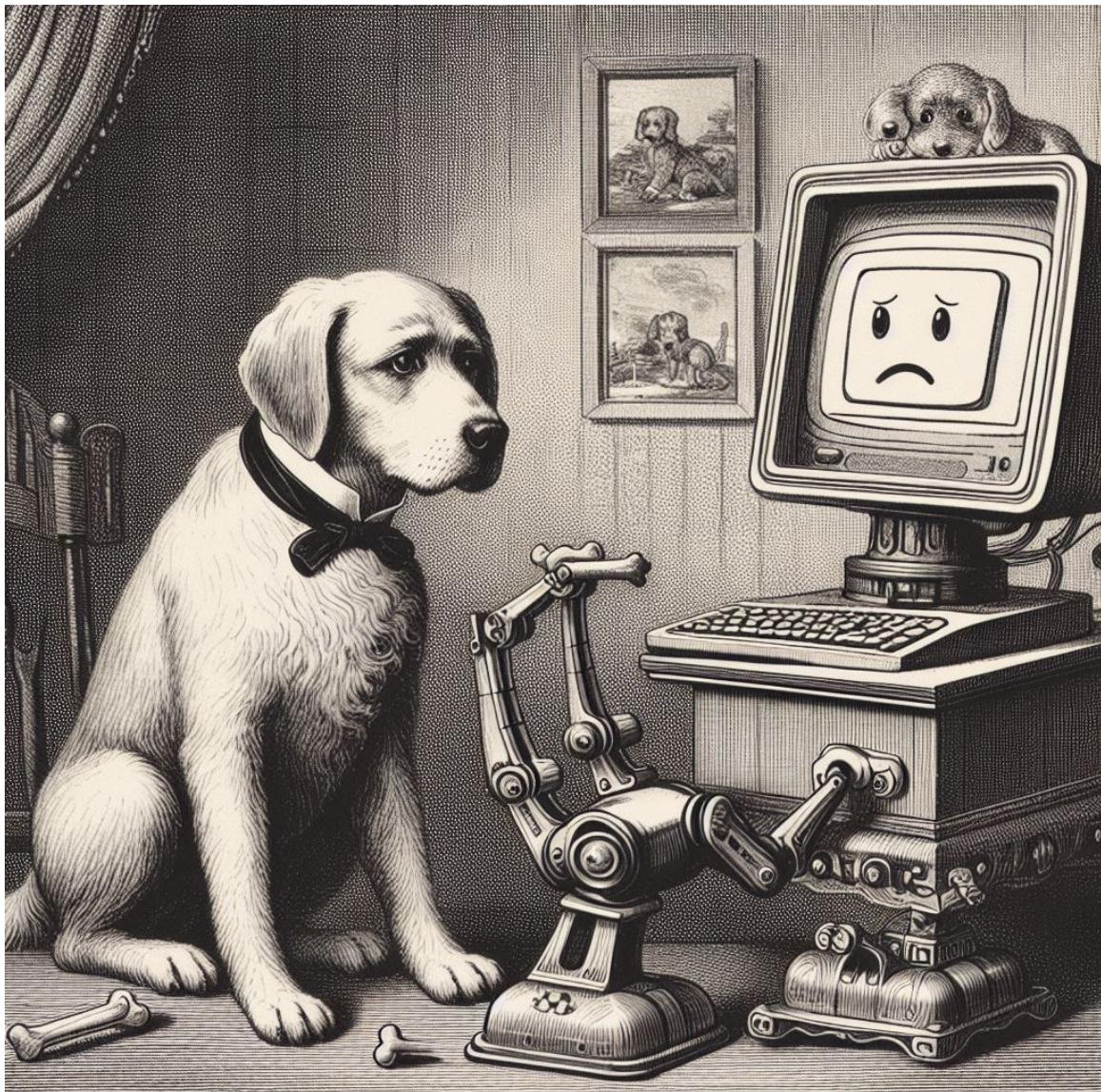


Figure 1: Automating the training of *Canis lupus familiaris*

9 Reinforcement Learning

9.1 Introduction

This workshop focusses on Reinforcement Learning. The idea here is to consider an ‘agent’ that is exploring a ‘world’ or environment, to achieve some goal. The goal is often phrased in terms of a reward. the Agent is normally able to make some observation of the World, based on their current location (state) in that world.

We will be using the library Open AI Gym library to support this workshop. AI Gym is documented here: <https://gym.openai.com/>

In this workshop we will focus on the ‘cart-pole’ problem. That is, the problem of balancing a pole by moving its base left and right. The observations of the world include the position of the trolley that the pole rests on, the angle of the pole and its speed.

This workshop presents two attempts to solve the problem. The first is rather simple .. and in fact does not really succeed! However, it does illustrate how the AI Gym software works.

The second approach uses ‘Q-Learning’. Q-Learning depends on playing the game many, many times. Each time the game is played, the various ‘states’ of the game are recorded. For each state, those actions that lead to a successful outcome are given a positive score. This positive score increases each time that the Agent has success performing a specific action in a specific state.

Agent learns the best policy iteratively. That is, it learns which is the most successful action to take in each state through a process of experimentation. The Agent is set up to select high-scoring actions more frequently than low-scoring actions. However, the Agent also has to do some ‘exploration’ - that is, sometimes it will take some random actions. This logic creates a balance between ‘exploration’ and ‘exploitation’.

9.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

💡 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class

8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

9.3 Creation of the Learning Environment (World)

First import the required libraries. We have used numpy and pandas multiple times during this course. The only new item here is ‘gym’ and this is the library for Open AI Gym.

- numpy : Conventionally named ‘np’
- pandas : Conventionally named ‘pd’
- math : Which is used for number conversion
- matplotlib.pyplot : Conventionally named ‘plt’
- gym

 Your code here

Open AI Gym provides a variety of different environments for exploring.

Add a code block to instantiate an object called ‘env’ using the ‘gym.make()’ method of gym. The name of the specific environment we will be using is ‘CartPole-v1’.

<https://gym.openai.com/envs/CartPole-v1/>

 Your code here

There is a requirement to re-set the environment before it is first used. It can also be reset at any time, if you want to write a programme that ‘experiments’ with different policies.

The reset() method also returns an array of values representing the starting state of the world.

 Tutor supplied code

```
obs = env.reset()  
print (f"obs = {obs}")
```

```
obs = [ 0.02070299  0.01927643  0.03545106 -0.00042302]
```

Observations are returned as an array of numbers:

- Position

- angle
- Angle
- Rotation angle

Add a code block that defines a function called ‘showObs()’ to print the return value in a neatly formatted manner.

 Your code here

Then call that function with a parameter of the value that was returned from the .reset() method.

 Your code here

```
Obs = [ 0.02070299  0.01927643  0.03545106 -0.00042302]
Position      = 0.02
    0 = centre, >0 = means to the right
angle         = 0.02
Angle         = 0.04
Rotation angle = -0.00
    Positive means clockwise
```

The ‘render()’ function for each environment is intended to create a window and draw a picture of the Environment and Actor.

I have commented out this command in my programme as I found that rendering the image crashed my programme.

You might experiment using the render() method **but please save your programme before you do so** to prevent any loss!

 Your code here

Open AI Gym provides a variety of environments for experimentation. They each offer a common set of functions for learning about the specific environment.

The number of options in the action space is provided by the attribute ‘action_space.n’.

 Your code here

```
env.action_space.n = 2
```

In this case, there are only two possible actions. These are each represented by numbers:

- 0 means ‘push the trolley to the left’ and
- 1 means ‘push the trolley to the right’

Create two variables ‘push_left’ and ‘push_right’ to represent these two cases.

💡 Your code here

```
push_left = 0  
push_right = 1
```

We can also find out about the observations the Agent can make. In this case, there are four things that the Agent can observe, represented by four numbers. The four numbers represent the following information:

1. Position of the trolley .. where ‘0’ is the centre of the environment and >0 means to the right
2. Speed
3. Angle of the pole from the vertical
4. Rotation speed .. where a positive number means a clockwise rotation

The ‘env’ object has attributes representing the maximum and minimum values for each of these observations:

- env.observation_space.low
- env.observation_space.high

These attributes are returned as a list of floating-point numbers

💡 Your code here

```
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]  
env.observation_space.low =  
    -4.8e+00,  
    -3.4e+38,  
    -4.2e-01,  
    -3.4e+38,  
env.observation_space.high =  
    4.8e+00,  
    3.4e+38,  
    4.2e-01,  
    3.4e+38,
```

Now, when I first worked on this code I found that the results were rather poor. After some investigation I discovered that whilst the *theoretical* range of the observations were as above .. in practice, the observed results were very much smaller. I have left the code I used to record the actual maximum and minimum values for observations in the code below. At this point, you may just note that rather than using the built-in ‘env.observation_space’, I will define my own set of values as follows:

 Tutor provided code

```
measured_obs_space_high = [ 2.5, 3.2, 0.4, 3.0 ]  
measured_obs_space_low = [-2.5,-3.2,-0.3,-3.0 ]
```

We can perform an Action in the environment using the ‘step’ function:

- Create a variable called ‘action’ and set it to the value ‘push_right’
- Pass this variable to the env.step() method
- The env.step() method will return 4 values, call these ‘obs’, ‘reward’, ‘done’ and ‘info’
- Use the ‘showObs()’ function defined above to display the new value of the observation

The method returns 4 separate pieces of information:

- Observation - for which we have defined a display function above
- Reward - A number representing a ‘score’ or reward for that round
- Done - A boolean variable that indicates ‘true’ when the simulation reaches an end-state
- Info - Specific data associated with the simulation (not used in this case)

We will deal with each of those returned values in turn:

 Your code here

```
Obs = [ 0.02108852  0.2138725   0.0354426  -0.28171329]  
Position      = 0.02  
    0 = centre, >0 = means to the right  
angle         = 0.21  
Angle         = 0.04  
Rotation angle = -0.28  
    Positive means clockwise
```

Add a code block to print the value of ‘reward’.

 Your code here

```
reward = 1.0
```

The reward in this case is just 1. For Cart-pole, the Agent gets a reward of 1 for each move they make without the pole falling over. The objective is to keep the pole upright for as long as possible.

Add a code block to display the value of ‘done’

The environment will signal if the particular ‘episode’ is finished. For example, if the pole has fallen over. In this case it has not.

💡 Your code here

```
done = False
```

Add a code block to display the value of ‘info’

Some environments provide some extra information. This might include how many ‘lives’ a player has. In the case of cart-pole there is no extra information available.

💡 Your code here

```
info = {}
```

9.4 A simple attempt at controlling the cart-pole (Does not work!)

First, we will use a very simple policy to try to solve the cart-pole problem. This Policy seems like an obvious idea ...

1. If the angle of the pole is to the left, then push the cart to the left
2. If the angle of the pole is to the right, then push the cart to the right

This seems to be a good idea! But sadly, you will see that it is too simple and does not work at all well.

First, here is the policy, written as a function. If the angle part of the Observation ('obs[2]') is less than zero it returns 0, otherwise it returns 1:

Define a function that takes an observation as a parameter. Determine the angle of the pole. If the angle is less than 0 then return 0 Otherwise return 1 (you may use the previously defined ‘push_left’ and ‘push_right’ variables for this). These values will be used as the actions.

 Your code here

In the following block we create a loop that plays the game multiple times. Each time the game is played we add up the total reward in ‘episode_rewards’. At the end of a specific game, we take this total reward for the game and add it to a list called ‘totals’. Thus, when the game is finished we will be able to plot a graph of all of the games to see how well this policy worked.

 Tutor supplied code

```
for episode in range(500):
    obs=env.reset()
    episode_rewards = 0
    for step in range(200):
        action = simple_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        # Remove the comment character (#) below
        # to draw the environment in each iteration
        # env.render()
        if done:
            break
    totals.append( episode_rewards )
```

Add a block to print a line chart showing the reward value recorded in the above loop. This can be easily achieved by creating a Pandas dataframe from the ‘totals’ list above, then calling the ‘plot.line()’ method on that dataframe.

 Your code here

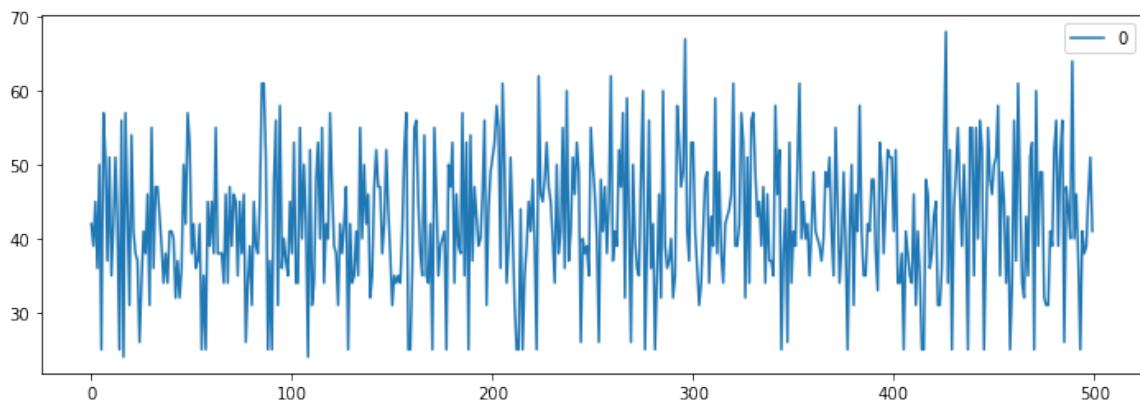


Figure 2: Line chart of Reward over 500 iterations for a naive control strategy

There is clearly no learning occurring. The graph above looks random! It certainly does not get better as each episode is played. Also, the scores are not very good.

Summarise the data using the pandas ‘describe()’ method.

Your code here

	0
count	500.000000
mean	42.186000
std	9.094439
min	24.000000
25%	36.000000
50%	41.000000
75%	49.000000
max	68.000000

9.5 Tabular Q-Learning

9.5.1 Defining the Q-table

We need to define the ‘Q-table’. This table has two parts:

1. A part representing the state of the Actor, and
2. A part containing a score for each of the possible actions that can be taken in that state.

As described in the lecture, the ‘Cart-pole’ environment represents its state as continuous numbers. We can’t build a Q-table based on continuous variables as it would be vast. So these numbers need to be broken down into discrete ‘buckets’.

Define a tuple called ‘buckets’. This defines the number of discrete ‘groups’ (buckets) that each of the 4 state variables are divided into. In the first instance, set the value of buckets to (1,1,6,12,)

Note that the number of buckets for each state variable was developed experimentally. That is, there was no deep theory or mathematics that lead to the following number of buckets. Rather, lot’s of different numbers were tried until one of them worked!

After you have completed this workshop, come back to this point and modify the number of buckets to see what impact it has on the performance of the learning algorithm.

 Tutor provided code

```
position_buckets = 1
speed_buckets = 1
angle_buckets = 6
rotation_buckets = 12
buckets=(position_buckets,
          speed_buckets,
          angle_buckets,
          rotation_buckets,)
print(f"buckets = {buckets}")
```

```
buckets = (1, 1, 6, 12)
```

Add a cell to create the ‘Q-table’.

We use numpy, to create a multi-dimensional array to store each of the state - action pairs:

 Tutor Supplied Code

```
Q = np.zeros(buckets + (env.action_space.n,))
```

The code above is quite complicated to read unless you are very familiar with Python. So let me explain ...

‘np.zeros’ can be used to create arrays of many dimensions and fill them with zeros. Thus, a simple example would be:

 Tutor Supplied Code

```
np.zeros(5)
```

```
np.zeros(5) = [0. 0. 0. 0. 0.]
```

Which creates a 1-dimensional array with 5 elements.

Or we could produce a more complex array like this:

 Tutor Supplied Code

```
array_shape = (2,3)
np.zeros(array_shape)
```

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

Which uses the tuple '(2,3)' to define an 2 x 3 array.

We can go further:

 Tutor Supplied Code

```
array_shape = (2,3,4)
np.zeros(array_shape)
```

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

      [[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

The tuple '(2,3,4)' defines an array of 2x3x4 ... and so on.

Now, we need an array that has space for each of the state variables (as defined in buckets) and also space for the possible actions. The following tuple gives us an array of the right shape:

 Your code here

```
buckets + (env.action_space.n,) = (1, 1, 6, 12, 2)
```

So, to be clear .. the Q-table is a ‘model’ of the state-space of the system. It provides to ‘cells’ for each position in the state space. These cells are going to be used to store the results from experiments .. that is the reward from making different actions. In each state of the system we can either push right or push left.

The Q-table tells us:

- When the system was in this state last time and the ‘push-left’ action was selected, this was the reward, and
- When the system was in this state last time and the ‘push-left’ action was selected, this was the reward

With the simplification being that, because the real state-space of the system is so large we have to ‘quantize’ (chunk) that space into ‘buckets’ simple to make the size of the Q-table tractable.

So at this point we can create a numpy array called ‘Q’ full of zeros in the required shape (code replicated from above):

 Your code here

```
Q.shape =  
(1, 1, 6, 12, 2)
```

9.5.2 Defining controlling variables

The following block sets the value of the various constants that define the behaviour of the programme. You learnt about some of these during the lecture.

 Tutor Supplied Code

```
# Number of training episodes  
n_episodes = 5  
  
# learning rate  
min_alpha = 0.1  
  
# exploration rate  
min_epsilon = 0.1  
  
# discount factor  
gamma = 1.0  
  
# Adaptation rate for alpha and gamma  
ada_divisor = 25
```

9.5.3 Translating System State from Continuous to Discrete values

We now need to define a function that takes an observation represented as a continuous number, and translates it into a discrete version. The discrete version will be used as an index into the Q-table.

Numpy provides a pair of useful functions that can help with this task:

- np.linspace(min_value, max_value, num_boundaries) :
 - Divides a range into equally spaced regions
 - Returns the values of each boundary
- np.digitize(input_number, bin_edges)
 - Given an array of ‘bin_edges’ returns the index of the bin a particular value would fall into

- Conveniently, the ‘bin_edges’ array is exactly what is returned by ‘np.linspace’
- Note that the function returns a one-based result. So if you are going to use the result as an index into a numpy array (which is zero-based) you will need to subtract 1 from the result

Add blocks of code to experiment with these functions so that you are comfortable with how they operate.

First producing the ‘boundaries’ using np.linspace:

 Your code here

```
my_boundaries = np.linspace(0, 10, 5) = [ 0.    2.5  5.    7.5 10. ]
```

Then translating arbitrary values into their respective buckets (‘quantization’ or ‘discretize’)

 Your code here

```
np.digitize(3, my_boundaries) = 2
```

We can use these functions in combination with variables we have already used above:

The number of buckets for each dimension in the state space

```
position_buckets = 1
speed_buckets = 1
angle_buckets = 6
rotation_buckets = 12
```

and the maximum and minimum values for the state-space described above:

- measured_obs_space_low
- measured_obs_space_low

Although not strictly necessary, the code is easier to read if we extract the values from these arrays into named variables:

 Your code here

```

position_low = env.observation_space.low[0] = -2.50e+00
position_high = env.observation_space.high[0] = 2.50e+00
speed_low = env.observation_space.low[1] = -3.20e+00
speed_high = env.observation_space.high[1] = 3.20e+00
angle_low = env.observation_space.low[2] = -3.00e-01
angle_high = env.observation_space.high[2] = 4.00e-01
rotation_low = env.observation_space.low[3] = -3.00e+00
rotation_high = env.observation_space.high[3] = 3.00e+00

```

It is then a simple matter to create arrays of boundary values that define buckets for each state variable:

 Your code here

```

position_buck_bounds =
[
-2.50,
]

speed_buck_bounds =
[
-3.20,
]

angle_buck_bounds
[
-0.30,
-0.16,
-0.02,
0.12,
0.26,
0.40,
]

rotation_buck_bounds =
[
-3.00,
-2.45,
-1.91,
-1.36,
-0.82,
-0.27,
0.27,
0.82,
]

```

```
1.36,  
1.91,  
2.45,  
3.00,  
]
```

All of this then allows us to create functions that will discretize (quantize) any of the state variables.

Add a function called ‘discretize’ that takes an observation ('obs') as a parameter. The signature should look like this:

```
def discretize(obs):
```

Use the ‘np.digitize’ function and the previously computer boundary position lists to compute a list of quantized values for an observation. Return the result as a list.

💡 Your code here

We can ‘smoke-test’ this function by running some episodes for the system and checking that the results are quantized as expected.

💡 Your code here

```
Obs = (0, 0, 3, 6)  
Position      = 0.00  
    0 = centre, >0 = means to the right  
angle         = 0.00  
Angle         = 3.00  
Rotation angle = 6.00  
    Positive means clockwise
```

It is these values that will be used to find a location in the Q-table.

9.5.4 Selecting an Action based on the Q-Table

The next function selects an action in a particular state. Remember that the programme does not always select the ‘best’ option in the Q-Table! Remember in the lecture that Mario sometimes has to open a random door, even if he knows that one of the doors has a reward.

This is called the ‘explore / exploit’ trade-off. As well as exploiting what we already know, we also sometimes need to take some risks and explore new options. This is the route of learning!

The amount of times we ‘explore’ rather than ‘exploit’ is defined by the ‘epsilon’ variable. So the following function will sometimes choose the best option from the Q-table, and sometimes select a random action, depending on the value of Epsilon.

The ‘best’ option is found using the ‘np.argmax’ function. ‘np.argmax’ find the position of the largest value.

Experiment with the following function until you understand what it does.

 Tutor provided code

```
np.argmax([1,12,1,60,2,4])
```

```
np.argmax([1,12,1,60,2,4]) = 3
```

In this case the largest value is at position ‘3’ ... remember that in Python, arrays are numbered from 0!

Given that, we can define a function that returns one of two values randomly.

- A random action from the possible action space, or
- The current ‘best’ option as defined by the Q table.

As mentioned above, the frequency of these two options is defined by the ‘epsilon’ parameter that represents explore / exploit.

 Tutor provided code

```
def choose_action(state, epsilon):  
    return (env.action_space.sample())  
        if (np.random.random() <= epsilon)  
        else np.argmax(Q[state])
```

9.5.5 Learning! .. Updating the Q-Table based on the reward

This next function is the heart of our learning algorithm. After we take an action we review the reward. This function needs to know the state that we were in **before** taking the action, the new state and the reward that was received.

In this function, ‘alpha’ is the learning rate. The value of alpha controls the influence that new information has on the Q-table.

‘gamma’ is the discount factor. The discount factor controls the impact of immediate reward compared with delayed reward.

The function used here was developed by Bellman, see:

https://en.wikipedia.org/wiki/Bellman_equation

It is often described in the form of an equation, as follows:

$$NewQ(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max Q'(s', a') - Q(s, a)]$$

The diagram shows the Bellman Equation with handwritten annotations below each term:
 - $Q(s, a)$: New Q value for that state and that action
 - $R(s, a)$: Current Q value
 - $\gamma \max Q'(s', a')$: Reward for taking that action at that state
 - α : Learning Rate
 - $1 - \alpha$: Discount rate
 - $Q(s, a) + \alpha [R(s, a) + \gamma \max Q'(s', a') - Q(s, a)]$: Maximum expected future reward given the new s' and all possible actions at that new state

Figure 3: Bellman Equation

Tutor provided code

```
def update_q(state_old, action, reward, state_new, alpha):
    Q[state_old][action] += (alpha
        * (reward
        + gamma * np.max(Q[state_new])
        - Q[state_old][action]))
    )
```

9.5.6 Adaptive rates for Learning and Exploration

As explained in the lecture, a common strategy in reinforcement Learning is to have the values of alpha and epsilon decay over a period of time. That means that the Actor adapts its learning behaviour the longer the game is played.

The following two functions compute decaying values for each of ‘epsilon’ and ‘alpha’

Tutor provided code

```
def get_epsilon(t):
    return max(min_epsilon,
               min(1, 1.0
                   - math.log10((t + 1) / ada_divisor)))

# Adaptive learning of Learning Rate
def get_alpha(t):
    return max(min_alpha,
               min(1.0, 1.0
                   - math.log10((t + 1) / ada_divisor)))
```

Add a block to plot a line chart of epsilon so that you can observe the general shape of the decay

Your code here

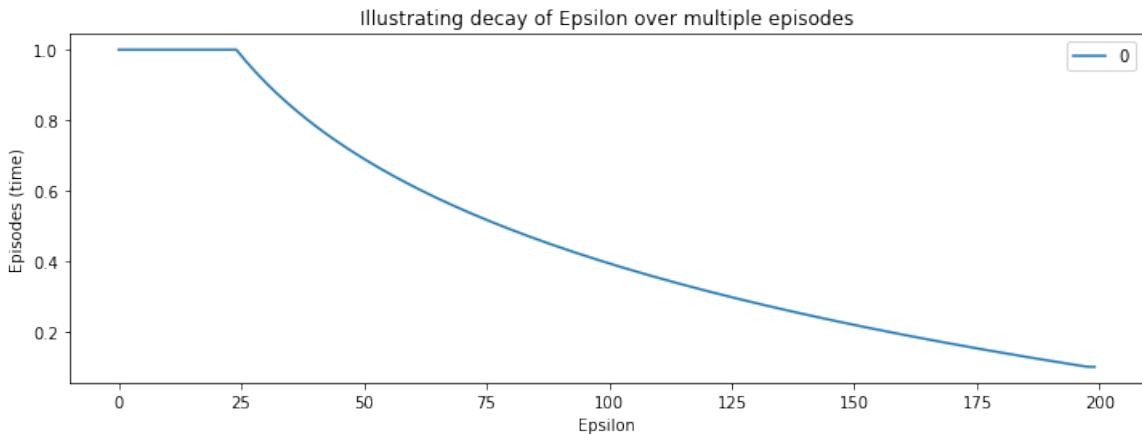


Figure 4: Line chart demonstrating epsilon decay

As explained in the first part of this tutorial, whilst the cartpole simulation does provide values for maximum and minimum observed values, in practice, the actual observed values run over a much smaller range. This leads to issues in the quantization of observations.

Consequently, at this point in the project I included some extra code to calibrate the observed state values coming from the simulation. I have not included this code in the ‘no-code’ version of this workbook as it is of little educational value and its inclusion detracts from the main narrative flow. However, in summary, the results from recording the actual observed maximum and minimum observed state values are encoded in the variables included above:

- measured_obs_space_high = [2.5, 3.2, 0.4, 3.0]
- measured_obs_space_low = [-2.5,-3.2,-0.3,-3.0]

The code to obtain these values is included in the ‘full-code’ version of this notebook.

9.5.7 The main learning loop

Having defined all of the above functions it is now time to define the main learning loop. The game is played many times - as defined by the variable ‘n_episodes’. Each time a game is played the programme does the following:

1. The state is observed
2. The learning and discount rates are calculated
3. The programme tries to balance the pole

Balancing the pole means:

- Selecting an action
- Observing the new new state of the system
- Obtaining the reward

- Based on that response .. updating the Q-table
- Continue doing this until either the pole falls over, or the game ends (200 steps)

 Tutor Supplied Code

```
n_episodes = 200

# These lists used to store the episode scores
y_chart = []
x_chart = []

# plt.figure(100)

# Set up the chart to display results
plt.xlim((0,n_episodes))
plt.ylim((0,210))
plt.title('Episode Score for Each Episode')
plt.ylabel('Score')
plt.xlabel('Episode')

for e in range(n_episodes):
    # As states are continuous, discretize them into buckets
    current_state = discretize(env.reset())

    # Get adaptive learning alpha and epsilon decayed over time
    alpha = get_alpha(e)
    epsilon = get_epsilon(e)

    done = False
    total_reward = 0

    while not done:
        # Render environment
        # Comment out to make the code run faster
        # env.render()

        # Choose action and take it
        action = choose_action(current_state, epsilon)
        obs, reward, done, _ = env.step(action)

        # Used to calibrate state-space observation
        store_obs_limits(obs)
        new_state = discretize(obs)

        # Update Q-Table
```

```

update_q(current_state, action, reward, new_state, alpha)
current_state = new_state
total_reward += reward
y_chart.append(total_reward)
x_chart.append(e)
env.close()

plt.plot(x_chart,y_chart, color='blue')
plt.show()

```

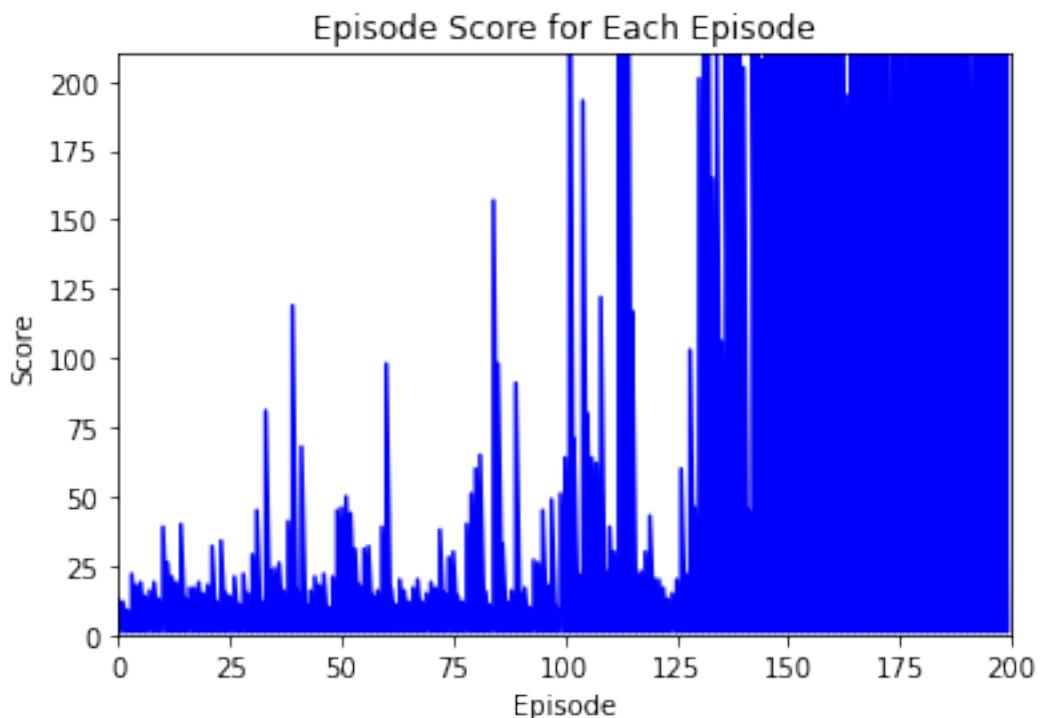


Figure 5: Line Chart showing Growth in Learning through multiple episodes

```

pos_recorded_low = -2.43
pos_recorded_high = 0.41
speed_recorded_low = -1.96
speed_recorded_high = 1.80
angle_recorded_low = -0.26
angle_recorded_high = 0.16
rotation_recorded_low = -2.86
rotation_recorded_high = 2.99

```

Finally we can see effective learning taking place. With these parameters the code takes around 200 iterations to perfect its skill in balancing the cart-pole.

A useful further exercise would be to modify the code above and change the number of buckets in the Q table. Does this make the overall learning faster or slower?

Chapter 10 - Classical Natural Language Processing

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2023-04-10

Table of contents

10 Classical Natural Language Processing	10 - 3
10.1 Introduction	10 - 3
10.2 Instructions for Students	10 - 3
10.3 Load the required libraries	10 - 4
10.4 Load Data	10 - 4
10.5 Bag-of-words : Organising the text for ease of analysis	10 - 5
10.6 Exploratory Data Analysis	10 - 6
10.6.1 Pareto of word frequency	10 - 6
10.6.2 Word-cloud	10 - 8
10.7 Part-of-Speech (POS) Tagging	10 - 10
10.7.1 Selecting words based on their ‘Part of Speech’ (PoS).	10 - 12
10.7.2 Charting words with a specific PoS Tag	10 - 14
10.8 Sentiment analysis	10 - 14
10.9 Identifying themes in text	10 - 19

List of Figures

1 Several attempts were made to build machines that could analyse text	10 - 2
2 Bar-chart of word Frequency	10 - 9
3 Word-Cloud of text from the Bank Complaint Corpus	10 - 10
4 Bar-chart of word Frequency with Specific PoS Tag	10 - 15
5 Histogram of compound sentiment score for bank reviews	10 - 18



Figure 1: Several attempts were made to build machines that could analyse text

10 Classical Natural Language Processing

10.1 Introduction

This workshop focusses on processing of text. That is, rather than tabulated data we have used so far we are going to start with text written in English.

The data-set we are using was extracted from a popular review website. The reviews are for a famous bank in the UK. In this case, the name of the bank has been replaced with the more generic text ‘bank_name’.

The motivation this workshop is the case when the information we are trying to model arrives in the form of text. Of course, there are many popular and valuable examples of where we may want to analyse text automatically:

- Books and stories - looking for patterns and deeper understanding
- Web sites - blogs, social media, review websites etc.
- E-mails
- Speech that have been transcribed into text - for example lectures and/or political speeches
- Text based ‘chat’ exchanged over the Internet.

This workshop is focussed on the ‘analysis’ of text rather than text production or translation.

10.2 Instructions for Students

In this workbook there are regular ‘callout’ blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided ‘clues’ towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do

6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

10.3 Load the required libraries

For this practical session we will need the following libraries:

- pandas : conventionally given the reference name ‘pd’
- numpy : Conventionally given the reference name ‘np’
- CountVectorizer : from sklearn.feature_extraction.text
- matplotlib.pyplot : Conventionally given the reference name ‘plt’
- wordcloud and STOPWORDS from WordCloud
- nltk
- stopwords : from nltk.corpus
- tagged : from nltk.corpus.reader
- OrderedDict : from collections
- SentimentIntensityAnalyzer : from vaderSentiment.vaderSentiment
- NMF : from from sklearn.decomposition

💡 Your code here

10.4 Load Data

Add a block to load the data into a pandas dataframe. The file is called “Bank Reviews with cust ID v3.csv”. This file contains some characters that may not be accepted by the default character encoding for pandas read_csv. In which case, set the encoding to be ‘latin1’ - which worked for me.

As in previous chapters, I have split my display of the dataframe over two cells to enable neat printing into this workbook format. You do not need to do this as you will be displaying on a screen and a scroll-bar will be available.

💡 Your code here

	customerID	summary
0	Customer_1	B*rstards who take pleasure in harrassing disab...
1	Customer_10	BANK_NAME - dont bother as they have little r...
2	Customer_100	I visited the Bognor Regis BANK_NAME
3	Customer_101	Abysmal business service
4	Customer_102	Disgusting company
...
1800	Customer_1805	Utterly time wasting
1801	Customer_1806	Amazing customer service from andreia
1802	Customer_1807	Is this Bank or...
1803	Customer_1808	worst bank in the UK
1804	Customer_1809	Simple matter of changing an address!!

	complaintText
0	As a disabled person in an empty branch, they ...
1	Bad customer service in branch and not enough ...
2	I visited the Bognor Regis BANK_NAME Branch t...
3	Abysmal business service. If you are a busine...
4	I have company account with BANK_NAME and got...
...	...
1800	Worst ever service I have ever received.a bun...
1801	You are lucky to have a lady like Andrea in y...
1802	You are more likely to get offended than sort...
1803	You know you have to change banks when a tran...
1804	You would think that a changing your address”...

As you see, we have 1805 rows of data. Each row is a review from a different customer. The review text is split into two parts:

- a ‘Summary’ sentence and the
- the main body of the review. The main review is typically several sentences of text.

10.5 Bag-of-words : Organising the text for ease of analysis

A key step in analysing the text is to find a representation that enables familiar ‘statistical’ Machine Learning tools to model the data. One method for doing this is called a ‘Bag of words’. Essentially words in each review are counted. A large array is created in which each column represents each of the words used in the entire collection of reviews. Each row of the array represents one review. The data in the row is a count of the number of times each word appears in that review.

Sklearn feature extraction provides a tool to do this: Count Vectorizer.

The data is returned as two data structures. The first, is a simple list of all of the words in the vocabulary of the review ('get_feature_names_out').

 Tutor provided code

```
vectorizer = CountVectorizer(stop_words='english', min_df =0.005)
bagOfWords = vectorizer.fit_transform(df['complaintText'])
vectorizer.get_feature_names_out()
```

```
array(['00', '000', '10', ..., 'yesterday', 'young', 'zero'], dtype=object)
```

The other, is an array-like structure that counts the word occurrences for each review.

Add a cell to convert 'bagOfWords' into a pandas dataframe, with the feature (column) names set to 'vectorizer.get_feature_names()'.

In the following I have displayed only columns 50 to 59 to enable neat display in this workbook. But you can display and review the complete dataframe.

 Your code here

	active	activity	actual	actually	add	added	additional	address	admit
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
...
1800	0	0	0	0	0	0	0	0	0
1801	0	0	0	0	0	0	0	0	0
1802	0	0	0	0	0	0	0	0	0
1803	0	0	0	0	0	0	0	0	0
1804	0	0	0	0	0	0	0	1	0

10.6 Exploratory Data Analysis

10.6.1 Pareto of word frequency

It is instructive to look at the frequency of occurrence of all words in the data-set. This can help identify any gross anomalies in the data and provide a view of the vocabulary for later analysis.

In the following cells you will create a sorted the count of words ordered so that the most frequent items occur first.

First, add a cell to sum the columns in the above table.

💡 Your code here

```
sum_words = [[ 25  43 160 ...  20  25  36]]  
type(sum_words) = <class 'numpy.matrix'>
```

Next, create a list of tuples that contain each word and its frequency. Then print the first few items in that list.

💡 Your code here

```
[('disabled', 32),  
 ('person', 156),  
 ('branch', 732),  
 ('told', 607),  
 ('walk', 20),  
 ('area', 20),  
 ('till', 24),  
 ('complained', 35),  
 ('said', 420),  
 ('reported', 12)]
```

Then, sort that list by the frequency.

(Note in the code below I use a ‘lambda’ function. We have not used this previously in the course. A lambda function applies a function and returns a result. Further information on lambda functions may be obtained here:

- https://www.w3schools.com/python/python_lambda.asp
- <https://towardsdatascience.com/lambda-functions-with-practical-examples-in-python-45934f3653a8>
- <https://www.geeksforgeeks.org/applying-lambda-functions-to-pandas-dataframe/>

The following code can be read as follows: + Sort the ‘words_freq’ list + When sorting, the sort key is obtained by looking at the second element of each tuple (x[1]) + Sort the items in reverse order

 Tutor provided code

```
words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
words_freq[0:20]
```

```
[('bank_name', 2152),
 ('account', 1893),
 ('bank', 1579),
 ('service', 907),
 ('customer', 841),
 ('money', 795),
 ('card', 774),
 ('branch', 732),
 ('time', 623),
 ('told', 607),
 ('banking', 578),
 ('phone', 547),
 ('just', 527),
 ('years', 507),
 ('business', 499),
 ('online', 471),
 ('said', 420),
 ('staff', 379),
 ('new', 379),
 ('ve', 360)]
```

Now create a cell to copy that list into a pandas dataframe and draw a bar-chart of the frequency counts.

 Your code here

Experiment with the range of words plotted in this chart. Do you get any insights when looking at the frequently occurring words?

10.6.2 Word-cloud

An alternate and popular method for reviewing word frequencies in text is called a ‘word-cloud’. Such graphics are principally intended to be ‘illustrative’ or ‘artistic’ rather than highly ‘analytic’. However, they do provide an attractive graphic that in some ways conveys a greater sense of the content of the reviews that the more ‘representational’ sorted bar-chart:

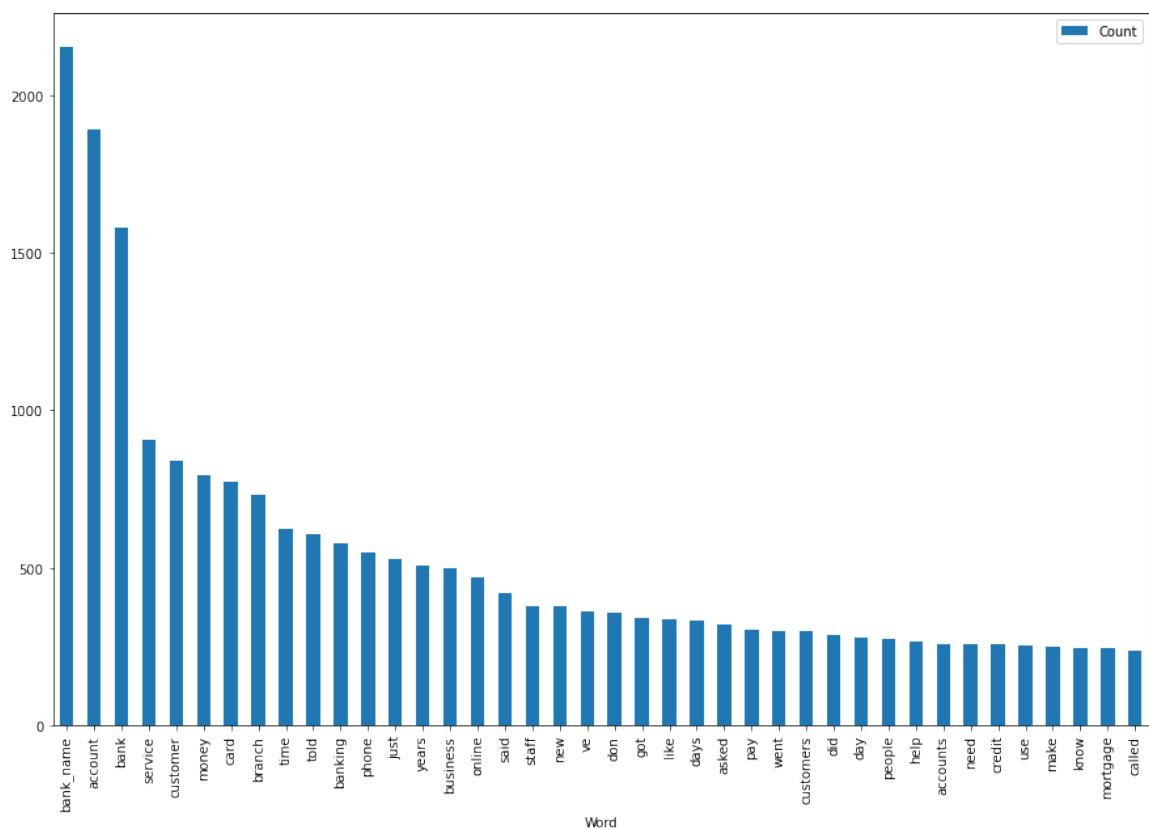


Figure 2: Bar-chart of word Frequency

 Tutor provided code

```
text = df['complaintText'].tolist()
text = ' '.join(text).lower()
wordcloud = WordCloud(stopwords = STOPWORDS,
                      collocations=True,
                      background_color='white',
                      width=2000,
                      height=1000).generate(text)

plt.figure( figsize=(15,15))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

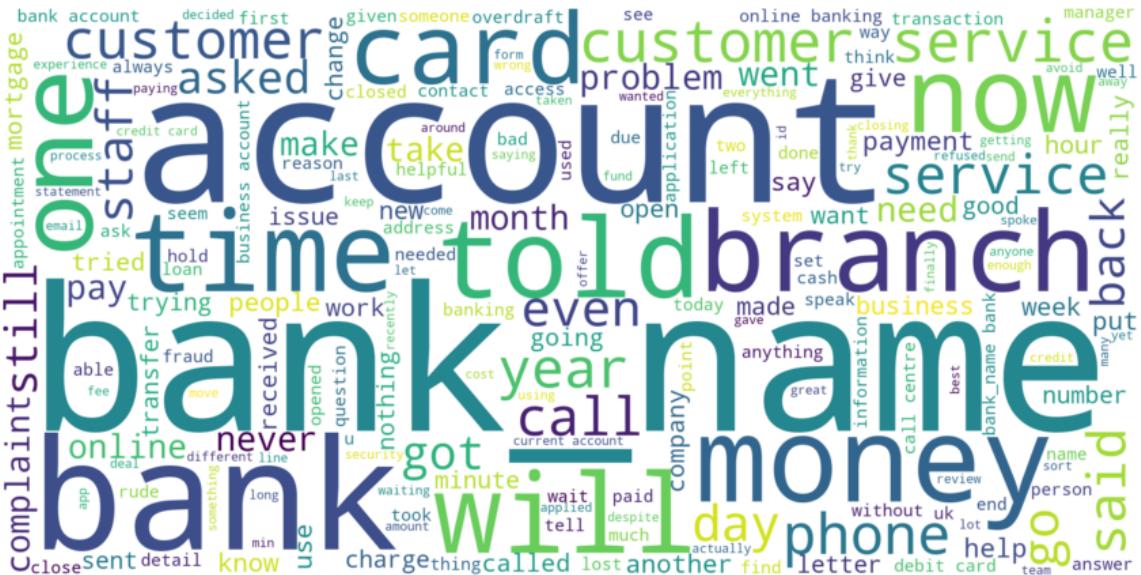


Figure 3: Word-Cloud of text from the Bank Complaint Corpus

10.7 Part-of-Speech (POS) Tagging

When dealing with Natural Language we will frequently need to think about grammar. Different words have different functions within the language and it will often be necessary to determine what use each word is being put to in a specific context. This is termed the ‘Part of Speech’ and refers to the ‘type’ of the word in its context - noun, verb, adjective etc.

Note, that in English single words can have different grammatical usages in different contexts (For example ‘rose’ - the flower (noun) and the past-tense of ‘rise’ (verb)). This means that a simple dictionary look-up table for each word will not be sufficient. A more complex algorithm is required that determines the PoS as words are used in a specific context.

As previously however, standard libraries come to the rescue. The ‘nltk’ (Natural Language Tool-kit’ library is a powerful library dedicated, as the name suggests, to the manipulation and analysis of human language.

You can learn more about nltk here: <https://www.nltk.org/>

In this section we will demonstrate nltk’s ability to perform grammar parsing. We will use the nltk parsing functions to perform some further, exploratory data-analysis. Specifically, to extract parts of speech (nouns, verbs, adjectives) and plot these as frequency plots.

The intention here is specifically to demonstrate grammar parsing however, this kind of analysis might be useful, for example, for a company who wants to understand how its customers are reacting to different products and services by reviewing specific nouns and adjectives.

nltk requires some data and libraries to be downloaded:

 Tutor provided code

```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
[nltk_data] Downloading package stopwords to C:\Users\Rob
[nltk_data]     Collins\AppData\Roaming\nltk_data...
[nltk_data] Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to C:\Users\Rob
[nltk_data]     Collins\AppData\Roaming\nltk_data...
[nltk_data] Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\Rob Collins\AppData\Roaming\nltk_data...
[nltk_data] Package averaged_perceptron_tagger is already up-to-
[nltk_data]     date!
```

True

‘stop words’ are terms that are used very frequently in a language and are often used to signal grammatical structure. Examples would be ‘the’ and ‘a’ in English. In some language analysis tasks these terms are not useful. Because they appear very frequently they can distort statistics. They are also likely to be common across large sections of text - so that they can distort clustering algorithms etc. Therefore, for some NLP tasks we can remove these from the set of words - and nltk provides facilities for doing that.

 Tutor provided code

```
stopwords_set = set(stopwords.words("english"))'off')
```

Add a cell to create an empty ‘wordcount’ dictionary which will be used to store a list of the words and their frequency count.

💡 Your code here

10.7.1 Selecting words based on their ‘Part of Speech’ (PoS).

In this section we are going to focus on ‘adjectives’.

nltk uses the Penn Treebank codes to represent various parts of speech. The complete list of tags is here:

https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Create a python set that contains the various tags that nltk uses to represent different type of adjectives, those being ‘jj’, ‘jjr’ and ‘jjs’.

When you have completed the section below, experiment with the value of ‘interestingTags’ below to produce some charts of word frequencies for other types of words.

💡 Your code here

```
(interestingTags = {'JJS', 'JJ', 'JJR'})
```

We will also need a facility to break sentences into individual words. nltk provides just such a function:

💡 Tutor provided code

```
tokenizer = nltk.tokenize.punkt.PunktSentenceTokenizer()
```

We are now in a position to:

- iterate through ‘complaintList’
- extract individual words
- determine their part-of-speech (PoS)
- filter the words that are adjectives
- Update the ‘wordCount’ dictionary to store only the count of adjectives

💡 Your code here

Which finally gives us a list of words, their corresponding PoS tag, and the frequency that the word occurs as that PoS in the review text.

Add a cell to print the the ‘first’ 20 items of the dictionary (strictly speaking python dictionaries are un-ordered but since Python 3.7 they do retain their insertion order)

💡 Tutor provided code

```
for item in list(wordCount.items())[:20]:  
    print (item)
```

```
(('disabled', 'JJ'), 25)  
((('empty', 'JJ'), 7)  
((('snaked', 'JJ'), 1)  
((('furthest', 'JJS'), 1)  
((('available', 'JJ'), 49)  
((('sadistic', 'JJ'), 1)  
((('enough', 'JJ'), 32)  
((('human', 'JJ'), 20)  
((('clear', 'JJ'), 43)  
((('biggest', 'JJS'), 7)  
((('£920', 'JJ'), 1)  
((('wifes', 'JJ'), 2)  
((('able', 'JJ'), 126)  
((('unaware', 'JJ'), 4)  
((('little', 'JJ'), 55)  
((('different', 'JJ'), 122)  
((('less', 'JJR'), 33)  
((('correct', 'JJ'), 27)  
((('awful', 'JJ'), 56)  
((('international', 'JJ'), 28)
```

As previously, it is informative to see these as a sorted list - putting the most frequent terms at the start of the list:

💡 Tutor provided code

```
sortedWordCount = OrderedDict(sorted(wordCount.items(),  
                                     key = lambda t: t[1],  
                                     reverse=True))  
for item in list(sortedWordCount)[0:20]:  
    print(f"{item} -> {wordCount[item]}")
```

```
('new', 'JJ') -> 363  
('good', 'JJ') -> 219  
('first', 'JJ') -> 171  
('bad', 'JJ') -> 159
```

```
('last', 'JJ') -> 158
('due', 'JJ') -> 143
('many', 'JJ') -> 140
('current', 'JJ') -> 130
('able', 'JJ') -> 126
('personal', 'JJ') -> 125
('local', 'JJ') -> 125
('online', 'JJ') -> 123
('different', 'JJ') -> 122
('next', 'JJ') -> 119
('helpful', 'JJ') -> 117
('wrong', 'JJ') -> 116
('worst', 'JJS') -> 113
('much', 'JJ') -> 109
('poor', 'JJ') -> 103
('terrible', 'JJ') -> 96
```

10.7.2 Charting words with a specific PoS Tag

As above, we can now plot these counts as a bar-chart

💡 Your code here

As you can see, the chart now focusses on descriptive term. The bank might take some comfort in the fact that the term ‘good’ is more frequent than the term ‘bad’ .. although I think I would worry about the occurrence rate of ‘wrong’, ‘worst’, ‘poor’ and ‘terrible’. (I would also not underestimate the tendency of British reviewers to be rather sarcastic!).

In the above code block, the range of words has been set to a small range using:

- listStart =0
- listEnd = 20

Try altering these values and view different sections of the sorted data-set

10.8 Sentiment analysis

Having looked at descriptive terms in the reviews, we will now consider the ‘mood’ of each review. That is, was the writer ‘happy’ or ‘angry’ - or expressed more technically, were they expressing positive sentiment or negative sentiment?

Sentiment analysis is useful when considering customer feedback and, for example, comments on social media. A useful function for a business might be to track the mood of its social media audience over time.

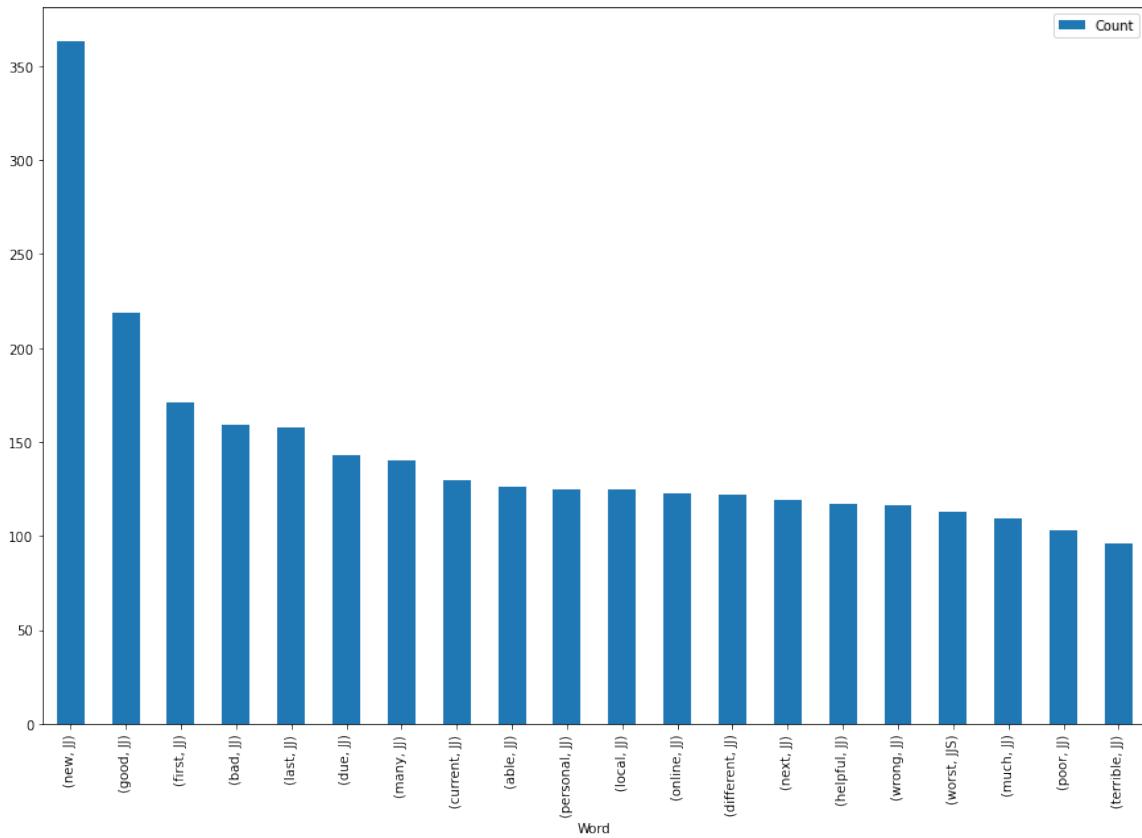


Figure 4: Bar-chart of word Frequency with Specific PoS Tag

In this case we are going to use the ‘Vader’ sentiment analysis tool:

<https://github.com/cjhutto/vaderSentiment>

Add a cell that creates an object ‘vader’ which is instantiated from ‘SentimentIntensityAnalyzer()’

💡 Your code here

Experiment with this library using some sentences that positive and negative emotion:

💡 Your code here

```
happyStudent = this is an excellent Machine Learning workshop
vader.polarity_scores(happyStudent) =
{'neg': 0.0, 'neu': 0.619, 'pos': 0.381, 'compound': 0.5719}
```

```
grumpyStudent = This Machine Learning workshop is terrible!
vader.polarity_scores(grumpyStudent) =
{'neg': 0.404, 'neu': 0.596, 'pos': 0.0, 'compound': -0.5255}
```

Four results are provided for each sentence: + The level of negative emotion + The level of neutral emotion + The level of positive emotion + A ‘score’ running between -1 and 1, that combines all of the above into a single metric.

The first three results are included as separate items since some sentences may express some positive and some negative emotion. This ‘bi-modal’ response might be lost in a overall average.

Experiment with the above two code blocks. Create other sentences and develop an understanding of the scoring that vader is applying.

Now, add a sentiment analysis score to each of the original items in our customer review dataframe within a feature called ‘scores’.

💡 Tutor provided code

```
pd.set_option('display.max_colwidth', 50)
df['scores'] = df['complaintText'].apply(lambda complaintText:
                                         vader.polarity_scores(text = complaintText))
df.iloc[0:5, 2:4]
```

	complaintText	scores
0	As a disabled person in an...	{'neg': 0.057, 'neu': 0.94...

	complaintText	scores
1	Bad customer service in br...	{'neg': 0.079, 'neu': 0.83...
2	I visited the Bognor Regi...	{'neg': 0.098, 'neu': 0.90...
3	Abysmal business service....	{'neg': 0.1, 'neu': 0.855,...
4	I have company account wi...	{'neg': 0.113, 'neu': 0.85...

Unfortunately it is difficult to read all of the text in the ‘scores’ feature as it is truncated.

Add a cell to print an example of the full text.

💡 Your code here

```
df['scores'][0] = {'neg': 0.057, 'neu': 0.943, 'pos': 0.0, 'compound': -0.5848}
```

Add a cell that extracts the value of the ‘compound’ score, and saves this into a separate column in the dataframe.

💡 Tutor provided code

```
df['compound'] = df['scores'].apply(
    lambda score_dict: score_dict['compound'])
df.iloc[0:4, 2:]
```

	complaintText	scores	compound
0	As a disabled person in an...	{'neg': 0.057, 'neu': 0.94...	-0.5848
1	Bad customer service in br...	{'neg': 0.079, 'neu': 0.83...	-0.1531
2	I visited the Bognor Regi...	{'neg': 0.098, 'neu': 0.90...	-0.9770
3	Abysmal business service....	{'neg': 0.1, 'neu': 0.855,...	-0.7853

It is interesting to consider the most positive and most negative sentiment expressed. We can now do that simply by sorting the dataframe on the ‘compound’ score column.

💡 Tutor provided code

```
sortedSentiment = df.sort_values('compound')
sortedSentiment.iloc[0:5, 2:]
```

	complaintText	scores	compound
207	Bunch of crooks Finally ...	{'neg': 0.12, 'neu': 0.835...}	-0.9962
696	Awful bank! They still ho...	{'neg': 0.282, 'neu': 0.68...}	-0.9955

	complaintText	scores	compound
190	Absolutely appalling expe...	{'neg': 0.122, 'neu': 0.85...	-0.9952
662	I have been a BANK_NAME Cu...	{'neg': 0.171, 'neu': 0.79...	-0.9945
1663	I've banked with BANK_NAM...	{'neg': 0.168, 'neu': 0.76...	-0.9944

It is instructive to look at a visualisation that shows the distribution of sentiment.

Add a cell that uses the ‘hist’ (histogram) method of a pandas dataframe to display a histogram of the ‘compound’ feature of df. Display the data 10 bins

💡 Your code here

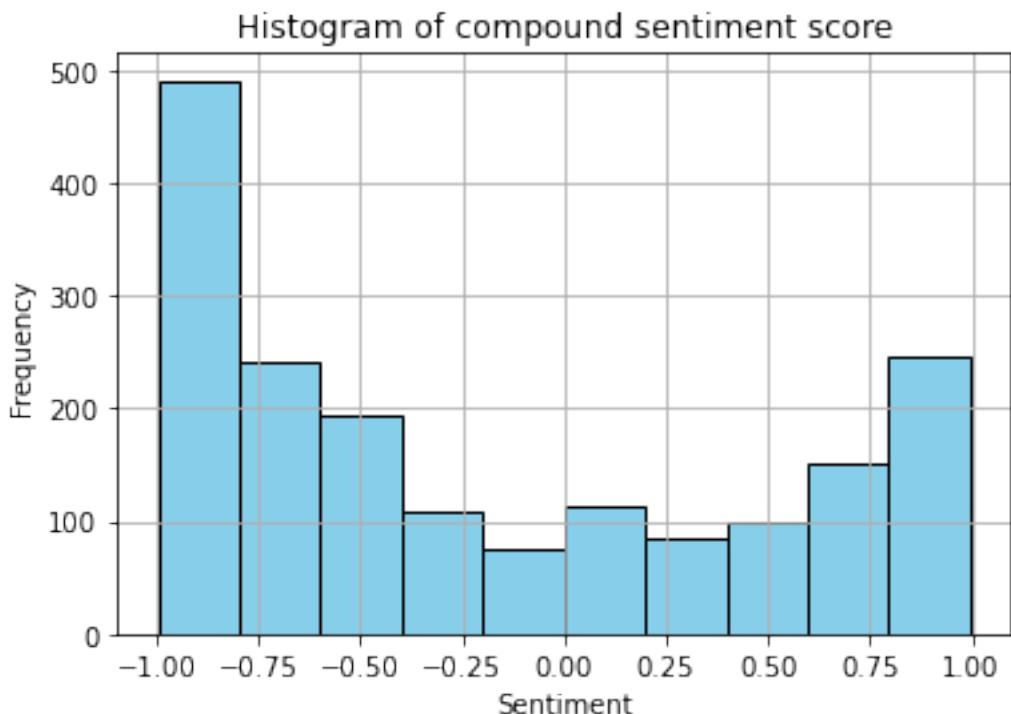


Figure 5: Histogram of compound sentiment score for bank reviews

This appears to be a ‘bi-modal’ distribution - which is what I would expect. People tend to express extreme views on review websites. They are motivated to contribute to review websites because they are either very happy with the service or very unhappy. Few people take the time to contribute to a review service in order to express neutral views.

If I were the bank, then I might worry that there is much more negative sentiment expressed than positive sentiment. However, statistically speaking and thinking critically we might again consider this as an example of sampling bias.

Experiment with the number of bins in the above chart. In your view, what is the most useful number of bins in terms of finding insights in the data?

10.9 Identifying themes in text

Finally in this workshop we are going to consider the problem of extracting ‘themes’ from large bodies of text. That is, are their ‘clusters’ of words that somehow express a common ‘theme’ or set of ideas. Simply put, are their groups of things that our customers tend to be talking about?

One way of approaching this task is called ‘Non-negative Matrix Factorization’ (NMF) - and this was explained in the lecture class. Below, we apply NMF to the bank reviews.

First, we return to the ‘bag of words’ representation described in section 2 above. We can create a Pandas dataframe that includes the vocabulary words as column names the the frequency of that words occurrence for each of the customer review - all 1805 of them.

Add a cell to re-display part of the ‘bow_pd’ dataframe for illustrative purposes.

💡 Your code here

	active	activity	actual	actually	add	added	additional	address	admit
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

We can make the printing of results neater if we define a short function to print results for us:

💡 Tutor provided code

```
def display_topics( model, feature_names, num_top_words):
    for ix, topic in enumerate(model.components_):
        print("\nTopic ", ix)
        print(", ".join([feature_names[i]
                        for i in topic.argsort()[:-num_top_words -1:-1]]))
```

The Non-negative Matrix Factorization algorithm is included within sklearn and it takes only a single function call to apply the function to the data:

 Tutor provided code

```
n_themes = 10                      # Number of themes I want to identify
n_iterations = 1000                  # Number of iterations before timing-out
n_top_words = 15                      # Number of relevant words in each theme

nmf_model = NMF(n_components=n_themes,
                 random_state=1,
                 alpha_W=0,
                 alpha_H=0,
                 l1_ratio=.5,
                 init='nndsvd',
                 max_iter = n_iterations).fit(bow_pd)
```

And then display the results:

 Tutor provided code

```
display_topics(nmf_model, vectorizer.get_feature_names_out(), n_top_words)
```

Topic 0

bank_name, company, accounts, use, years, investor, say, problem, smart, took

Topic 1

account, open, opened, closed, close, current, opening, reason, direct, went

Topic 2

card, debit, new, sent, address, pin, received, blocked, cards, home

Topic 3

bank, went, team, transaction, pay, worst, want, like, closed, week

Topic 4

branch, appointment, local, address, went, needed, id, centre, called, change

Topic 5

money, pay, paid, account, payment, transfer, cash, want, people, don

Topic 6

service, customer, years, customers, terrible, poor, team, rude, services, 20

Topic 7

business, personal, accounts, banking, years, manager, know, need, don, number

Topic 8

mortgage, property, new, solicitor, woolwich, weeks, application, years, having, house

Topic 9

banking, online, telephone, access, transfer, pin, tried, need, app, accounts

Topic 10

said, asked, went, didn, did, know, customer, manager, open, let

Topic 11

phone, number, hold, minutes, calls, security, hour, contact, department, hours

Topic 12

credit, years, pay, overdraft, loan, customers, month, paid, balance, limit

Topic 13

just, got, wasn, letter, little, really, saying, don, department, actually

Topic 14

time, minutes, times, help, long, waste, tried, banks, wait, 30

Topic 15

complaint, letter, received, financial, called, ombudsman, complaints, did, going, team

Topic 16

told, tried, people, went, wouldn, rang, appointment, away, received, sort

Topic 17

ve, fraud, don, years, going, work, transactions, really, password, visa

Topic 18

staff, people, customers, like, disabled, manager, stated, member, asked, wait

Topic 19

days, payment, called, day, new, working, later, details, weeks, 10

You will find that you obtain rather different results depending on the hyper-parameters you set for the NMF function, and also depending on the number of themes you wish to extract. It will often take some experimentation to find useful or interesting themes.

A common criticism for Non-Negative Matrix Factorization for text is that it sometimes produces results that are hard to interpret. They are ‘mathematically’ correct - and yet lack straight-forward semantic interpretation. In this case the results are certainly not conclusive, but they do point towards some clusters of ideas in the review dataset.

Chapter 11 - Gaussian Mixture Models

No-Code' Version for Student Practical Classes

(c) Dr Rob Collins 2023

2023-05-08

Table of contents

11 Gaussian Mixture Models	11 - 4
11.1 Introduction	11 - 4
11.2 Instructions for Students	11 - 4
11.3 Importing required libraries	11 - 5
11.4 Manual fitting of a Gaussian Mixed Model (GMM)	11 - 5
11.4.1 Loading the source data	11 - 5
11.4.2 Generating data to experiment with fitting	11 - 7
11.4.3 Experimenting to match synthetic data to given data-set	11 - 10
11.5 GMM the easy way - using sklearn	11 - 12
11.6 Optional (but fun!) - Astrophysics example	11 - 14
11.7 References:	11 - 17
11.8 Fitting a Gaussian Mixture Model Algorithmically	11 - 17
11.8.1 Determining the value of a Gaussian for a specific x value	11 - 17
11.8.2 The core GMM algorithm	11 - 19

List of Figures

1 Johann Carl Friedrich Gauss - the so called 'Prince of Mathematicians' engrossed in the third iteration of the now famous 'mixture model'	11 - 3
2 kde plot of the mixed population data	11 - 7
3 Histogram of the mixed population data illustrating the impact of a poor choice of bucket count	11 - 8
4 kde plot 1st synthetic population	11 - 9
5 kde plot for each synthetic population plotted separately	11 - 10
6 kde plot of the synthetic combined data	11 - 11
7 Example of a Gaussian Mixture Model generated manually	11 - 12
8 kde plots for original 'mixed_population' data set and synthetic data-set composed of two Gaussians	11 - 13
9 kde plots for GMM model for 'mixed_population' generated by sklearn	11 - 15

10 Synthetic data suggesting an image of two star clusters 11 - 16

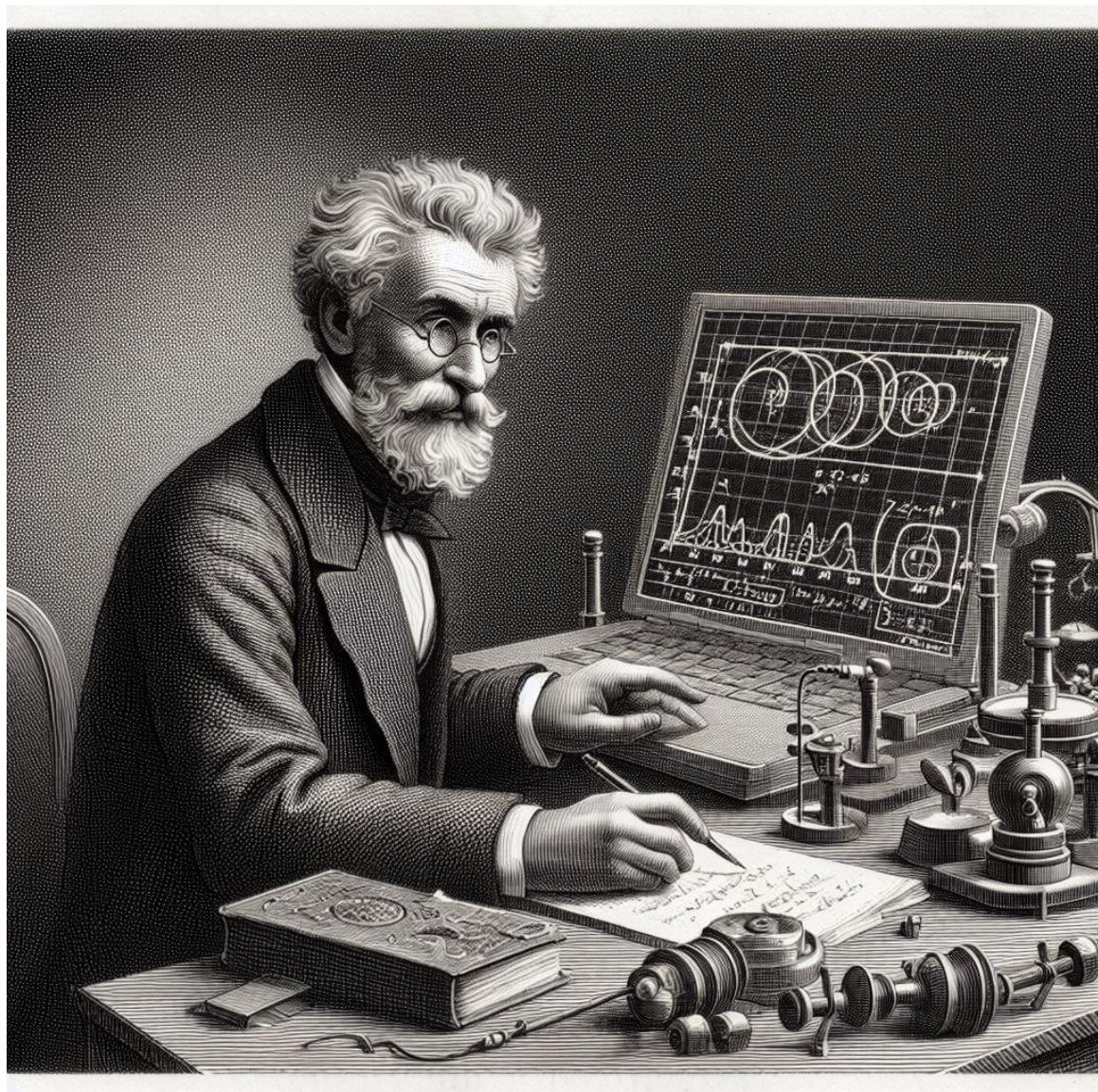


Figure 1: Johann Carl Friedrich Gauss - the so called ‘Prince of Mathematicians’ engrossed in the third iteration of the now famous ‘mixture model’

11 Gaussian Mixture Models

11.1 Introduction

In this workshop we will be experimenting with fitting multiple, over-lapping Gaussian Models to data-sets.

Gaussian Mixture Models have been used to model scientific data in diverse domains including medicine, biology and astro-physics. They are useful when dealing with data that may include multiple populations - but we don't know the characteristics of those populations.

Thus, we may have a data-set of what we believe is a single population - for example a collection of biological specimens that appear superficially to be the same. However, when we look more closely at the data it may become apparent that a better model is achieved by assuming that we actually have multiple populations, each with their own Gaussian Distribution. That is, each population has a set of variables that are characterised by a mean (average) a standard deviation and follow a characteristically 'normal' (bell-shaped) population.

The task is to recreate the most likely set of individual, overlapping Gaussian distributions from a mixed population.

Note : Although there are many cells in this workshop activity you will see that many of them repeat. These include all of the cells that display graphs of the data we use. Thus you should find it easy to copy-paste section of earlier code into later sections.

11.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

 Your code here

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do

6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the ‘In [n]’ text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

11.3 Importing required libraries

As always we start by importing any required libraries. In this case, we will be using Numpy to generate some data-sets. Those data-sets will contain a mixture of multiple populations that each have a Gaussian Distribution. Initially we will be building our Gaussian Mixture Models (GMM) ‘by hand’, but later we will be using a algorithm that is part of the sklearn library to complete the task quickly and efficiently. We also use a KDE plot to display the data.

For this practical session we will need the following libraries:

- ‘pandas’ : Conventionally given the reference name ‘pd’
- ‘preprocessing’ from sklearn
- ‘metrics’ from sklearn
- ‘numpy’ : Conventionally given the reference name ‘np’
- ‘norm’ : from scipy.stats
- ‘GaussianMixture’ : ’from sklearn.mixture

 Your code here

11.4 Manual fitting of a Gaussian Mixed Model (GMM)

In this section we are going to attempt to fit a mixed Gaussian model ‘by hand’. I have created a data-set and it will be your task to create a simple GMM for the data.

11.4.1 Loading the source data

Add a cell to load the data from the csv file “Mixed population Data.csv” in a dataframe called ‘mixed_population’. Display the first 10 rows of the dataframe and a description of the data.

 Your code here

```
The_Data
count    982.000000
mean     80.265438
std      34.307295
min     -5.150000
25%     49.980000
50%     84.825000
75%    110.805000
max    147.660000
```

	The_Data
0	113.56
1	61.05
2	37.02
3	108.99
4	110.36
5	20.46
6	50.60
7	110.84
8	119.30
9	126.40

As you can see, this is a single column of data consisting of 981 data points. Such data might be, for example, a set measurements of the size of Turtle shells measured by a biologist. The question is .. how best to model this data?

Let's first take a look at the data and see if that can provide some clues.

Add a cell to plot a kde plot for the data.

💡 Your code here

We could have equally well plotted a more familiar histogram at this point. However, has been stated in previous practicals an issue is that they can provide a misleading visualisation. In the following cell I have illustrated this with a rather extreme example - plotting the data with only 5 buckets. But ‘too few buckets’ is only one of the artifacts that histograms are prone to; issues may also occur when there are too many buckets and sometimes when the bucket-boundaries coincide with specific peaks in data etc.

On the other hand, histograms are more familiar to a broader audience. If you employ kde plots in your analysis for a less expert audience you may have to explain and justify your choice of chart. You should make a critical judgment regarding your use of data display in each case.

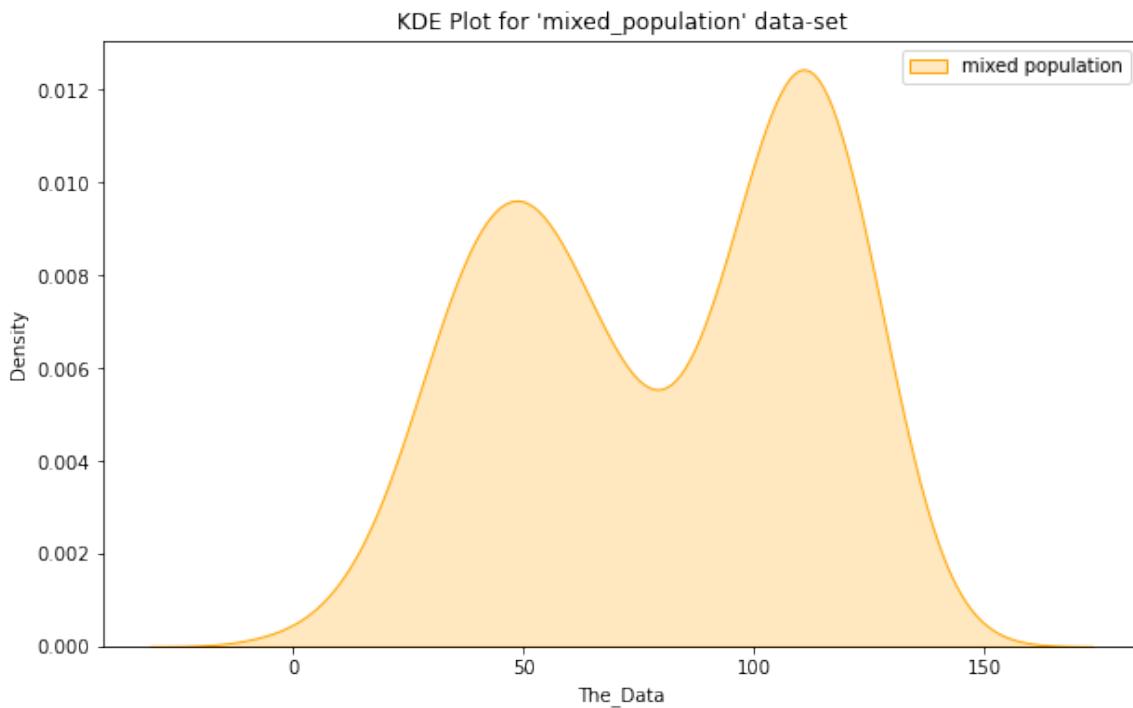


Figure 2: kde plot of the mixed population data

Your code here

The overall shape of the combined data does not appear to be Gaussian. This data appears to be ‘bi-modal’ : That is, there are two peaks. Each of these peaks may be Gaussian - but the data-set as a whole is not. This may suggest to us that we are looking at two (or more) separate populations. We may have collected data for two different species of turtle!

So the idea here is to generate multiple Gaussian Distributions (in this case 2) - and fit those to the peaks and troughs in the above distribution.

11.4.2 Generating data to experiment with fitting

In this section we are going to generate our own data-set using Python. Then later we will experiment with this data set to try to make it match the data in the ‘Mixed population Data.csv’ file

11.4.2.1 Data for the first Gaussian (‘Normal’) Distribution

Add a cell to define the number of samples to be generated .. in this case 2000

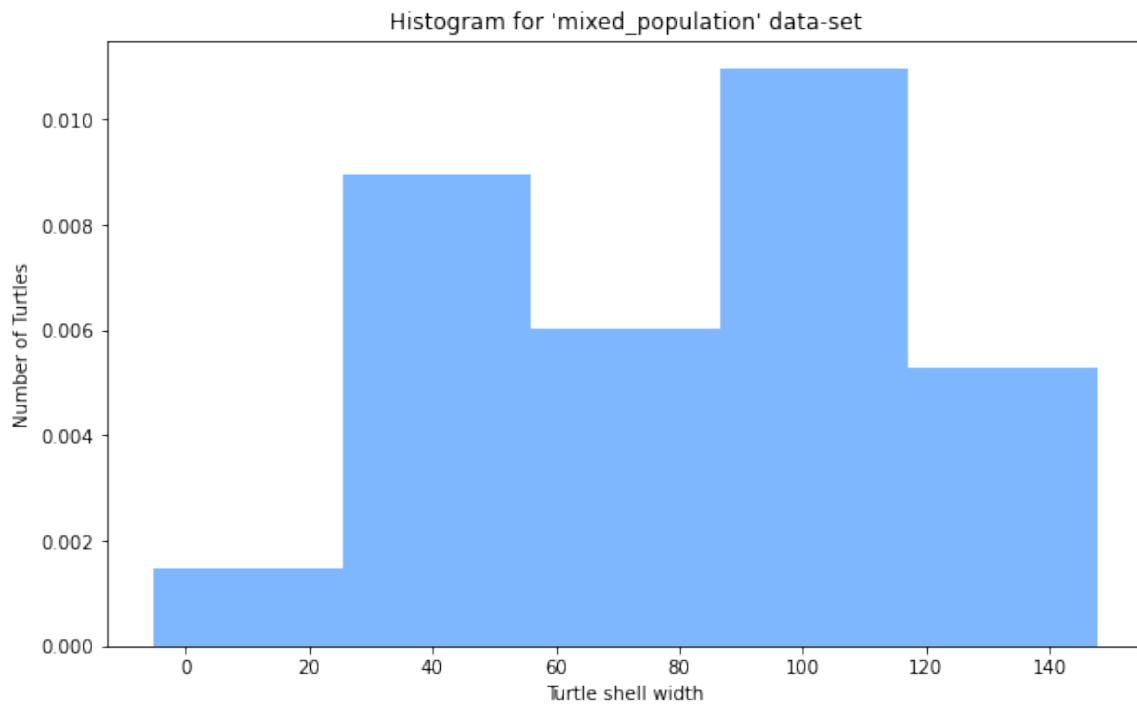


Figure 3: Histogram of the mixed population data illustrating the impact of a poor choice of bucket count

💡 Your code here

```
number_of_samples = 2000
```

Add a cell to define parameters for the first of the Gaussian Distributions. Start wth a mean of -0.5 and a standard deviation of 0.7

💡 Your code here

```
sample_mean = -5.0
sample_sd = 0.7
```

The synthetic data-set can be generated using `numpy.random.normal`:

💡 Tutor provided code

```
samples = np.random.normal(sample_mean, sample_sd, number_of_samples)
```

Note that Numpy uses the term ‘normal’ rather than the term ‘Gaussian’ - but they are the same thing.

11.4.2.2 Visualising the first population

Add a cell to visualise this data using a kde plot

Your code here

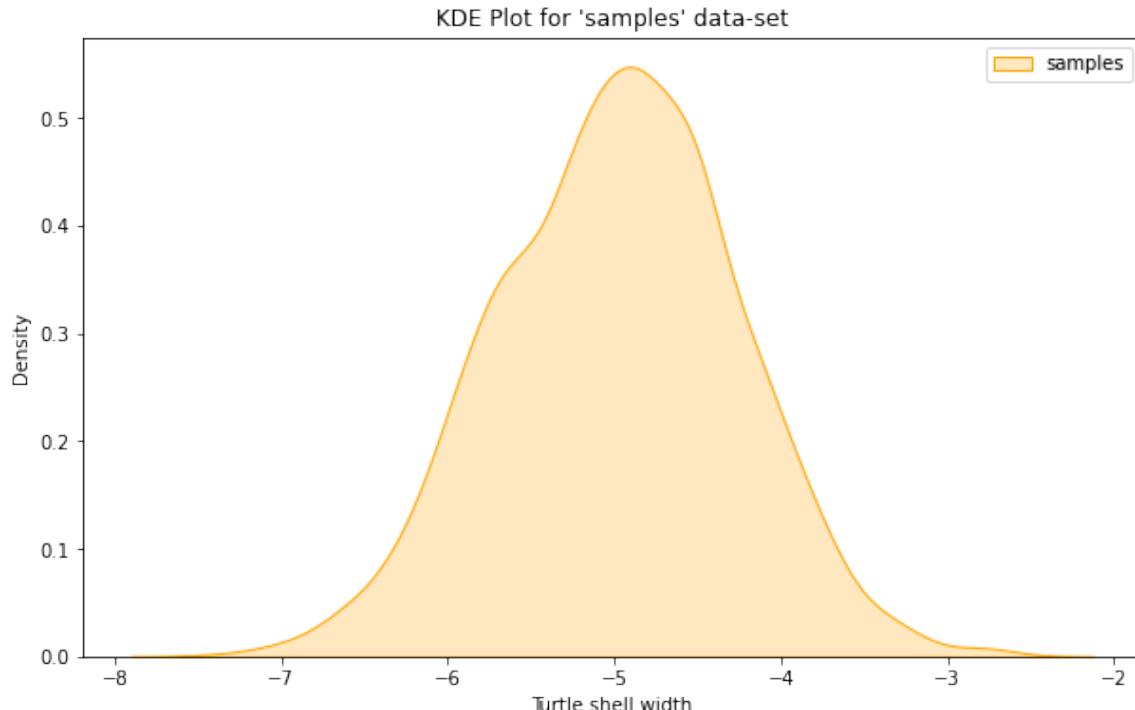


Figure 4: kde plot 1st synthetic population

11.4.2.3 Adding a second population

Add a cell to create a second synthetic distribution called ‘samples2’. Set the parameters as follows:

- Mean = 1
- Standard deviation = 0.3
- Number of samples = number_of_samples

Your code here

```
samples2.shape = (2000,)
```

Then add a cell to plot both data-sets on a combined graph:

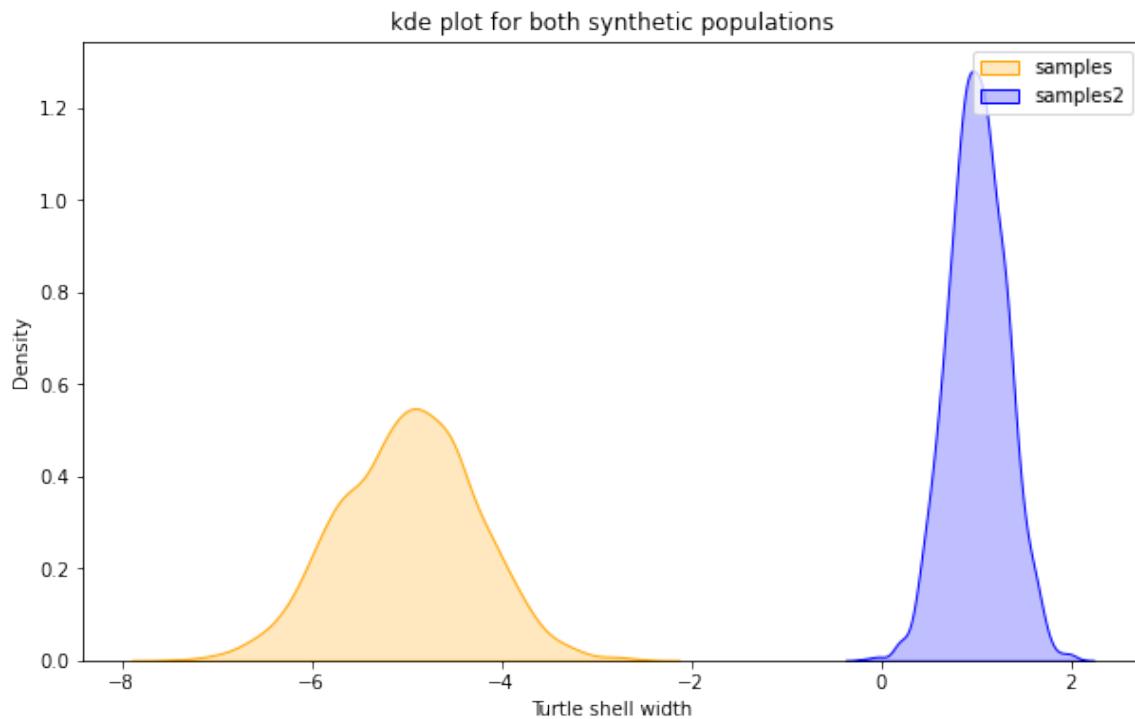


Figure 5: kde plot for each synthetic population plotted separately

At this point we have two, separate Gaussian Distributions. But in the original data the two populations are combined. We can mimic that by combining our two distributions.

Add a cell to that creates a new numpy array called ‘combined_samples’ by appending ‘samples2’ to ‘samples’)

Your code here

Then add a cell that creates a kde plot for the ‘combined_samples’ synthetic data-set. This is the first attempt at replicating the bi-modal data from the original data-set.

11.4.3 Experimenting to match synthetic data to given data-set

So we have created a bi-modal distribution which is the combination of two Gaussian distributions. It has broadly the same shape as our original data-set (two peaks), but the size and shape of those peaks does not match our original data.

We need to do some experiments to try to make the data-match.

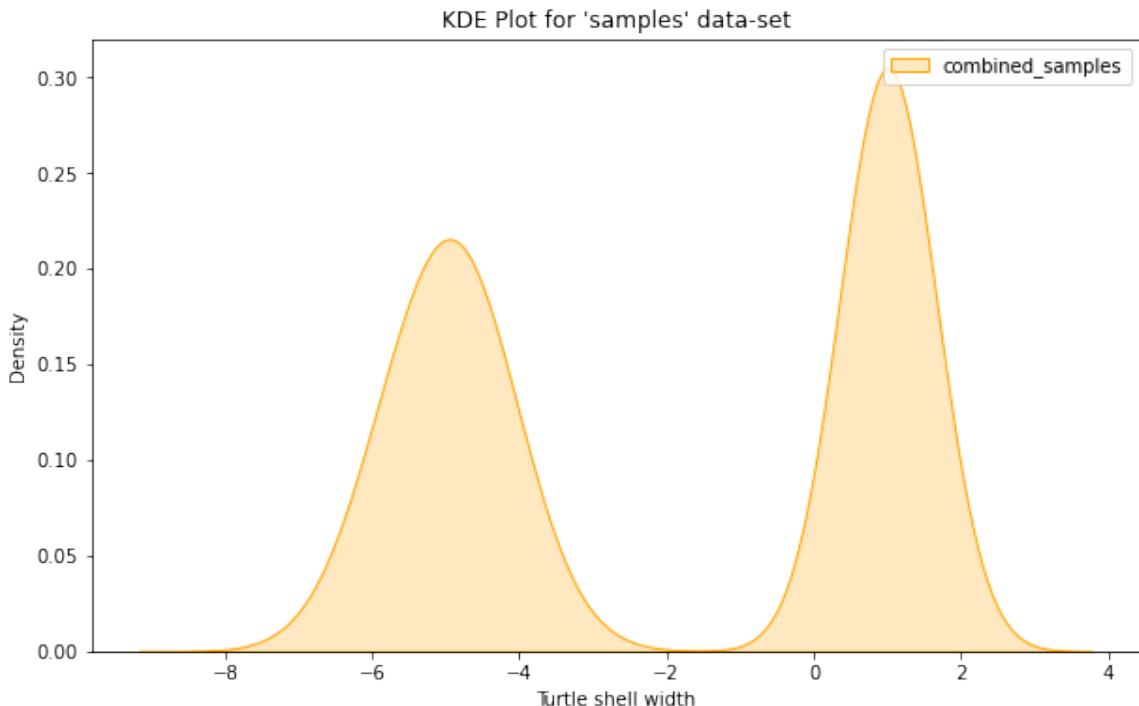


Figure 6: kde plot of the synthetic combined data

Experiment by modifying the parameters (numbers) in the following cell to try to make your generated data match that of the original collected data. The cell below provides a combined plot of your generated data and the original data I supplied. Note that the match will never be perfect. After all, the data is generated randomly. However, you should be able to create a reasonably good match.

The graphic below is the result I obtained after some experimentation. You should try to produce something like this.

Create a single cell that replicates some of the above code into a single cell:

- Create a numpy array called ‘samples’ containing data for a Gaussian distribution, with a given mean and standard deviation and size
- Create a second numpy array called ‘samples2’ containing data for another Gaussian distribution with a different mean, standard deviation and size
- Create a combined data-set called ‘combined_samples’ by appending ‘samples2’ to ‘samples’

Your code here

Create a numpy array called ‘mixed_population_np’ from the ‘The_Data’ column of the original ‘mixed_population’ dataframe (we need this because it makes plotting the kde plot easier in the following cells).

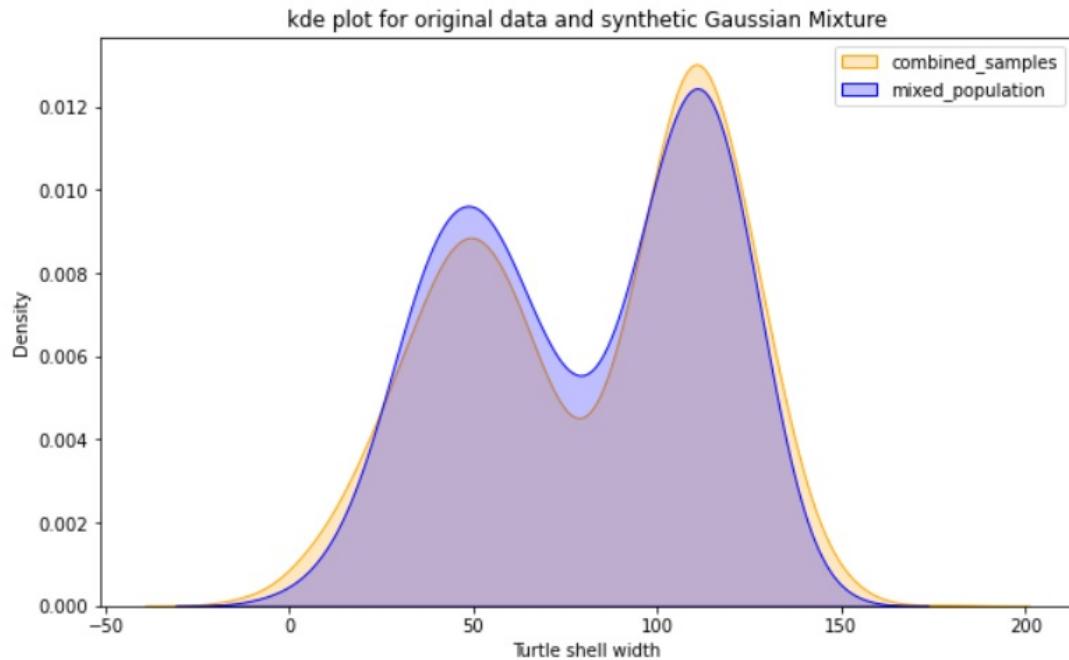


Figure 7: Example of a Gaussian Mixture Model generated manually

Your code here

Now plot a kde chart of both the ‘mixed_population’ data (‘mixed_population_np’) and the synthetic, bi-modal data just generated (‘combined_samples’).

Your code here

11.5 GMM the easy way - using sklearn

Having worked created a Gaussian Mixture Model manually, you will be pleased to know that sklearn includes a library function that implements a fast version of the complete algorithm. It is almost trivially easy to fit the required model.

Tutor provided code

```
gmm = GaussianMixture(n_components=2, covariance_type="full", tol=0.001)
gmm = gmm.fit(X=mixed_population)
```

Now add cells to print the modelled mean values:

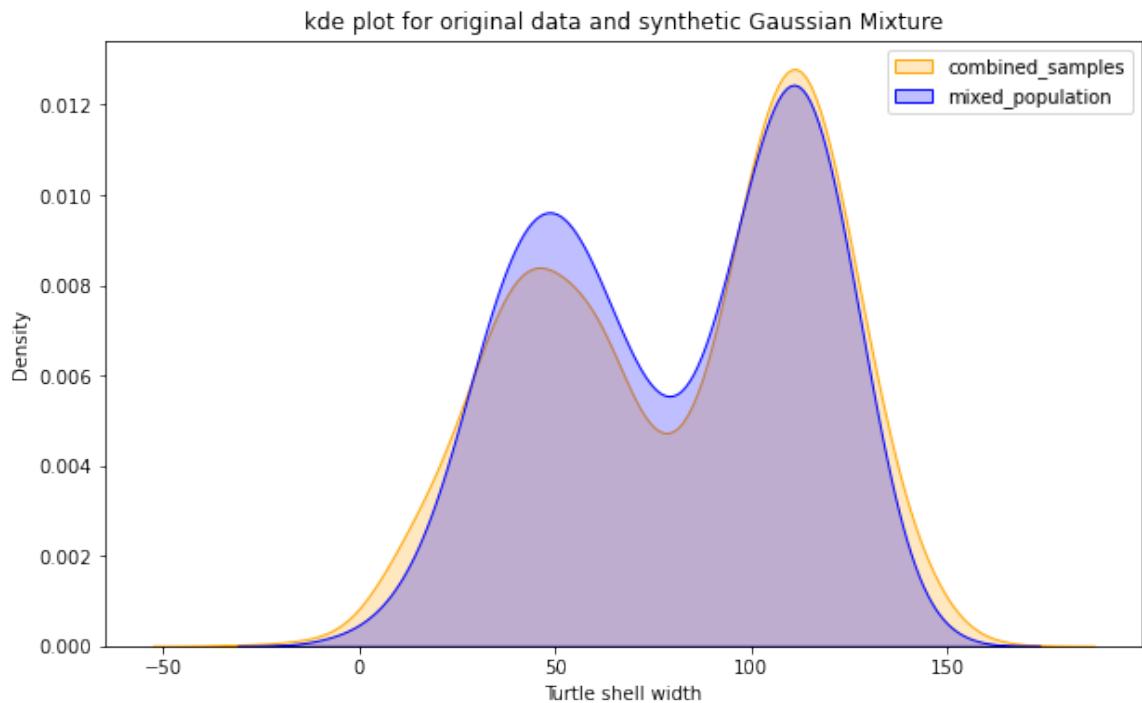


Figure 8: kde plots for original ‘mixed_population’ data set and synthetic data-set composed of two Gaussians

 Your code here

```
gmm.means_ =
[[109.5768361 ]
 [ 48.85430785]]
```

Print the Standard Deviations:

 Your code here

```
np.sqrt(gmm.covariances_) =
[[[14.07012376]
 [17.7847952 ]]]
```

Print the weights (that is, the proportion of items in each population). In the case of our data, their are an equal numer of items of each sub-population in the total population. Of course, that will often not be the case, and we need to recover the proportion in each.

```
gmm.weights_ =  
[0.51728956 0.48271044]
```

As above, let's visualise this result.

Note that the ‘item()’ function is used in Numpy to extract a single value from an array. This is required since without it the result would be returned as a single item array.

 Tutor provided code

```
mean_1 = gmm.means_[0].item()  
mean_2 = gmm.means_[1].item()  
  
sd_1 = np.sqrt(gmm.covariances_[0]).item()  
sd_2 = np.sqrt(gmm.covariances_[1]).item()  
  
totalPopulation = 4000  
  
# Numpy requires an integer for the 3rd  
# parameter of 'np.random.normal'  
weight_1 = int(totalPopulation * gmm.weights_[0].item())  
weight_2 = int(totalPopulation * gmm.weights_[1].item())
```

As previously, generate two synthetic data-sets based on these values, combine the results and plot them:

 Your code here

Then plot the result:

11.6 Optional (but fun!) - Astrophysics example

In this final section I am going to demonstrate how we might apply GMM to a problem in Astrophysics. We are going to use the model to locate the centre of galaxies from photographs.

(If you are less interested in the stars - then you might prefer to imagine this example as finding the centre of cells in medical images or possibly the centre of populations from archeological data. GMM is a very flexible tool!"

First lets create some ‘image data’. You have to imagine that this was really obtained from astro-photography. In this case we are creating the data since it enables you to experiment with different images and to decide how well you think it works.

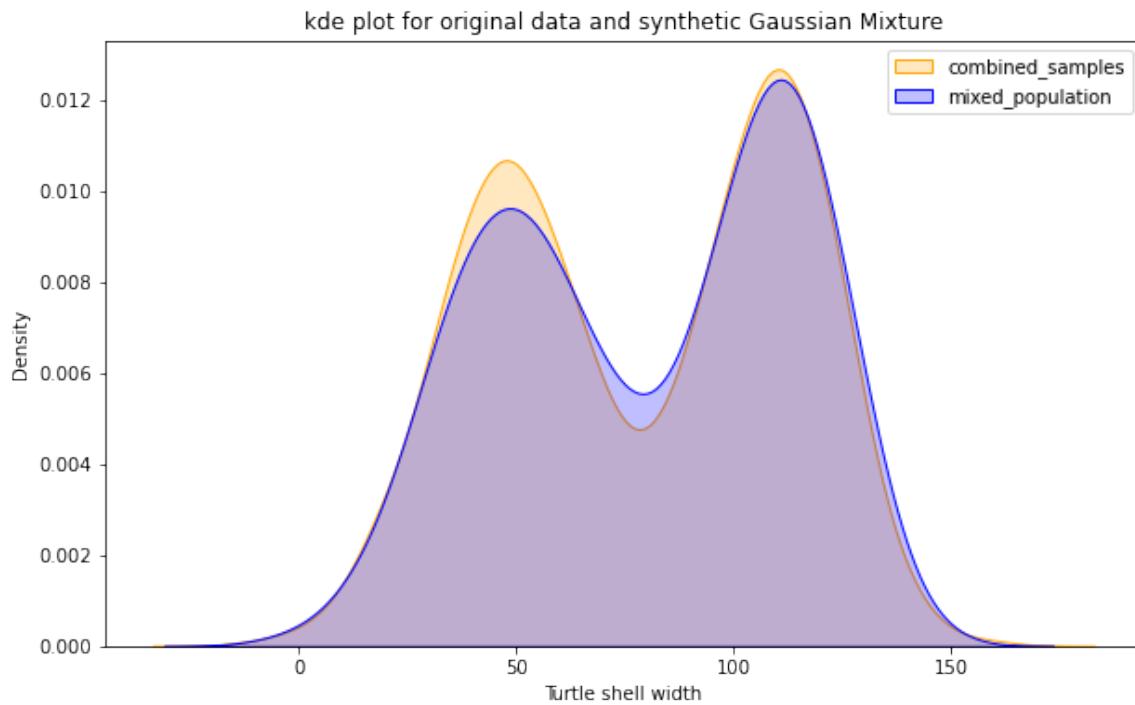


Figure 9: kde plots for GMM model for ‘mixed_population’ generated by sklearn

First we decide the number of ‘dots’ in our images. These will be points of light that have been detected by our imaging system. To start the experiment I am going to use 3000 points, but you may experiment with this value.

Add a cell to define a variable ‘n_samples’ with a value of 3000, then print it.

Your code here

```
n_samples = 3000
```

Our first galaxy is purely spherical. It just appears as a circle of points in our final image.

To start, I have located this galaxy at coordinates 5,5. You should experiment with these values to see how well the GMM detects the centre of each galaxy.

Tutor provided code

```
galaxy_1 = np.random.randn(n_samples, 2) + np.array([5, 5])
```

Our second galaxy is not spherical. It is a disk galaxy and we are viewing it from a slight angle.

 Tutor provided code

```
C = np.array([[0., -0.7], [3.5, .7]])
galaxy_2 = np.dot(np.random.randn(n_samples, 2), C)
```

We combine all of this data into a single data-set to form our ‘photograph’:

 Tutor provided code

```
astro_photograph = np.vstack([galaxy_1, galaxy_2])
```

Now we can visualise this data-set. I have set the colours to make it look like a photograph of the night sky through a powerful telescope.

 Tutor provided code

```
ax = plt.axes()
ax.set_facecolor("black")
plt.scatter(astro_photograph[:, 0],
            astro_photograph[:, 1],
            c="white",
            s=1)
plt.show()
```

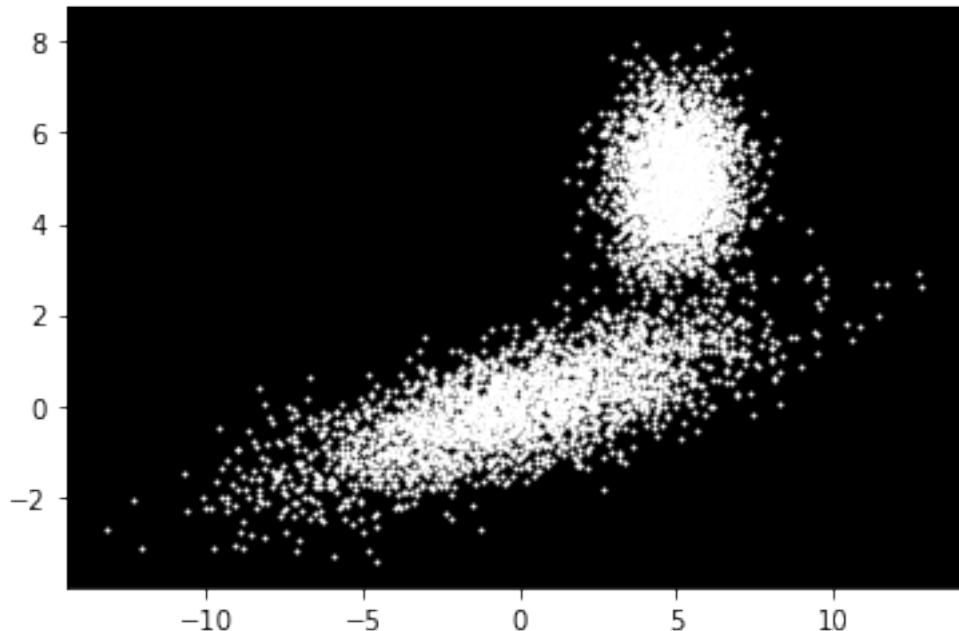


Figure 10: Synthetic data suggesting an image of two star clusters

Of course, in reality, the photograph is all we would have .. we would not know the true

centres of these two galaxies. We could use a GMM to model the data and to decide the best estimate of where the centres are:

 Tutor provided code

```
gmm = GaussianMixture(n_components=2, covariance_type="full", tol=0.001)
gmm = gmm.fit(X=astro_photograph)
```

What is the centre of each galaxy based on the results from the model?

 Tutor provided code

```
galaxy_1_modelled_center = gmm.means_[0]
print("Centre of galaxy 1 modelled as ", galaxy_1_modelled_center)

galaxy_2_modelled_center = gmm.means_[1]
print("Centre of galaxy 2 modelled as ", galaxy_2_modelled_center)
```

Centre of galaxy 1 modelled as [-0.09628218 -0.00336387]

Centre of galaxy 2 modelled as [4.97927821 4.99314605]

Student task: Experiment by changing the location of the two galaxies my modifying the values in the cell labelled ‘Student task step 4’. To what extent is the GMM able to recover the true centre of the two galaxies?

11.7 References:

The ‘astro-physics’ section of this workshop is based loosely on the sklearn documentation here:

https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_pdf.html#sphx-glr-auto-examples-mixture-plot-gmm-pdf-py

11.8 Fitting a Gaussian Mixture Model Algorithmically

This section is unfinished and TBD

11.8.1 Determining the value of a Gaussian for a specific x value

We will first need a function that computes a continuous Gaussian random variable. This will be used to compute the probability of getting a particular value within a distribution with a specified mean and standard deviation. Such a function is available within ‘scipy.stats’.

The function below answers this question: If we have a Gaussian Distribution with mean = 7 and a standard deviation of 2, what would be the probability of selecting the value ‘4’ at random.

 Tutor provided code

```
x_value = 10
mean = 7
standard_dev = 2
print(f"norm.pdf(x, loc=mean, scale=standard_dev) = ")
print(f"    {norm.pdf(x_value, loc=mean, scale=standard_dev):.3f}")
```

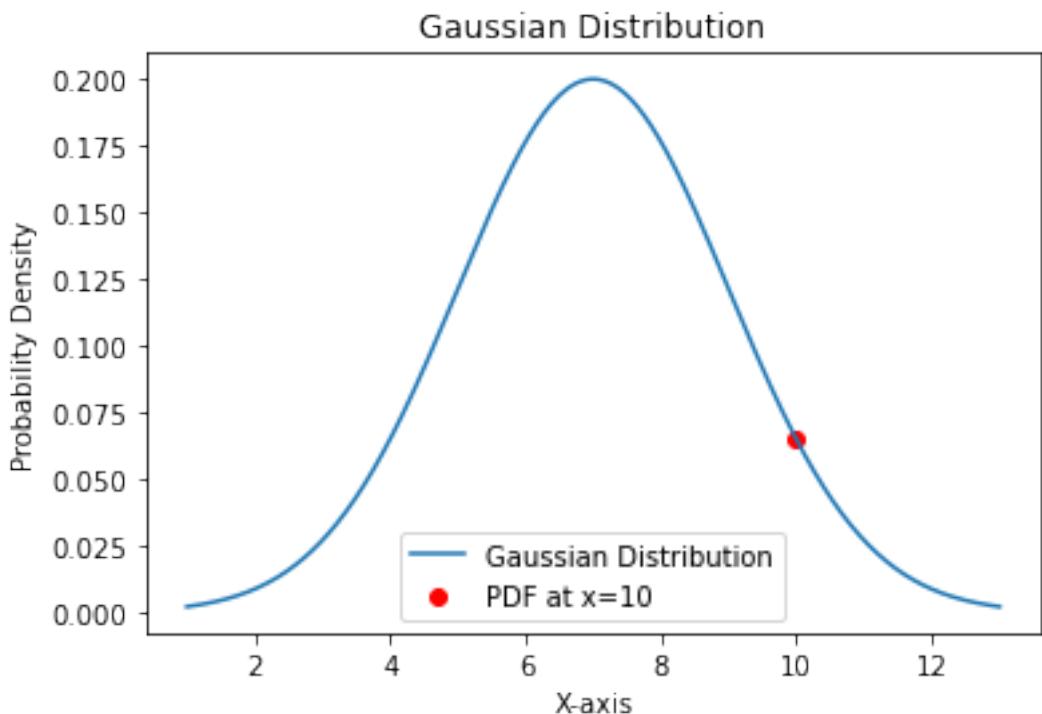
```
norm.pdf(x, loc=mean, scale=standard_dev) =
0.065
```

This can be illustrated as follows:

 Tutor provided code

```
x = np.linspace(mean - 3*standard_dev, mean + 3*standard_dev, 1000)
pdf_values = norm.pdf(x, loc=mean, scale=standard_dev)

plt.plot(x, pdf_values, label='Gaussian Distribution')
plt.scatter([x_value], [norm.pdf(x_value, loc=mean, scale=standard_dev)], color='red', l
```



11.8.2 The core GMM algorithm

Define ‘k’ to be the number of populations that we want to model .. in this case 2

Your code here

```
k = 2
```

The algorithm allows us to model mixed populations of different sizes. The proportion of the total population in each sub-population is defined in the ‘weights’ variable. This is initialised with the assumption that there is an equal number of members of each population

Tutor provided code

```
weights = np.ones((k)) / k
weights
```

```
weights = [0.5 0.5]
```

We also need to initialise the mean value and the variance (square of the standard-deviation) for each population. These are arbitrary values - since the algorithm will update these

values step-by-step. Normally these values would be set as random numbers. However, in this case I am going to first set the values to be known, fixed values. This will ensure that we get similar results when demonstrating the algorithm in class.

You can experiment with setting the initial values to some random values.

 Tutor provided code

```
means = np.array([10, 30])
variances = np.array([1, 1])

print(f"means = {means}")
print(f"variances = {variances}")
```

```
means = [10 30]
variances = [1 1]
```

'eps' below is just a small number intended to prevent a 'divide by zero' error in the following algorithm

 Tutor provided code

```
eps=1e-8
```

Just for reference we will maintain a count of the number of iterations that we execute before completion:

 Tutor provided code

```
iterationCount = 0
```

The following cell implements the core of the algorithm.

Note that this cell only implements a single iteration of the algorithm. To obtain a good model, the following cell must be executed multiple times. I had to execute the following cell around 80 times to obtain a good model. I created the code in this way so that you could watch the model evolving towards a good solution.

Execute the following cell ('Student task step 1' multiple times and look at the printed results as the converge on a good model. You may also execute the following two cells ('Student task step 2' and 'Student task step 3' to visualise the evolving model

Optional student task : After a while you may get bored of repeatedly executing the following cell by hand. If you wish you may modify the following code to add an outer loop that executes the code multiple times.

 Tutor provided code

```
# Student task step 1

iterationCount = iterationCount + 1
print("Iteration: ", iterationCount)

# Initialise the vector of likelihood values
likelihood = []

# Expectation step
for j in range(k):
    # print("Means[j]", means[j], "Standard Deviation", np.sqrt(variances[j]))
    # likelihood.append(norm_dist(X, means[j], np.sqrt(variances[j])))
    likelihood.append(norm.pdf(x, loc=means[j], scale=np.sqrt(variances[j])))

likelihood = np.array(likelihood)

b = []

# Maximization step
for j in range(k):
    print("Population: ", j+1)
    # use the current values for the means and standard deviations to compute
    # the likelihood that each point would be within that population
    b.append((likelihood[j] * weights[j]) /
              (np.sum([likelihood[i] * weights[i] for i in range(k)], axis=0)+eps))

    # update mean and variance
    means[j] = np.sum(b[j] * X) / (np.sum(b[j]+eps))
    print("      Mean : ", means[j])
    variances[j] = np.sum(b[j] * np.square(X - means[j])) / (np.sum(b[j]+eps))
    print("      Standard deviation :", np.sqrt(variances[j]))

# update the weights
for j in range(k):
    weights[j] = np.mean(b[j])

print("weights[i] :", [weights[i] for i in range(k)])
```

We can visualise the generated model by generating a ‘synthetic’ distribution as in section 1:

Chapter 12 - Tutor Demonstrations 1

Naïve Approaches to Model Fitting

(c) Dr Rob Collins 2023

2023-03-05

Table of contents

12 Naïve Approaches to Model Fitting	12 - 3
12.1 Generating data which we can use to experiment with fitting	12 - 3
12.2 Various Naïve attempts at creating a model for this data	12 - 5
12.2.1 Method 1 : Manual, human ‘by inspection’	12 - 5
12.2.2 Method 2 : Measuring the gap (error) between the model and the data-points	12 - 7

List of Figures

1	Medieval craftsmen deliberating over the optimum dimensions for straight beams to be fitted to badly aligned stone fittings	12 - 2
2	Linear data with Gaussian Noise	12 - 5
3	Comparing the original data and the model	12 - 6
4	Comparing the original data and the model	12 - 8



Figure 1: Medieval craftsmen deliberating over the optimum dimensions for straight beams to be fitted to badly aligned stone fittings

12 Naïve Approaches to Model Fitting

12.1 Generating data which we can use to experiment with fitting

As always we start by importing any required libraries. In this case, we are going to generate some data that essentially falls on a straight line, but also has a small amount of ‘noise’ (randomness).

First import the ‘numpy’ and ‘random’ libraries

```
1 import numpy as np
2 import random
```

The following function generates a set x and y data. The y data has a slope defined by ‘A’, a constants term (a ‘bias’ defined by ‘K’ and a level of random noise defined by ‘varience’.

```
1 def genData(numPoints, A, K, variance):
2     x = np.zeros(shape=numPoints)
3     y = np.zeros(shape=numPoints)
4     for i in range(0, numPoints):
5         x[i] = i
6         y[i] = (A * i + K) + random.uniform(-1, 1) * variance
7     return x, y
```

Generate some random parameters for our linear data:

Create and set the following variables:

- ‘A’ as a random number between -20 and 20.
- ‘k’ to be a random number between -100 and 100
- ‘variance’ = int(A)

The last of these is arbitrary, but I want our variation (randomness) to be visible on the charts we draw and this helps us do that. You can actually experiment with different values for the variance if you wish.

```
1 A = random.uniform(-20, 20)
2 K = random.uniform(-100, 100)
3
4 # Following is arbitrary - but I wanted a variance
5 # that scaled with the slope to make it more
6 # visible on charts
7 variance = int(A*10)
```

Generate some linear data random parameters and some Gaussian noise. Use the ‘genData’ function defined above. Use ‘A’, ‘k’ and ‘variance’ as the parameters. Return the results into variables called ‘x’ and ‘y’

```
1 x, y = genData(numPoints = 100, A = A, K = K, variance = variance)
```

Review graphically:

Note : The ‘matplotlib inline’ command forces the chart to appear as part of your Jupyter notebook rather than in a separate window

```
1 %matplotlib inline
2 import seaborn as sns
3 import matplotlib.pyplot as plt
4
5 fig = plt.figure(figsize=(5, 5))
6 fig = sns.scatterplot(x = x, y=y, s = 100)
7 fig.set(xlabel = "x", ylabel = "y",
8         title ='Linear data with Gaussian Noise')
9 plt.legend(labels=['Data'])
10 plt.plot()
11 plt.show()
```

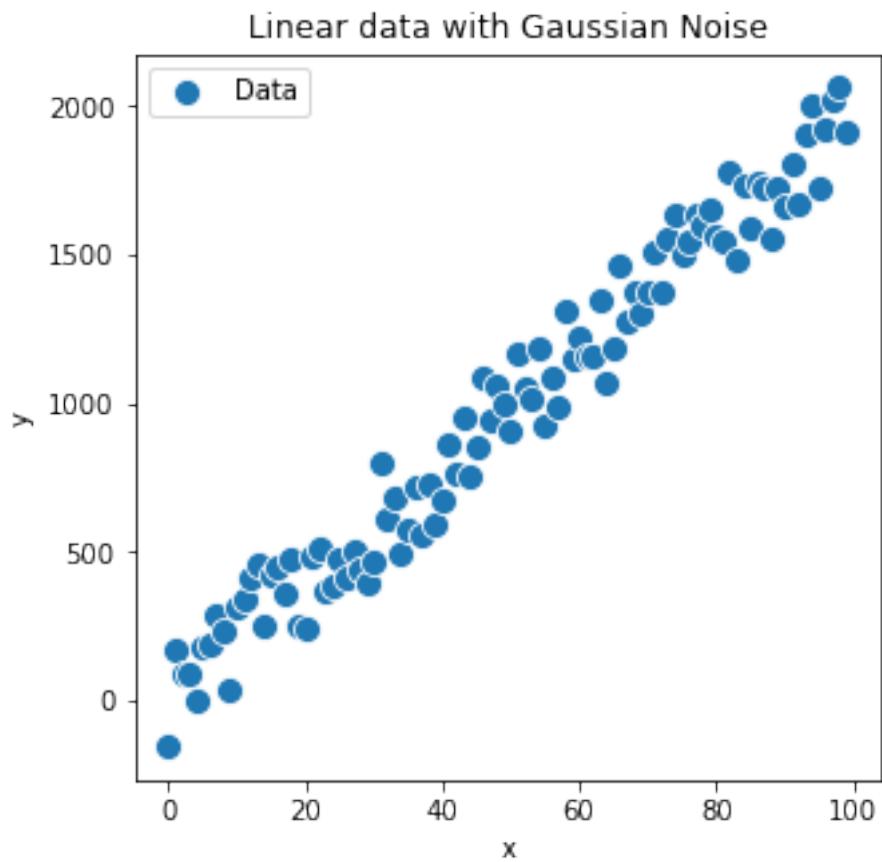


Figure 2: Linear data with Gaussian Noise

12.2 Various Naïve attempts at creating a model for this data

12.2.1 Method 1 : Manual, human ‘by inspection’

We are going to ‘guess’ the values ‘A’ and ‘k’ for a model of our data. For the model we can set the variance to be some small value so that we get a simple, linear set of data.

Create model parameter ‘A_model’ and ‘K_model’ with your guessed parameters.

Create a variable ‘variance_model’ = 0.00001 as we do not wish to add randomness to our model.

Use the function ‘genData’ again, to elaborate your model into a set of points that we can plot.

```

1 A_model = 5.
2 k_model = -100
3 variance_model = 1e-9
4
```

```

5 x_model, y_model = genData(numPoints = 100,
6     A = A_model,
7     K = k_model,
8     variance = variance_model)

```

Then copy the plotting functions from above and modify it to plot a scatterplot of both the original data and the new model data.

```

1 fig = plt.figure(figsize=(5, 5))
2 fig = sns.scatterplot(x = x, y=y, s = 100, legend = True)
3 fig = sns.scatterplot(x = x_model, y=y_model, s = 20, legend = True)
4 fig.set(xlabel = "x", ylabel = "y",
5         title ='Comparing the original data and the model')
6 plt.legend(labels=['Data', 'Model'])
7 plt.show()

```

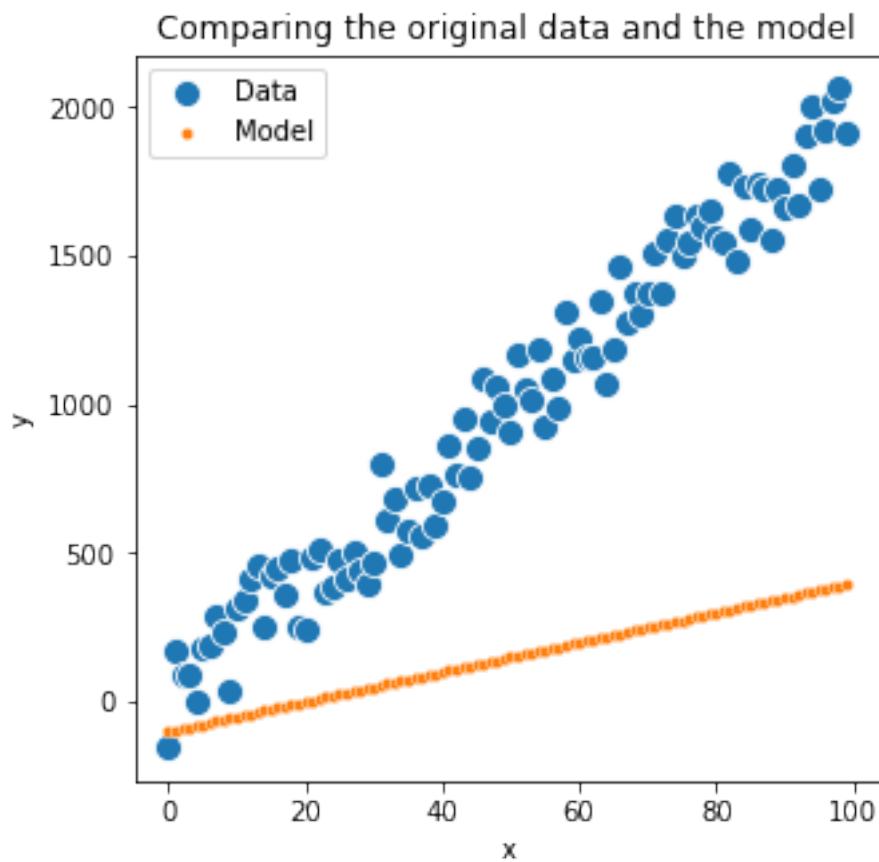


Figure 3: Comparing the original data and the model

Go back, modify the values for ‘A_model’ and ‘k_model’ and re-run the above cells to

re-plot your model chart. Repeat this until you have a ‘reasonable’ fit (i.e. the two lines match)

How well did we do in our ‘by inspection’ method?

```
1 print(f"A = {A:.2f}")  
2 print(f"A_model = {A_model:.2f}")  
3 print();  
4 print(f"K = {K:.2f}")  
5 print(f"K_model = {k_model:.2f}")
```

A = 19.55

A_model = 5.00

K = 11.02

K_model = -100.00

12.2.2 Method 2 : Measuring the gap (error) between the model and the data-points

Generate fresh Data

```
1 previous_error = float('inf') # We need this shortly  
2  
3 A = random.uniform(-20, 20)  
4 K = random.uniform(-100, 100)  
5  
6 # Following is arbitrary - but I wanted a variance  
7 # that scaled with the slope to make it more  
8 # visible on charts  
9 variance = int(A*10)  
10  
11 x, y = genData(numPoints = 100, A = A, K = K, variance = variance)
```

Manual ‘guess’

```
1 A_model = -8.2  
2 K_model = 40  
3 variance_model = 0.0001  
4  
5 x_model, y_model = genData(numPoints = 100,  
6     A = A_model,  
7     K = K_model,  
8     variance = variance_model)
```

Plot the source data and the model

```
1 fig = plt.figure(figsize=(5, 5))
2 fig = sns.scatterplot(x = x, y=y, s = 100)
3 fig = sns.scatterplot(x = x_model, y=y_model, s = 20)
4 fig.set(xlabel ="x", ylabel = "y",
5         title ='Comparing the original data and the model')
6 plt.legend(labels=['Data', 'Model'])
7 plt.show()
```

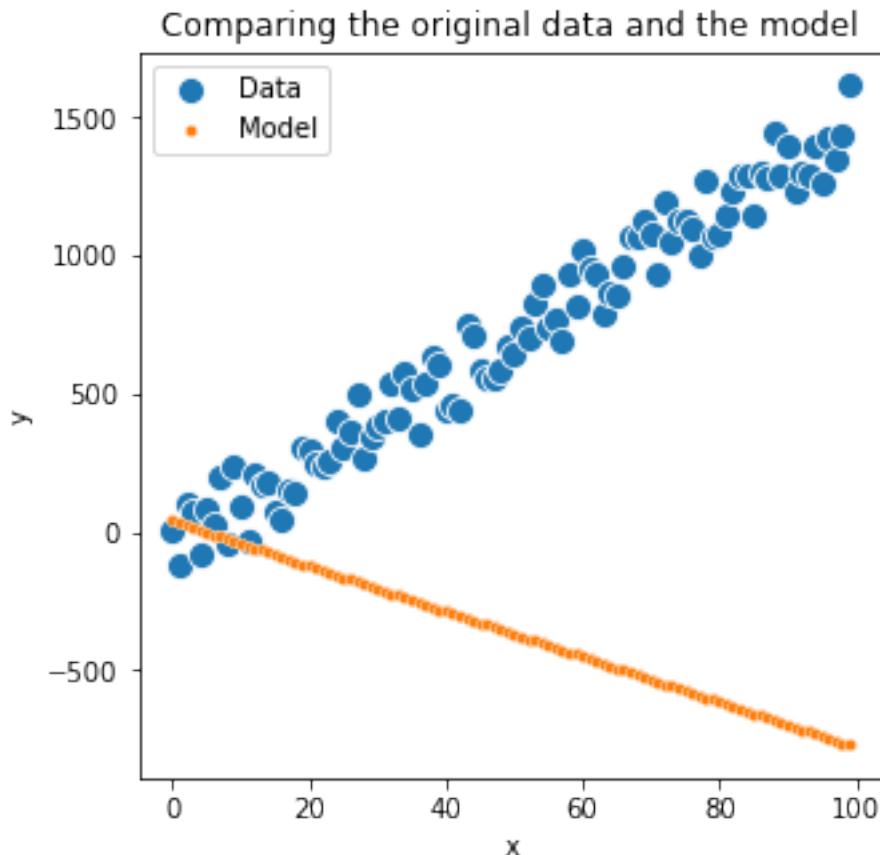


Figure 4: Comparing the original data and the model

Calculate the total gap (error) between the data and the mode

N.B. The following is, in fact, not commonly used .. for reasons that will become obvious later in the course.

```
1 total_error = 0
2 for i in range(len(y)):
```

```

3     total_error += abs(y[i] - y_model[i])
4
5 print(f"Previous error = {previous_error:.2f}")
6 print(f"total_error      = {total_error:.2f}")
7 if (abs(total_error) < abs(previous_error)):
8     print ('Improved model!')
9 else:
10    print ('Worse model')
11
12 previous_error = total_error # remember this for next time

```

```

Previous error = inf
total_error      = 107462.71
Improved model!

```

N.B. A ‘Pythonesque’ way of doing this might be:

```

1 total_error = sum([abs(a - b) for a, b in zip(y, y_model)])
2 print('total_error = ', total_error)

```

```
total_error = 107462.70736995531
```

How well did we do?

```

1 print(f"A =          {A:.2f}")
2 print(f"A_model = {A_model:.2f}")
3 print();
4 print(f"K =          {K:.2f}")
5 print(f"K_model = {k_model:.2f}")

```

```

A =          15.45
A_model = -8.20

K =          -56.83
K_model = -100.00

```

Chapter 13 - Model Fitting

Student Activity - Naïve Approaches to Model Fitting

(c) Dr Rob Collins 2023

2023-03-05

Table of contents

13 Student Activity : Naïve Approaches to Model Fitting	13 - 3
13.1 Introduction	13 - 3
13.2 Generating data which we can use to experiment with fitting	13 - 3
13.2.1 Review graphically:	13 - 4
13.3 Develop your own 'DIY' curve-fitting algorithm	13 - 4
13.3.1 Method 1 : 'The Shot-gun'	13 - 6
13.3.2 Method 2 : 'The Brute'	13 - 6
13.3.3 Method 3 : 'Global - local'	13 - 8
13.4 Your DIY methods here!	13 - 10

List of Figures

1	Journeypersons Struggling to Complete 'Learnist' Examination (c. 1590) . . .	13 - 2
2	A quick Test of our charting function for linear Regression	13 - 5
3	Method 1 : 'The Shot-gun'	13 - 7
4	Method 2 : 'The Brute'	13 - 8
5	Method 3 : 'Global - local'	13 - 9

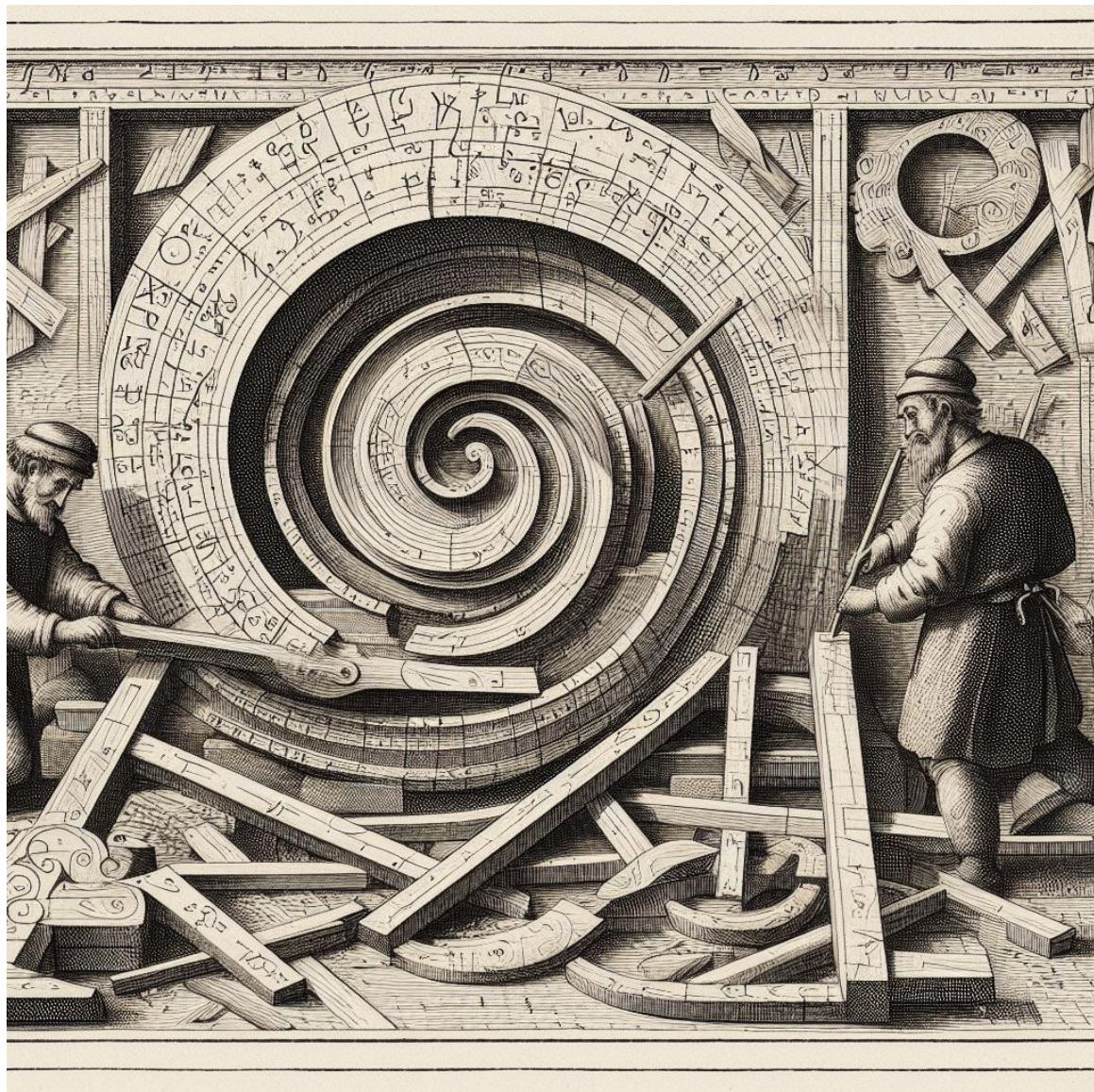


Figure 1: Journeypersons Struggling to Complete 'Learnist' Examination (c. 1590)

13 Student Activity : Naïve Approaches to Model Fitting

13.1 Introduction

This activity is intended to be a ‘fun’ warm-up activity rather than a serious attempt at algorithmics. The objectives for this section are to:

- Get you thinking about model fitting algorithms and how they work
- Enable you to work creatively in teams (if you do this as a team activity)
- Refresh your Python skills if you need to

13.2 Generating data which we can use to experiment with fitting

As previously import the ‘numpy’, ‘random’, ‘seaborn’ and ‘matplotlib.pyplot’ libraries

 Tutor provided code

```
import numpy as np
import random
import seaborn as sns
import matplotlib.pyplot as plt
```

Copy the ‘genData’ function from the previous demonstration to enable generation of data for this activity

 Tutor provided code

```
def genData(numPoints, A, k, variance):
    x = np.zeros(shape=numPoints)
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i] = i
        y[i] = (A * i + k) + random.uniform(-1, 1) * variance
    return x, y
```

And use this to generate some random data for use with your modelling algorithms (again, a copy from the previous demonstration)

 Tutor provided code

```
A_low = -20
A_high = 20

k_low = -100
k_high = 100
```

```

A = random.uniform(A_low, A_high)
k = random.uniform(k_low, k_high)

# Following is arbitrary - but I wanted a variance
# that scaled with the slope to make it more
# visible on charts
variance = int(A*10)

x, y = genData(numPoints = 100, A = A, k = k, variance = variance)

```

13.2.1 Review graphically:

Again, copy the plotting function from the previous demonstration so that you can use this to plot your data visually.

To improve on the previous demonstration, edit this in to a re-usable function with the signature:

```
{ def plotXY( x, y, x_model, y_model): }
```

 Tutor provided code

```

def plotXY( x, y, x_model, y_model):
    %matplotlib inline
    fig = plt.figure(figsize=(5, 5))
    fig = sns.scatterplot(x = x, y=y, s = 100, legend = True)
    fig = sns.scatterplot(x = x_model, y=y_model, s = 20, legend = True)
    fig.set(xlabel = "x", ylabel = "y",
            title ='Comparing the original data and the model')
    plt.legend(labels=['Data', 'Model'])
    plt.show()

```

Perform a quick test of your plotting code by calling it like this:

```
plotXY( x, y, x, y)
```

 Tutor provide code

```
plotXY( x, y, x, y)
```

13.3 Develop your own 'DIY' curve-fitting algorithm

This section is more of a fun, warm-up than a serious challenge. The idea is to get you thinking about how you might build a Machine Learning model. Your algorithms do not



Figure 2: A quick Test of our charting function for linear Regression

have to be fast or sophisticated or even that good!

The objective here is to generate as many different naive methods for fitting data as you can in the time available.

Note : Better to have 2-3 bad methods ‘working’ than one sophisticated method that does not.

The sections below are the ones I invented over about a 20 minute period. You can try these .. or otherwise (much better!) .. Invent your own!

13.3.1 Method 1 : ‘The Shot-gun’

Iterate whilst generating random parameters - remember the best solution!

💡 Your code here

```
total_error = 451907639.8389
total_error = 136297359.5334
total_error = 1753909.5508
total_error = 1171506.3687
total_error = 1138147.0347
total_error = 1117210.2809
total_error = 1113338.3643
A =      17.17
A_best = 16.96

K =      69.41
K_best = 79.32
```

Then plot your best fit as a chart

💡 Your code here

13.3.2 Method 2 : ‘The Brute’

Sweep through all possible values of model parameters at some given level of resolution

💡 Your code here



Figure 3: Method 1 : 'The Shot-gun

```
A = 17.17
```

```
A_best = 17.00
```

```
k = 69.41
```

```
k_best = 67.00
```

Then plot the chart again

💡 Your code here

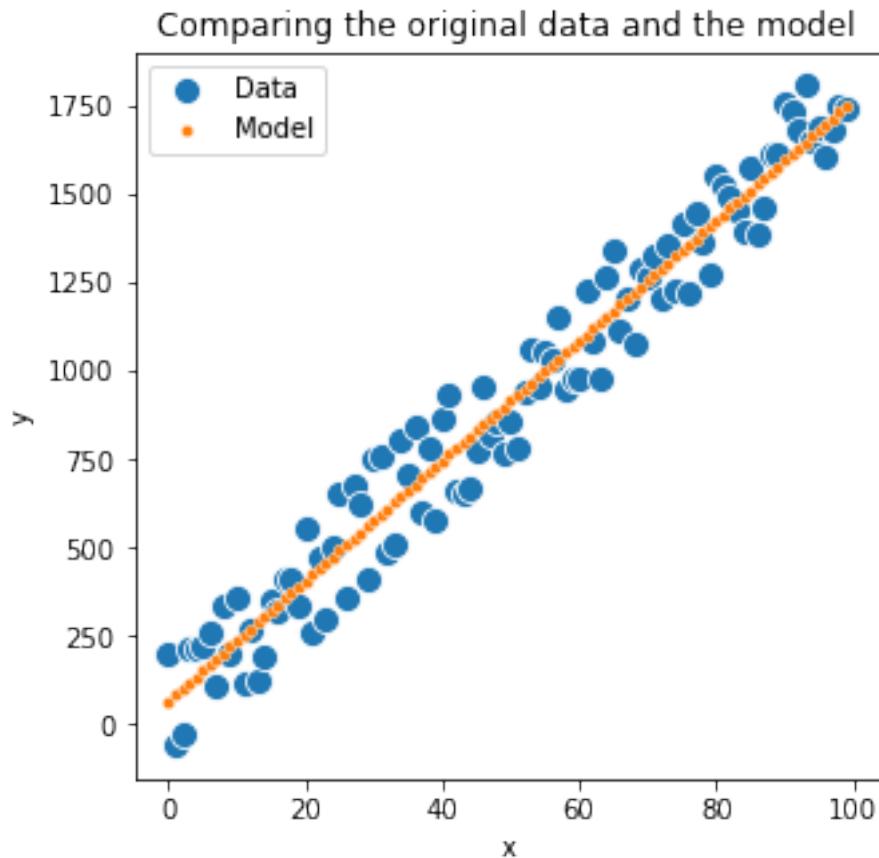


Figure 4: Method 2 : ‘The Brute’

13.3.3 Method 3 : ‘Global - local’

A refinement of ‘the brute’. the algorithm that sweeps quickly across the parameter space in large strides. Selects the best ‘stride’ and then refines the search within that smaller area.

💡 Your code here

```
A =      17.17  
A_best = 16.69
```

```
K =      69.41  
K_best = 88.77
```

And as always, plot the chart:

💡 Your code here

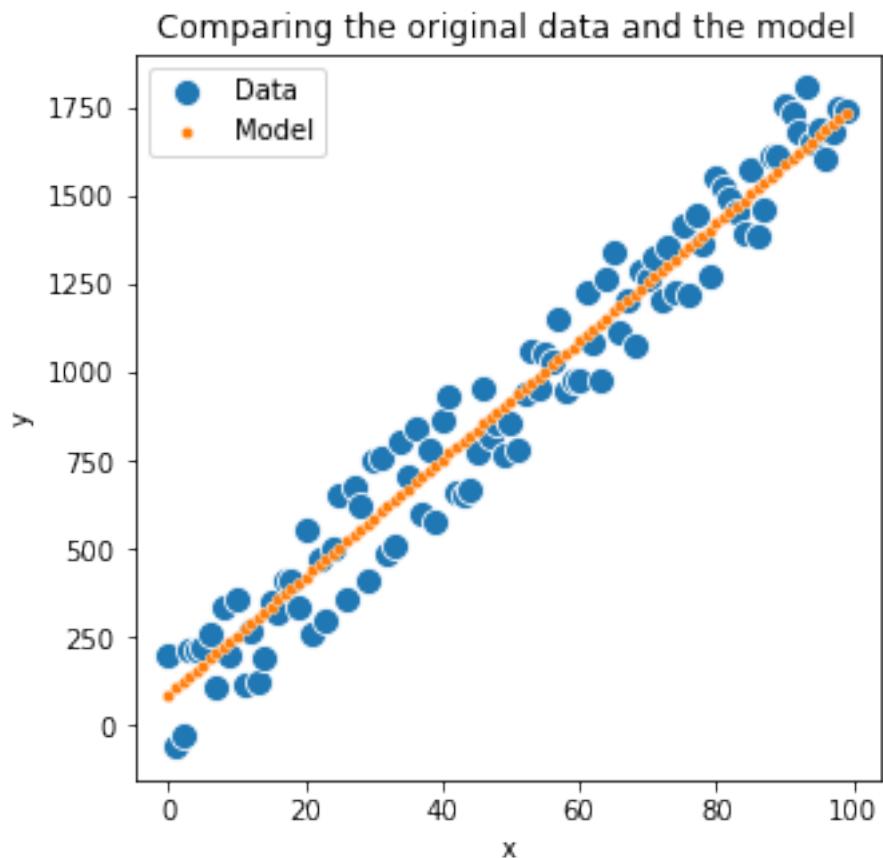


Figure 5: Method 3 : ‘Global - local’

13.4 Your DIY methods here!

Rather than using my naive algorithms above - why not invent your own?

Chapter 14 - Tutor Demonstrations 2

Non-parametric models - Step Function

(c) Dr Rob Collins 2023

2023-03-05

Table of contents

14 Non-parametric models : Step Function	14 - 2
14.1 Libraries and first data-set	14 - 2
14.2 Algorithm : Step function	14 - 4
14.3 Application of the algorithm to the data	14 - 6
14.4 Second dataset	14 - 7
14.5 Third dataset	14 - 8

List of Figures

1 Construction of the Great Pyramids during the fourth dynasty of the old kingdom (around 2580–2560 BCE) was completed without the aid of modern parametric models. Illustrated here using steps to approximate the desired geometry	14 - 2
2 Complicated data stream measured from an industrial machine	14 - 4
3 Non-parametric model fitted to machine data	14 - 7
4 A 2nd data-set fitted with the non-parametric model	14 - 8
5 A 3rd data-set fitted with the non-parametric model	14 - 9

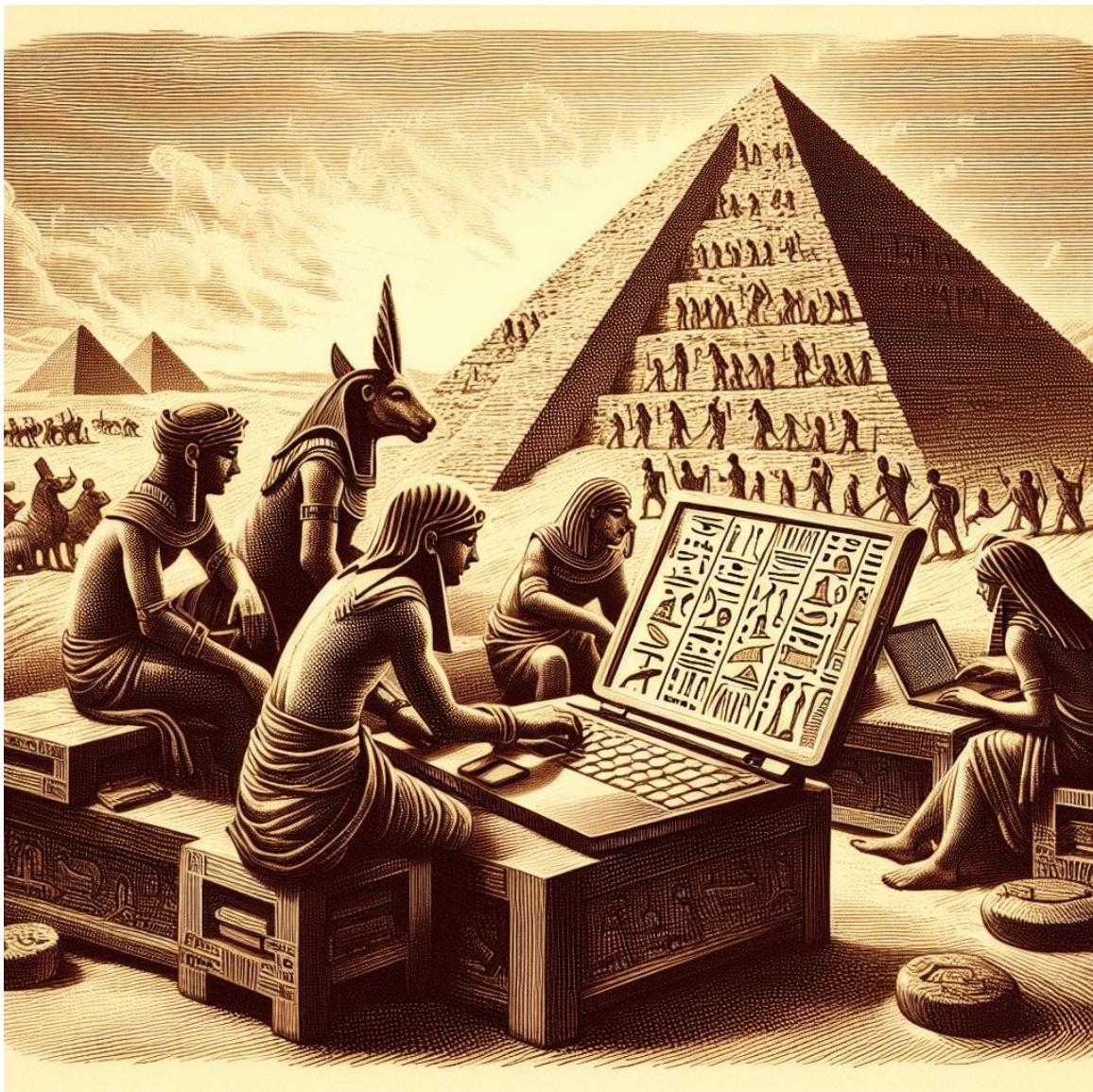


Figure 1: Construction of the Great Pyramids during the fourth dynasty of the old kingdom (around 2580–2560 BCE) was completed without the aid of modern parametric models. Illustrated here using steps to approximate the desired geometry

14 Non-parametric models : Step Function

14.1 Libraries and first data-set

```
1 import pandas as pd  
2 import matplotlib.pyplot as plt
```

```
3 import numpy as np  
  
1 filename = '03d - non_param_1.csv'  
2 df = pd.read_csv(filename)  
3 df
```

	Unnamed: 0	x	y
0	0	0.000000	-0.019858
1	1	0.050505	-0.046841
2	2	0.101010	0.071795
3	3	0.151515	0.140241
4	4	0.202020	-0.456774
...
95	95	4.797980	4.930462
96	96	4.848485	5.668955
97	97	4.898990	5.422268
98	98	4.949495	5.345203
99	99	5.000000	6.237989

```
1 plt.figure(figsize=(8, 6))  
2 plt.plot(df['x'], df['y'], label='data', color='blue')  
3 plt.xlabel('X')  
4 plt.ylabel('Y')  
5 plt.legend()  
6 plt.grid(True)  
7 plt.show()
```

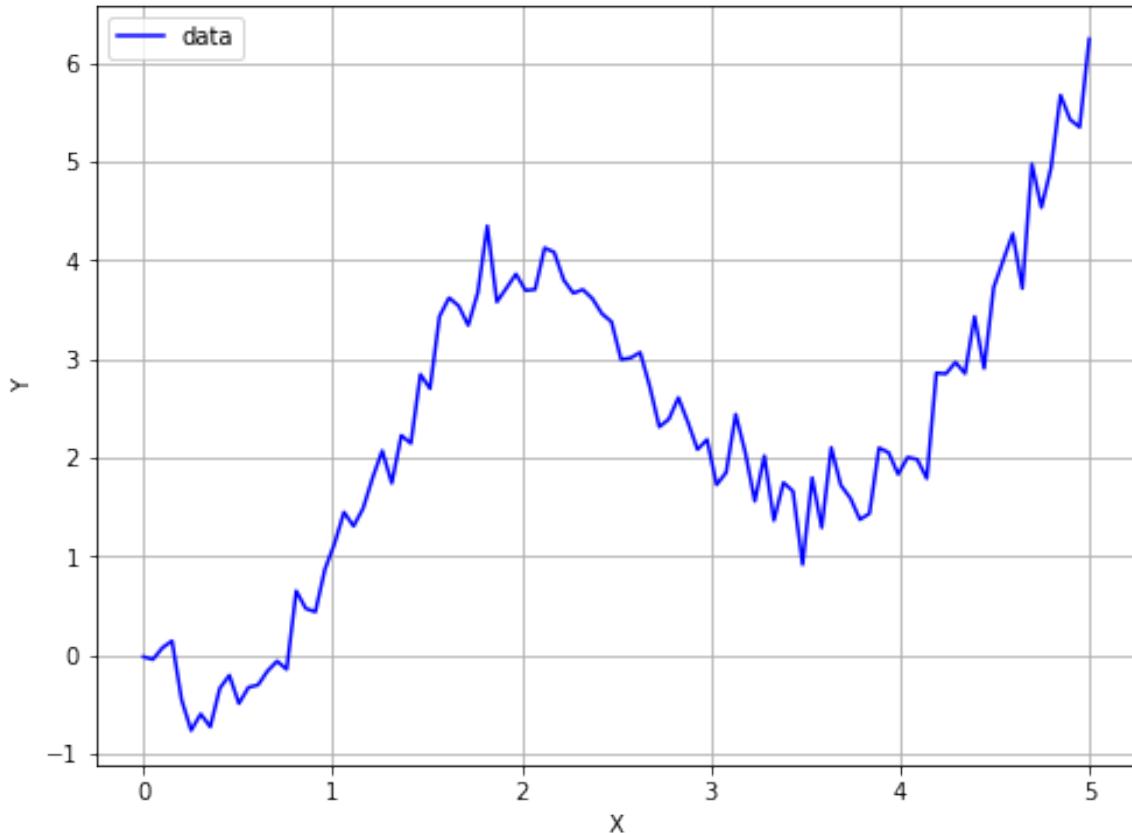


Figure 2: Complicated data stream measured from an industrial machine

14.2 Algorithm : Step function

```

1  number_of_bins = 4

2  class StepModel:
3      def __init__(self, num_bins = 10):
4          self.num_bins = num_bins
5          self.x_thresholds = []           # Lower value of items in the bin
6          self.y_values = []             # Corresponding y value
7          self.x_min = 0
8          self.x_max = 0
9
10     def fit(self,x,y):
11         x_len = len(x)
12         y_len = len(y)
13
14         # A little bit of sanity checking

```

```

14     if(x_len < self.num_bins ):    # more bins than data .. error out
15         raise ValueError("Error : More bins than data-points")
16
17     # need to have equal amount of data in both x and y
18     if(x_len != y_len ):
19         raise ValueError("Error : x and y data has different lengths")
20
21     # Ensure that x values are sorted in increasing order
22     # (and that y values match!)
23     zipped_lists = list(zip(x, y))
24
25     # Sort the zipped list based on the values in x
26     sorted_zipped_lists = sorted(zipped_lists, key=lambda x: x[0])
27     sorted_x, sorted_y = zip(*sorted_zipped_lists)
28
29     # Create lists representing the bins (now sorted)
30     x_bins = np.array_split(sorted_x, self.num_bins)
31     y_bins = np.array_split(sorted_y, self.num_bins)
32
33     # Populate the model
34     for bin_x, bin_y in zip(x_bins, y_bins):
35         # The lower threshold for this bin
36         self.x_thresholds.append(min(bin_x))
37         self.y_values.append(sum(bin_y) / len(bin_y))
38
39     # Remember the maximum and minimum x values over which
40     # the model was trained
41     self.x_min = x[0]
42     self.x_max = bin_x[-1]
43
44
45     def get_results(self,x):
46         # print('-----')
47         # print('x = ', x)
48         result = self.y_values[-1]
49         if (x < self.x_min):
50             # Any x less than the model training range
51             # is assumed to be at the low value
52             result = self.y_values[0]
53         elif (x > self.x_max):
54             # Any x great than the model training range
55             # is assumed to be at the high value
56             result = self.y_values[-1]
57         else:
58             last_y = self.y_values[0]

```

```

59     for threshold,y in zip(self.x_thresholds, self.y_values):
60         if (x < threshold):
61             result = last_y
62             break
63         last_y = y
64
65     return (result)

```

14.3 Application of the algorithm to the data

Create model instance

```

1 step_model_obj = StepModel(20)

```

Fit the model to the data

```

1 step_model_obj.fit(df['x'].to_list(), df['y'].to_list())
1 df['step'] = df['x'].apply(step_model_obj.get_results)

```

```

1 plt.figure(figsize=(8, 6))
2 plt.plot(df['x'], df['y'], label='data', color='blue')
3 plt.plot(df['x'], df['step'], label='model', color='red')
4 plt.xlabel('X')
5 plt.ylabel('Y')
6 plt.legend()
7 plt.grid(True)
8 plt.show()

```

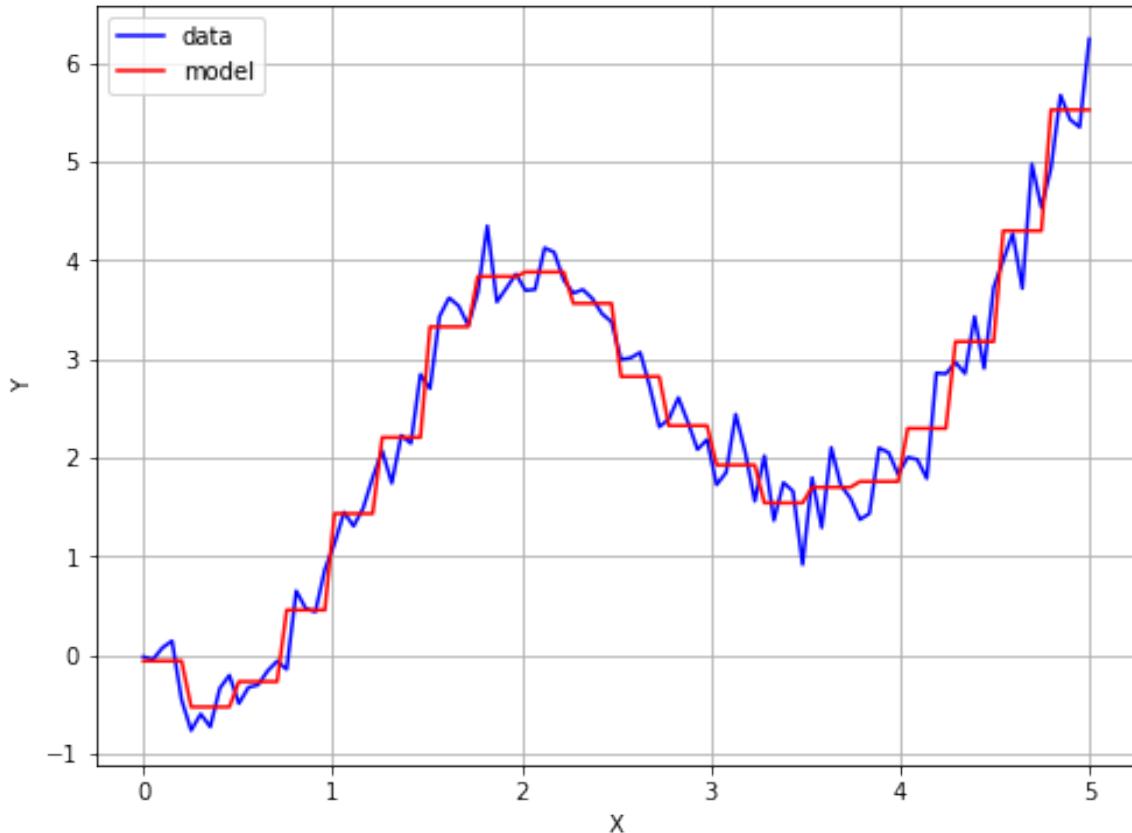


Figure 3: Non-parametric model fitted to machine data

14.4 Second dataset

```

1 filename = '03d - non_param_2.csv'
2 df = pd.read_csv(filename)

1 step_model_obj = StepModel(25)
2 step_model_obj.fit(df['x'].to_list(), df['y'].to_list())
3 df['step'] = df['x'].apply(step_model_obj.get_results)

1 plt.figure(figsize=(8, 6))
2 plt.plot(df['x'], df['y'], label='data', color='blue')
3 plt.plot(df['x'], df['step'], label='model', color='red')
4 plt.xlabel('X')
5 plt.ylabel('Y')
6 plt.legend()
7 plt.grid(True)
8 plt.show()

```

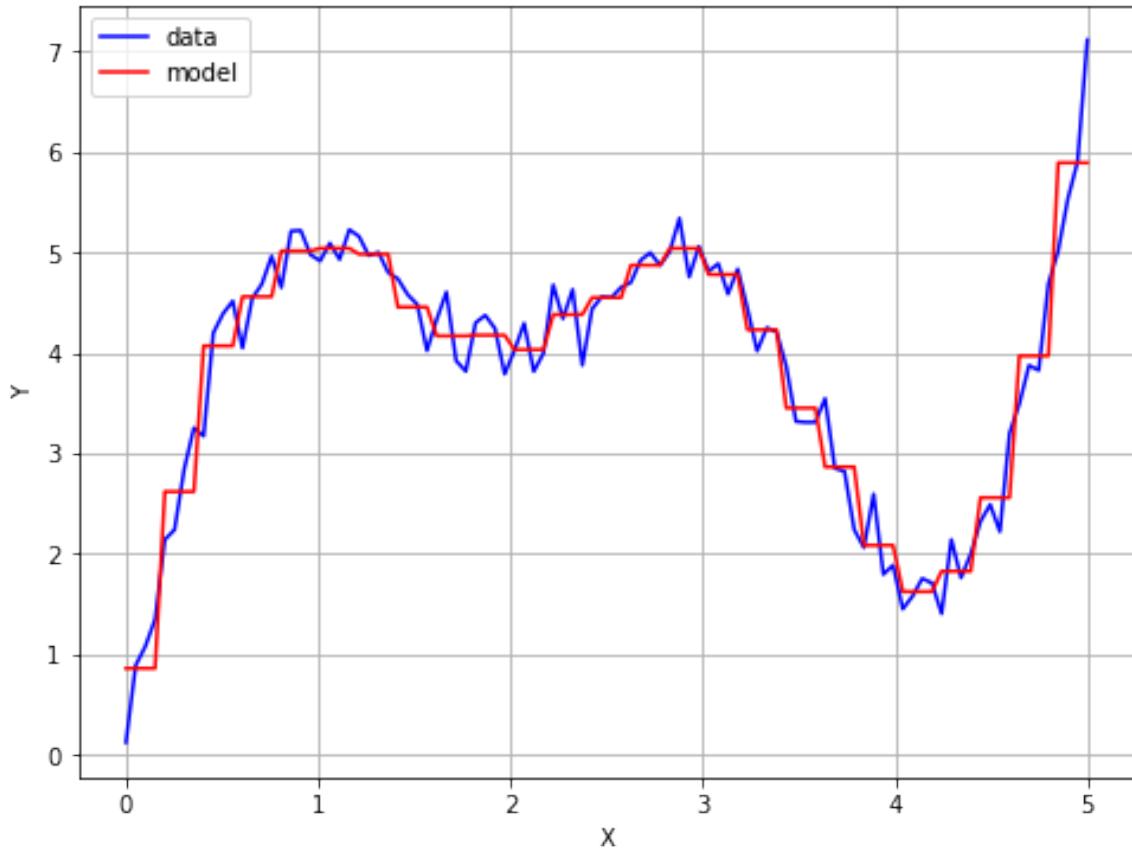


Figure 4: A 2nd data-set fitted with the non-parametric model

14.5 Third dataset

```

1 filename = '03d - non_param_3.csv'
2 df = pd.read_csv(filename)

1 step_model_obj = StepModel(30)
2 step_model_obj.fit(df['x'].to_list(), df['y'].to_list())
3 df['step'] = df['x'].apply(step_model_obj.get_results)

1 plt.figure(figsize=(8, 6))
2 plt.plot(df['x'], df['y'], label='data', color='blue')
3 plt.plot(df['x'], df['step'], label='model', color='red')
4 plt.xlabel('X')
5 plt.ylabel('Y')
6 plt.legend()
7 plt.grid(True)
8 plt.show()

```

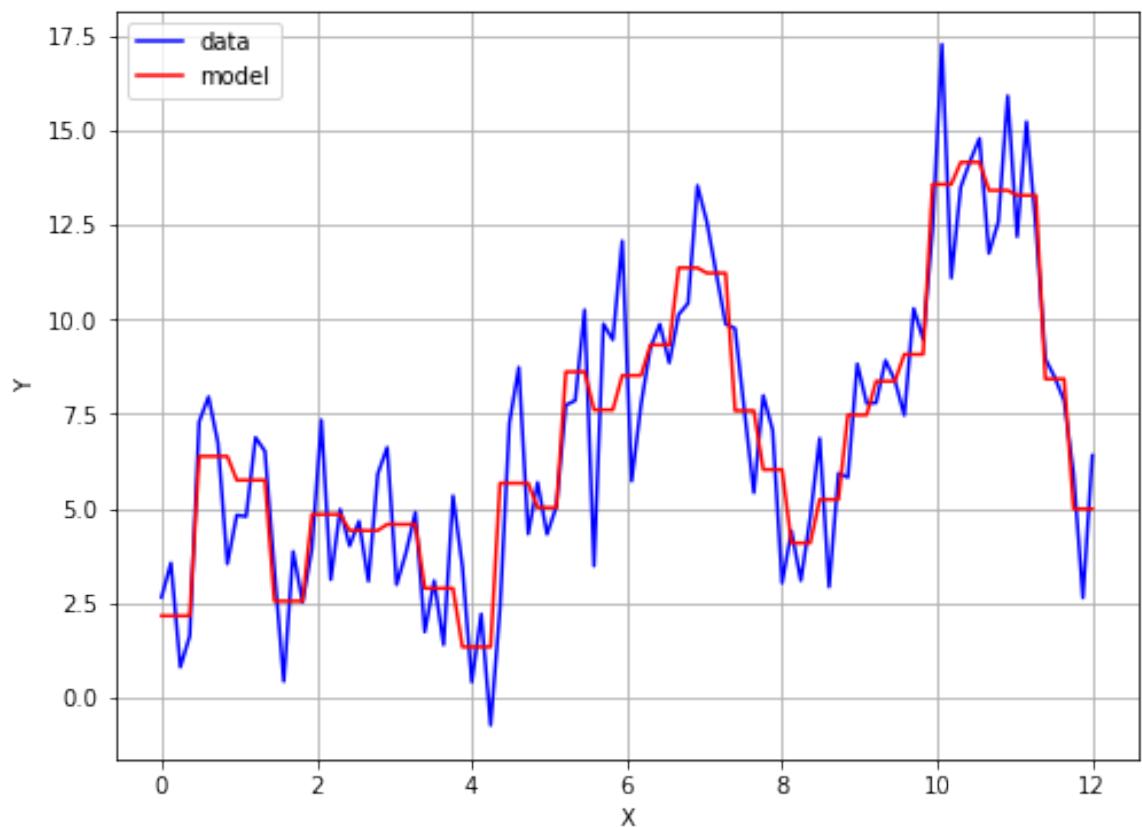


Figure 5: A 3rd data-set fitted with the non-parametric model

Chapter 15 - Tutor Demonstrations 3

A brief introduction to Dimensionality Reduction

(c) Dr Rob Collins 2023

2023-12-06

Table of contents

15 Tutor Demonstration - A brief introduction to Dimensionality Reduction	15 - 3
15.1 Importing libraries	15 - 3
15.2 Loading data	15 - 3
15.3 Summarising data	15 - 3
15.4 Plotting data	15 - 4
15.5 Applying Principle Component Analysis	15 - 5
15.6 Plotting the Result	15 - 6

List of Figures

1 14th Century craftsman struggling to reduce dimensions of 3-dimensiona wooden blocks	15 - 2
2 Scatter matrix of Features from turtle data-set	15 - 5
3 Explained Variance Ratio and Cumulative Percentage by Principal Compo nent for turtle data-set	15 - 7

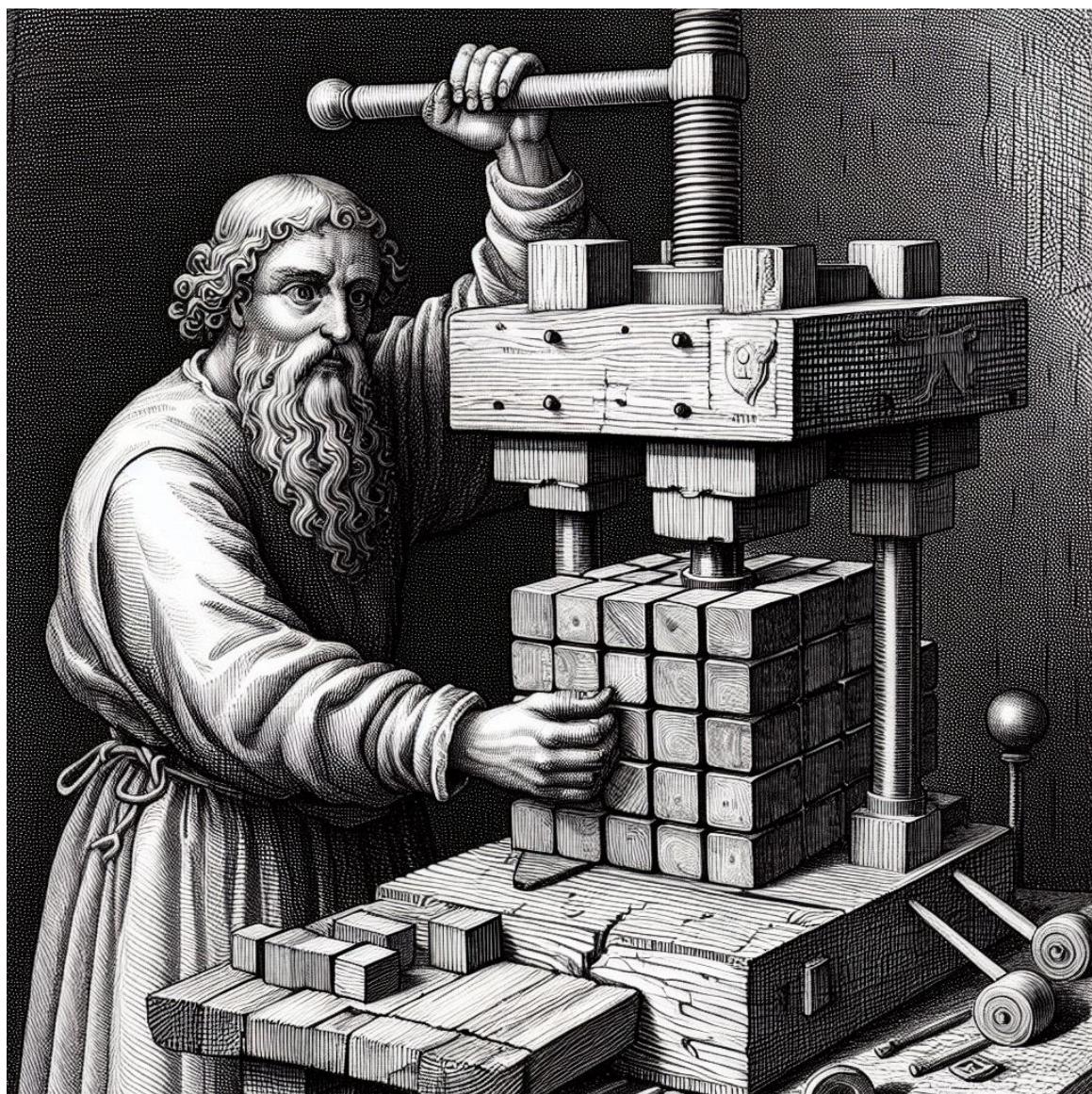


Figure 1: 14th Century craftsman struggling to reduce dimensions of 3-dimensiona wooden blocks

15 Tutor Demonstration - A brief introduction to Dimensionality Reduction

15.1 Importing libraries

```
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from pandas.plotting import scatter_matrix
```

15.2 Loading data

```
1 pd.set_option('display.float_format', '{:.2f}'.format)
2 df = pd.read_csv('turtle research data.csv')
3 df
```

	Weight	Length	Height	Width	Shell_damage_index	Beak_length	Body_fat
0	10.47	33.30	41.63	32.32	31.36	31.39	36.74
1	8.74	27.62	36.18	29.26	29.22	28.99	33.80
2	9.14	26.89	37.80	27.09	27.78	27.73	33.89
3	8.47	26.38	35.62	28.20	27.55	27.48	32.02
4	9.87	30.37	39.41	28.15	30.89	30.02	35.42
...
995	9.16	29.39	37.77	28.62	29.83	27.73	32.79
996	10.29	29.64	39.71	28.79	30.75	29.35	35.02
997	9.08	27.41	38.46	29.15	26.83	28.44	33.11
998	11.09	31.02	44.04	31.21	32.33	30.78	37.36
999	9.39	28.12	37.39	28.57	28.02	28.40	32.32

15.3 Summarising data

```
1 df.describe()
```

	Weight	Length	Height	Width	Shell_damage_index	Beak_length	Body_fat
count	1000.00	1000.00	1000.00	1000.00	1000.00	1000.00	1000.00
mean	9.98	29.89	39.92	29.96	29.96	29.94	34.97
std	1.17	2.56	3.67	2.10	2.10	2.12	2.61

	Weight	Length	Height	Width	Shell_damage_index	Beak_length	Body_fat
min	6.18	22.29	26.89	24.08	24.05	23.33	26.28
25%	9.16	28.12	37.45	28.53	28.47	28.57	33.04
50%	10.00	29.92	39.94	30.06	29.98	29.92	34.93
75%	10.75	31.67	42.30	31.38	31.41	31.41	36.73
max	13.84	38.53	52.49	38.83	37.27	36.52	45.40

15.4 Plotting data

```
1 scatter_matrix(df, figsize=(12, 12));
2 plt.show()
```

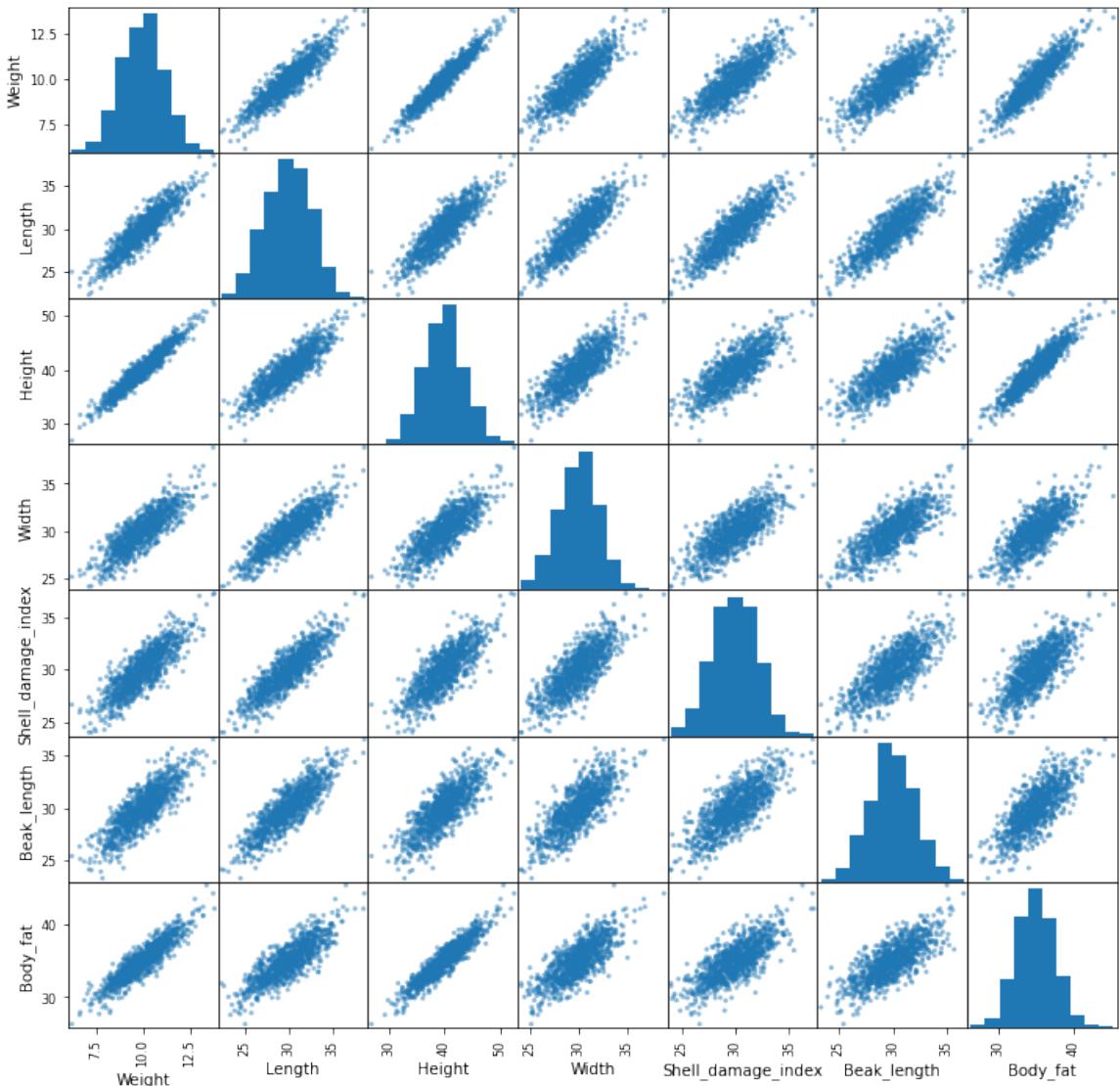


Figure 2: Scatter matrix of Features from turtle data-set

Oh dear - all of our data seems to be highly correlated. Could we use PCA to reduce this down to some key dimensions?

15.5 Applying Principle Component Analysis

```

1 from sklearn.decomposition import PCA
2 from sklearn.preprocessing import StandardScaler
3 import matplotlib.pyplot as plt

```

```
1  scaler = StandardScaler()  
2  df_scaled = scaler.fit_transform(df)
```

```
1  pca = PCA()
```

```
1  principal_components = pca.fit_transform(df_scaled)
```

15.6 Plotting the Result

```
1  explained_variance_ratio = pca.explained_variance_ratio_  
  
2  
3  
4  cumulative_variance_ratio = explained_variance_ratio_.cumsum()  
5  
6  
7  fig, ax1 = plt.subplots()  
8  ax1.bar(range(1, len(explained_variance_ratio) + 1),  
9          explained_variance_ratio,  
10         alpha=0.7,  
11         label='Explained Variance Ratio')  
12 ax1.set_xlabel('Principal Component')  
13 ax1.set_ylabel('Explained Variance Ratio', color='tab:blue')  
14  
15 ax2 = ax1.twinx()  
16 ax2.plot(range(1, len(cumulative_variance_ratio) + 1),  
17           cumulative_variance_ratio,  
18           color='tab:red',  
19           marker='o',  
20           label='Cumulative %')  
21 ax2.set_ylabel('Cumulative %', color='tab:red')  
22  
23 plt.title("Explained Variance Ratio and Cumulative Percentage by Principal Component")  
24 plt.legend(loc='upper left')  
25  
26 plt.show()
```

Explained Variance Ratio and Cumulative Percentage by Principal Component

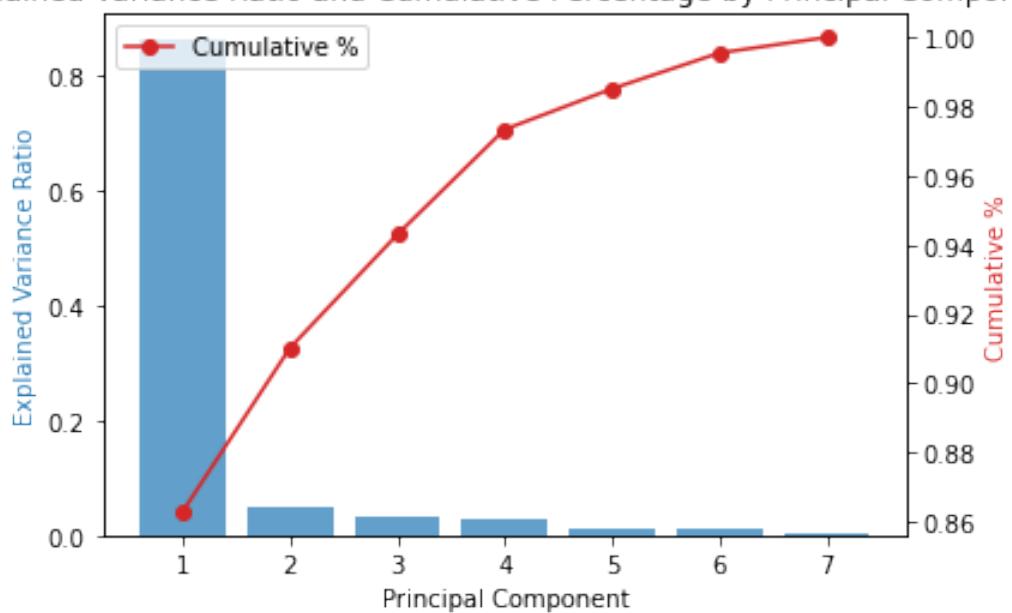


Figure 3: Explained Variance Ratio and Cumulative Percentage by Principal Component for turtle data-set

ISBN 978-173850580-7



A standard linear barcode representing the ISBN number 978-173850580-7. The barcode is composed of vertical black bars of varying widths on a white background.

9 781738
505807