Chapter 1 - Feature Engineering

Solution Workbook for Student Practical Classes

Dr Rob Collins 2023

2024-07-12

Table of contents

List of Figures

Figure 1: Medieval stone mason calculating the exact proportion of features for a frieze 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:

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 scalers, arrays and multi-dimensional arrays.

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

The 'matplot' 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.

```
import pandas as pd
from pandas.plotting import scatter_matrix
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
import missingno as msno
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
```

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.

```
pd.set_option('display.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.

```
df = pd.read_csv('machine_params_and_cost_v2.csv')
```

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

```
print(f"df.shape = {df.shape}")
```

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

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

```
print("df.info() = ")
df.info()
```

```
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:

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

	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

```
1 df.loc[0:5,'T2':'B3']
```

	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

```
1 df.loc[0:5,'T4':'B4']
```

	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

```
df.loc[0:5,'Contamination_Defect':'Burnishing_Defect']
```

	Contamination_Defect	Crystalisation_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

```
df.loc[0:5,'m1_cost':]
```

	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 occurence 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

```
def count_ordinal(df, feature):
    # Count the number of occurrences for each ordinal value
    value_counts = df[feature].value_counts()

    # Create a bar chart
    value_counts.plot(kind='bar')

    # Add labels and a title
```

```
    plt.xlabel('Category')
    plt.ylabel('Count')
    plt.title('Count of Ordinal Values for feature ' + feature)

    # Show the plot
    plt.show()
```

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

```
count_ordinal(df, 'ag')
```

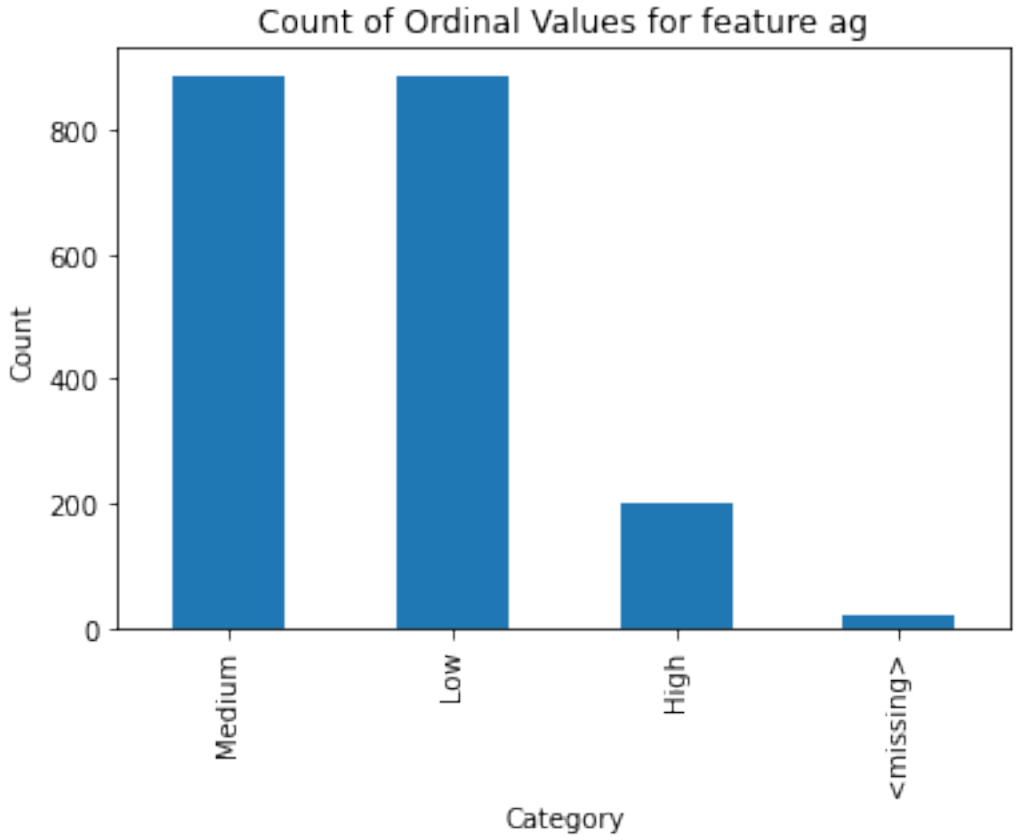


Figure 2: Bar-chart showing the number of items of each type within the ‘ag’ feature

```
count_ordinal(df, 'bg')
```

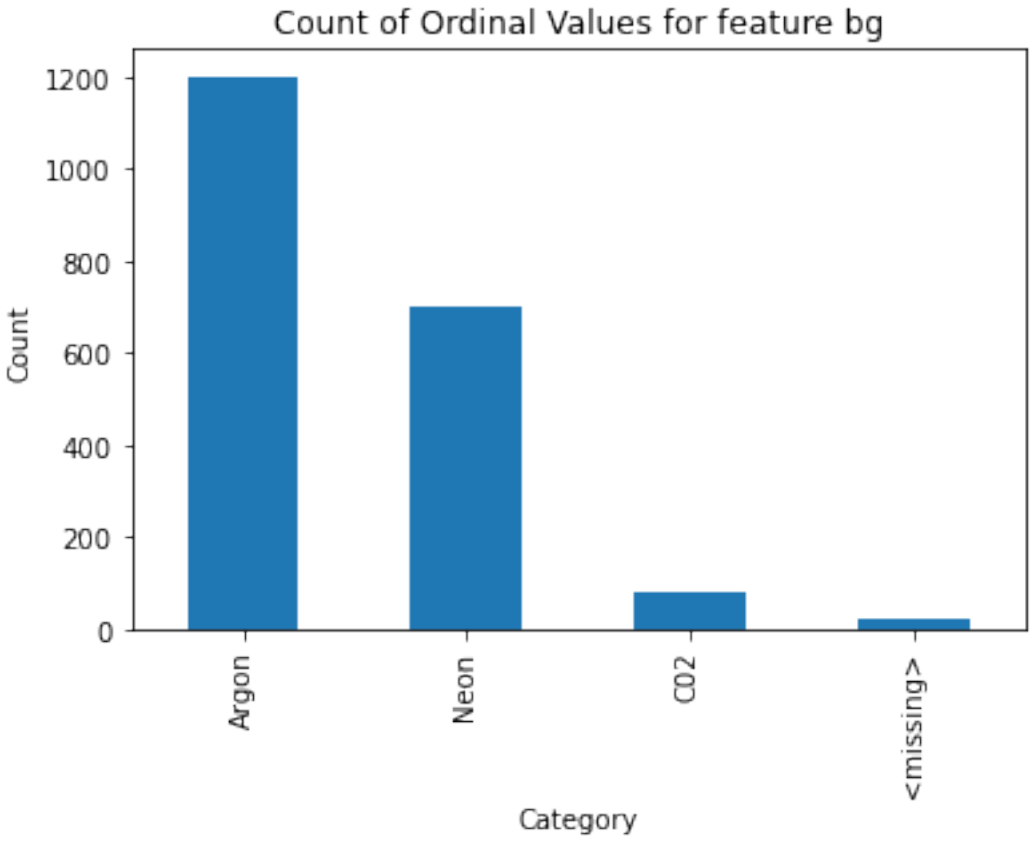


Figure 3: Bar-chart showing the number of items of each type within the 'bg' feature

1.9 Using seaborn 'countplot' to create a chart showing a count of each value in a 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.

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.

```
def count_ordinal_sns(df, feature):
    # Create a count plot
    sns.set(style="whitegrid")  # Optional, for grid lines
    plt.figure(figsize=(8, 4))  # Optional, for adjusting the figure size
    sns.countplot(data=df, y=feature)

    # Add labels and a title
    plt.xlabel('Category')
```

```
    plt.ylabel('Count')
    plt.title('Count of Ordinal Values within ' + feature)

    # Show the plot
    plt.show()
```

```
count_ordinal_sns(df, 'ag')
```

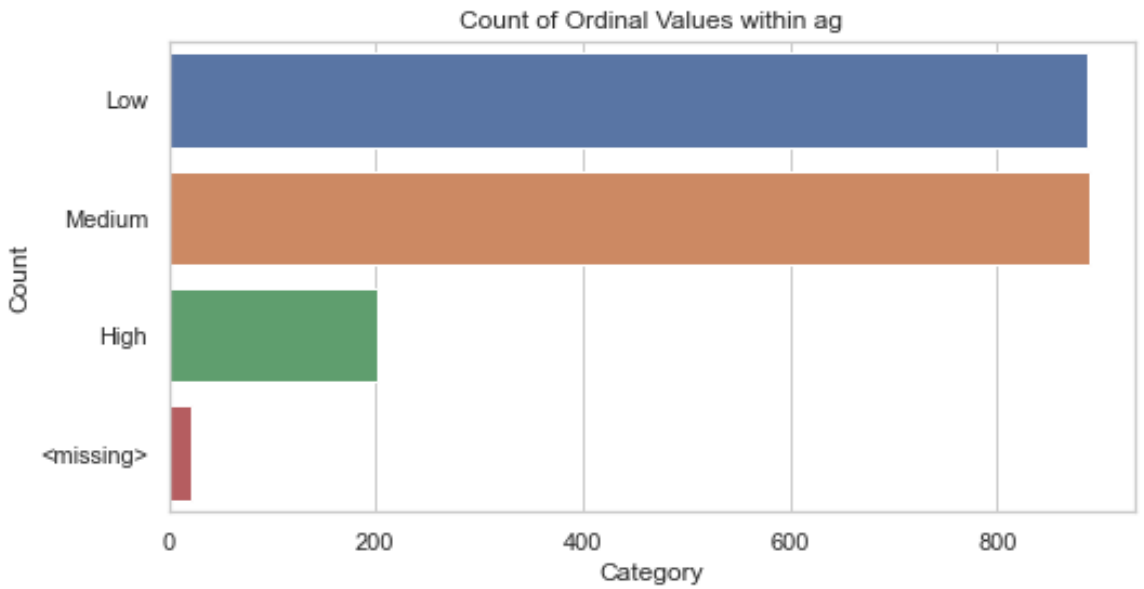


Figure 4: Seaborn version of Bar-chart for 'ag' feature

```
count_ordinal_sns(df, 'bg')
```

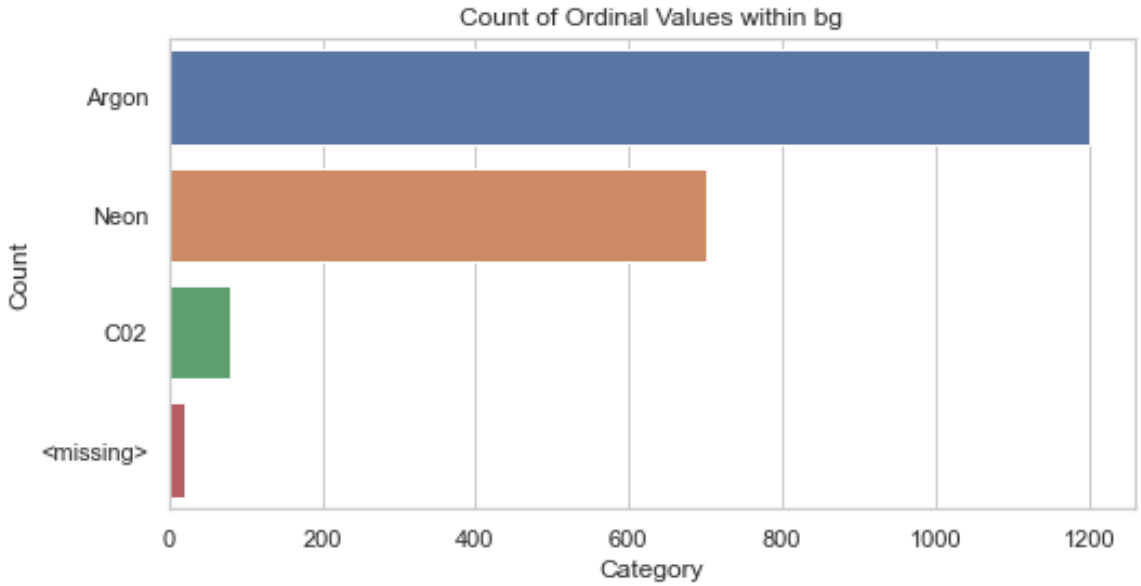


Figure 5: Seaborn version of Bar-chart for 'bg' feature

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 ever item labelled as 'missing' into a 'Nan' value. Write a line of code that converts each cell labelled `\<missing\>` into a 'Nan' value

```
df = df.replace('<missing>', np.nan)
```

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

```
count_ordinal_sns(df, 'ag')
```

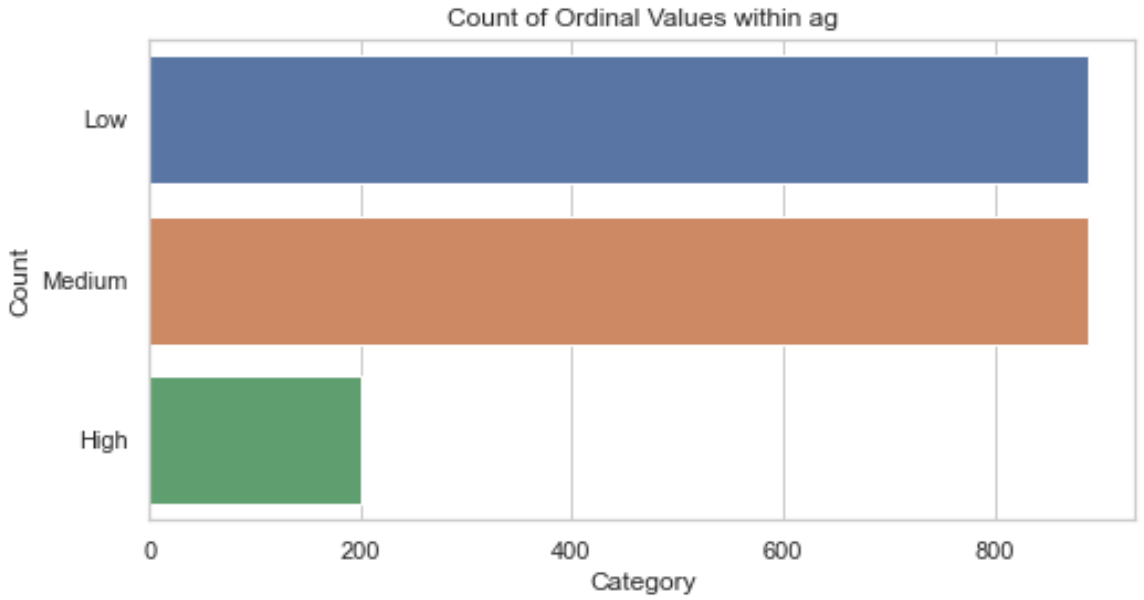


Figure 6: ‘ag’ feature after removal of values

```
1 count_ordinal_sns(df, 'bg')
```

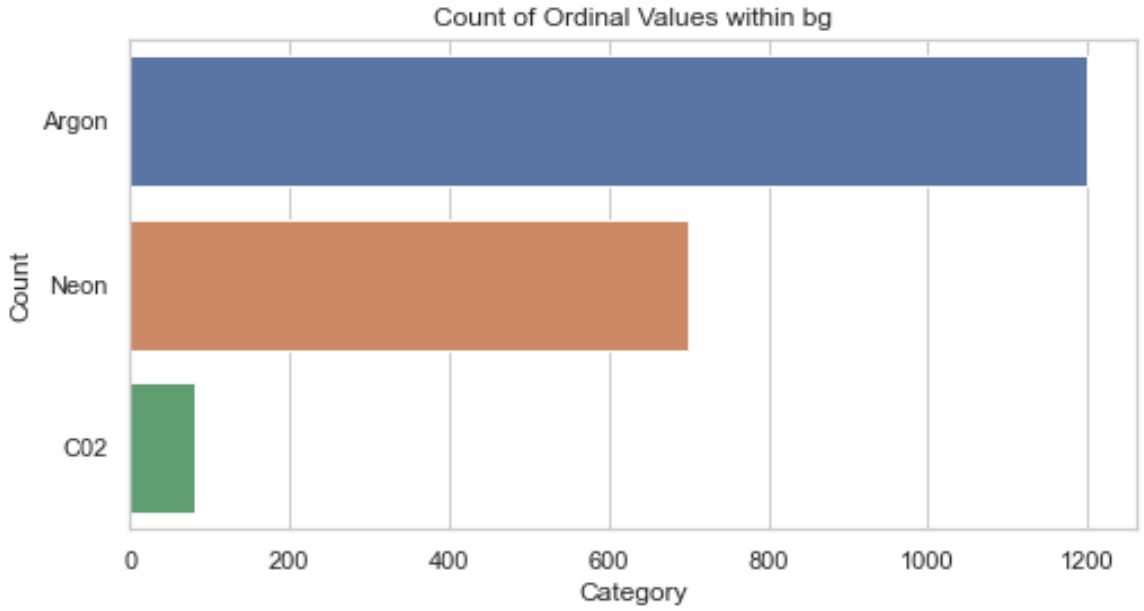


Figure 7: ‘bg’ feature after removal of values

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 od the types of each object and a count of the number of non-null items

```
print("df.info() = ")
df.info()
```

```
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.

```
df = df.drop(columns=['ID'])
```

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 colums at a time to improve layout on a printed page)

```
df.loc[:, 'T1':'B1'].describe()
```

	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

```
df.loc[:, 'T2':'A3'].describe()
```

	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

```
df.loc[:, 'P3':'B4'].describe()
```

	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

```
df.loc[:, 'Contamination_Defect':'Burnishing_Defect'].describe()
```

	Contamination_Defect	Crystalisation_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

```
df.loc[:, 'm1_cost':'m4_cost'].describe()
```

	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 visualy review any missing data.

```
msno.matrix(df)
plt.show()
```

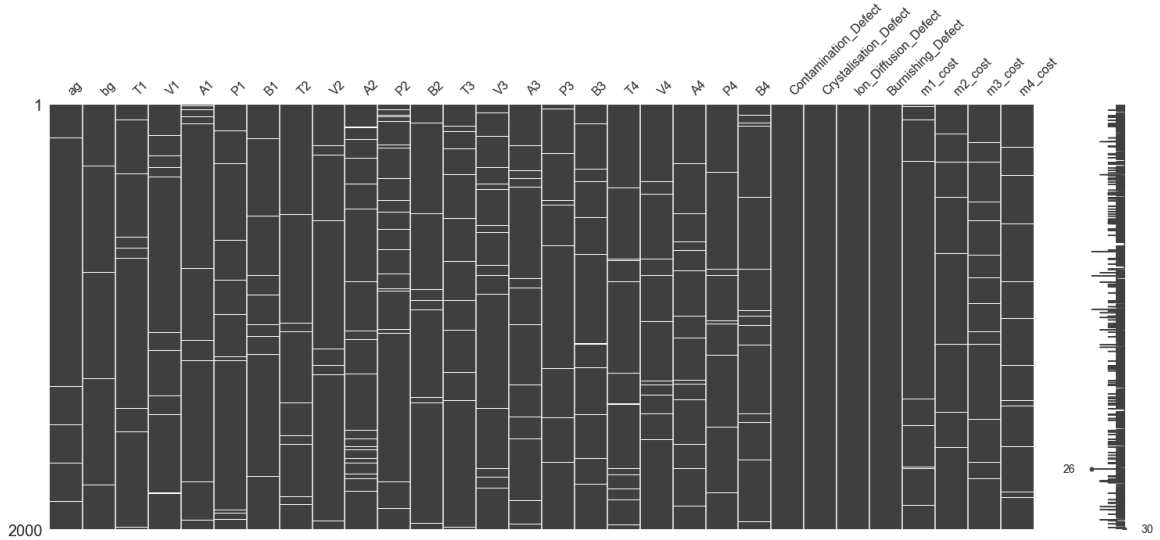


Figure 8: missingno 'matrix' function showing missing data in the df dataframe

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

```
msno.bar(df)
plt.show()
```

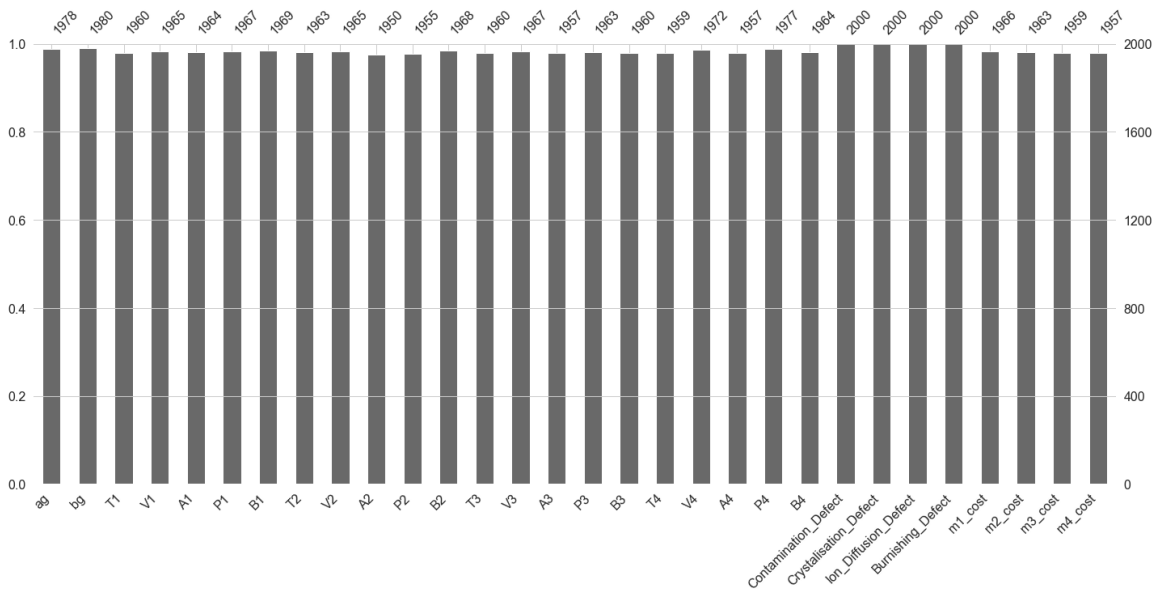


Figure 9: missingno 'bar' function showing missing data in the df dataframe

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.

```
msno.heatmap(df)
plt.show()
```

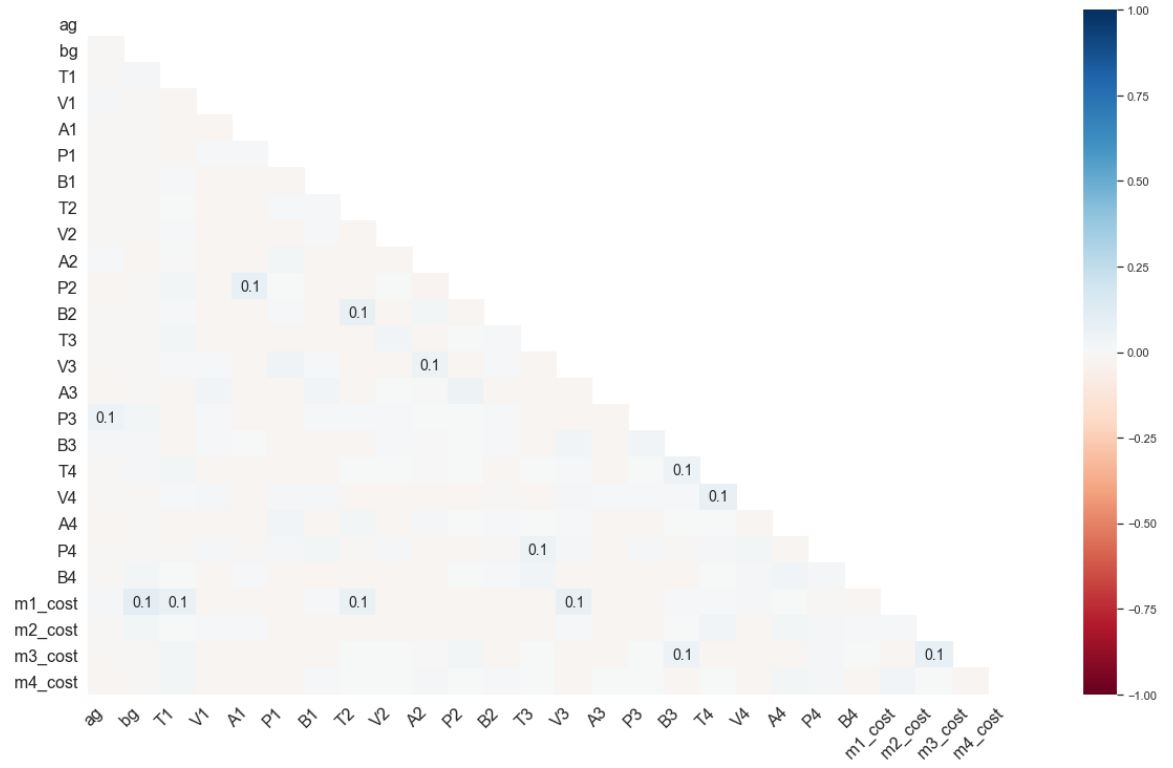


Figure 10: missingno 'heatmap' function showing missing data in the df dataframe

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

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.

```
df_no_nan = df.dropna()
msno.bar(df_no_nan)
plt.show()
```

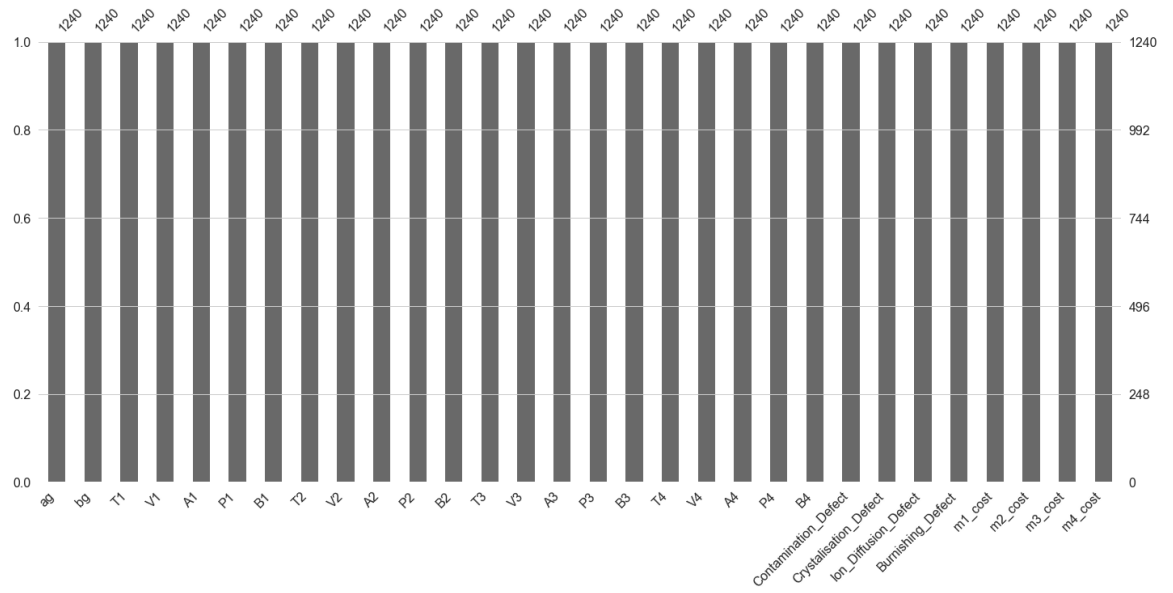


Figure 11: missingno 'bar' chart showing impact of dropping rows containing missing values

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'.

```
scatter_matrix(df, figsize=(10, 10));                    # Comment out during development
```

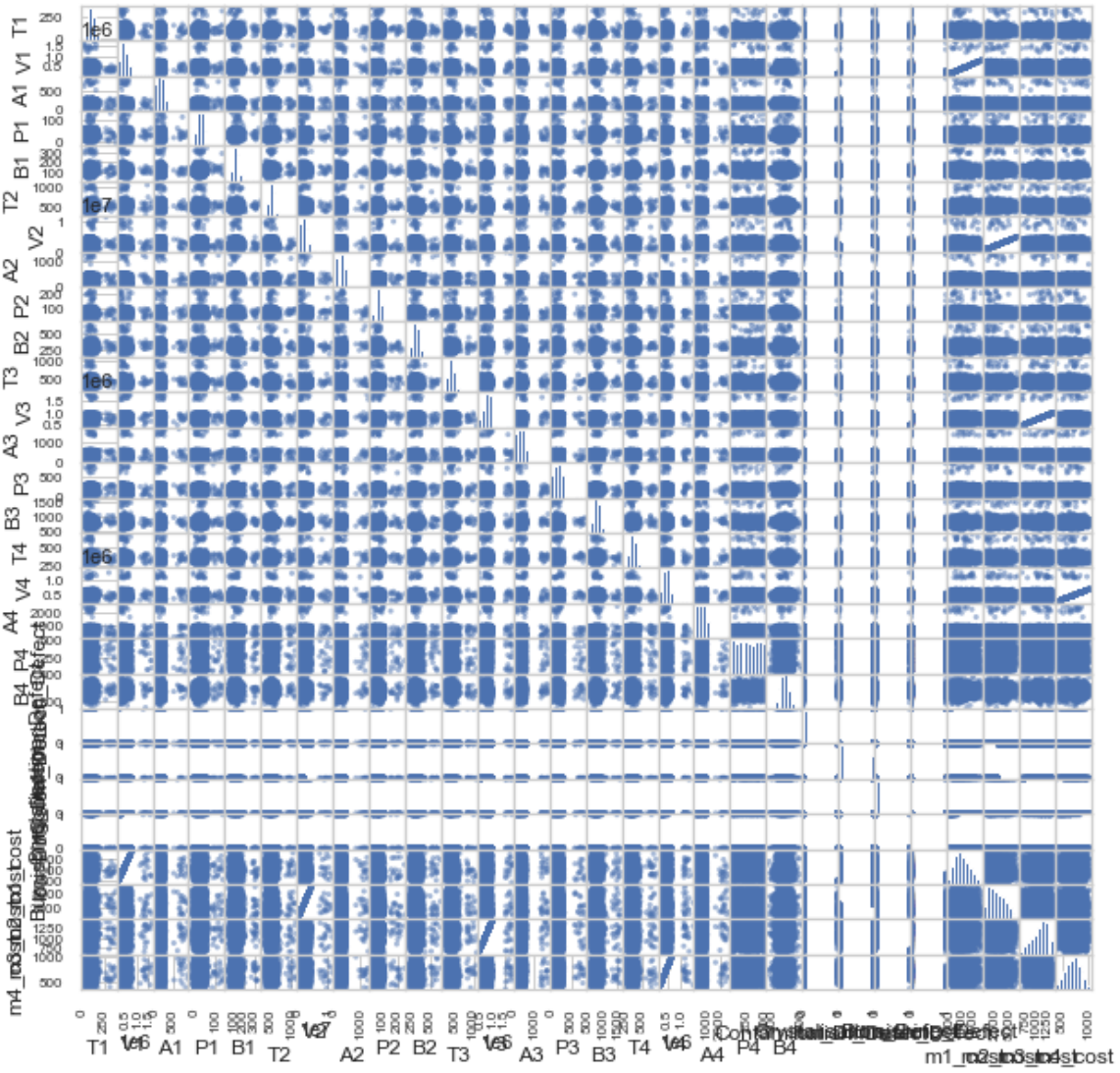


Figure 12: High-level overview of all data using the 'scatter_matrix' function

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

```
features = ['T1', 'B1', 'A1', 'V1']
scatter_matrix(df[features], figsize=(15, 15));
```

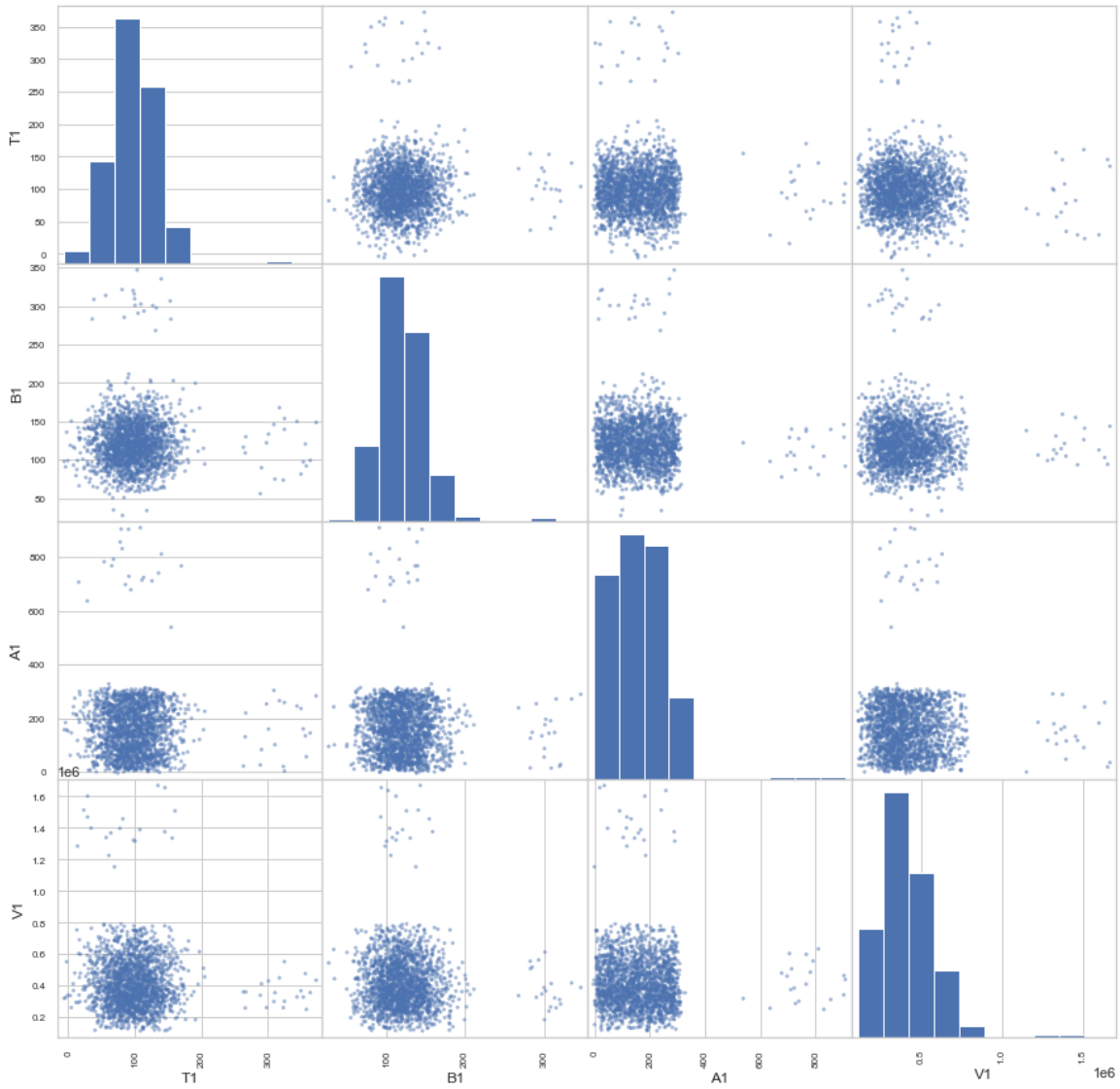


Figure 13: Visualisation of a sub-set of the dataframe using 'scatter_matrix' function

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

```
def histogram(df, feature, bins):
    # Create a histogram
    df[feature].plot.hist(bins=bins, edgecolor='k')

```

```
    # Add labels and a title
    plt.xlabel(feature)
    plt.ylabel('Frequency')
    plt.title(feature + ' Histogram')

    # Show the plot
    plt.show()
```

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

```
histogram(df,'V1',20)
```

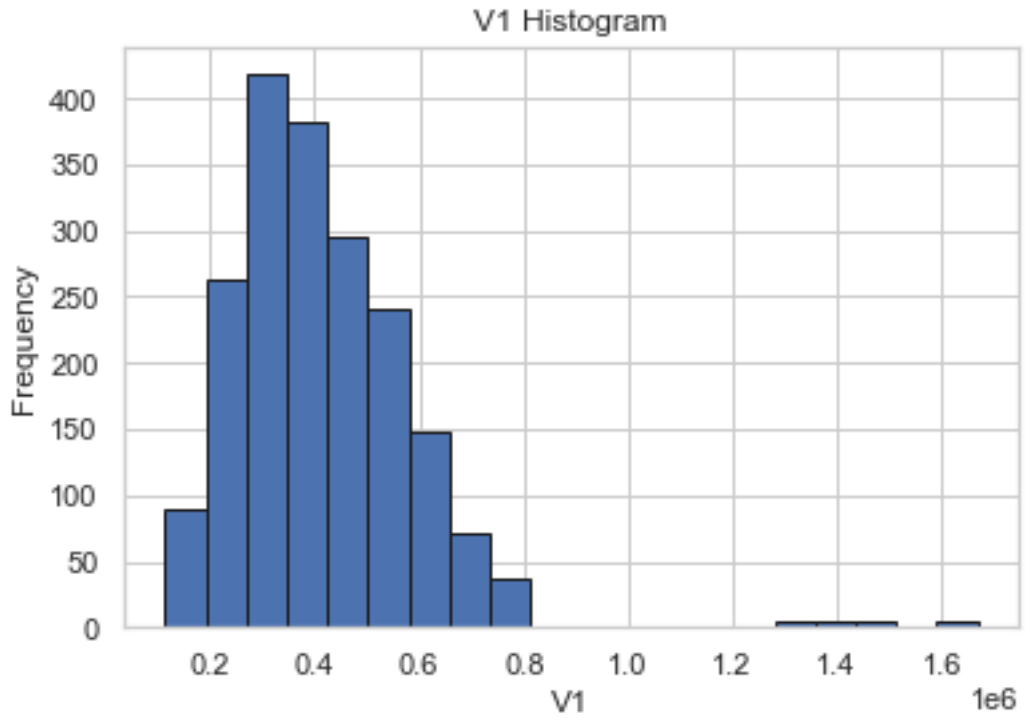


Figure 14: Histogram showing detailed view of data distribution for V1

```
histogram(df,'V2',20)
```

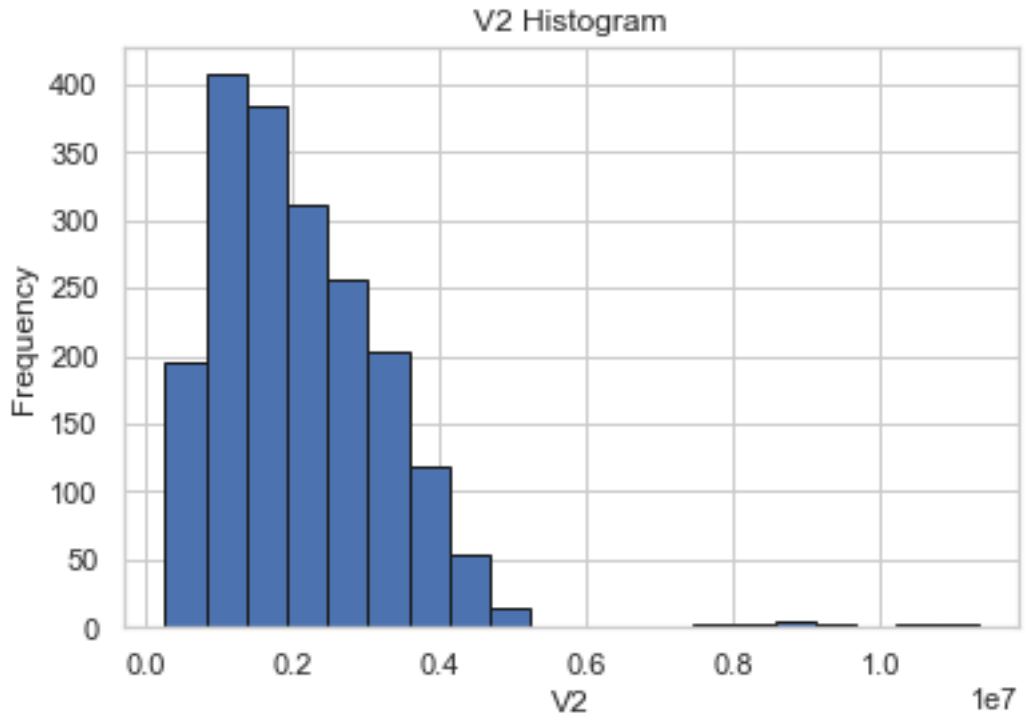


Figure 15: Histogram showing detailed view of data distribution for V2

```
1 histogram(df,'V3',20)
```

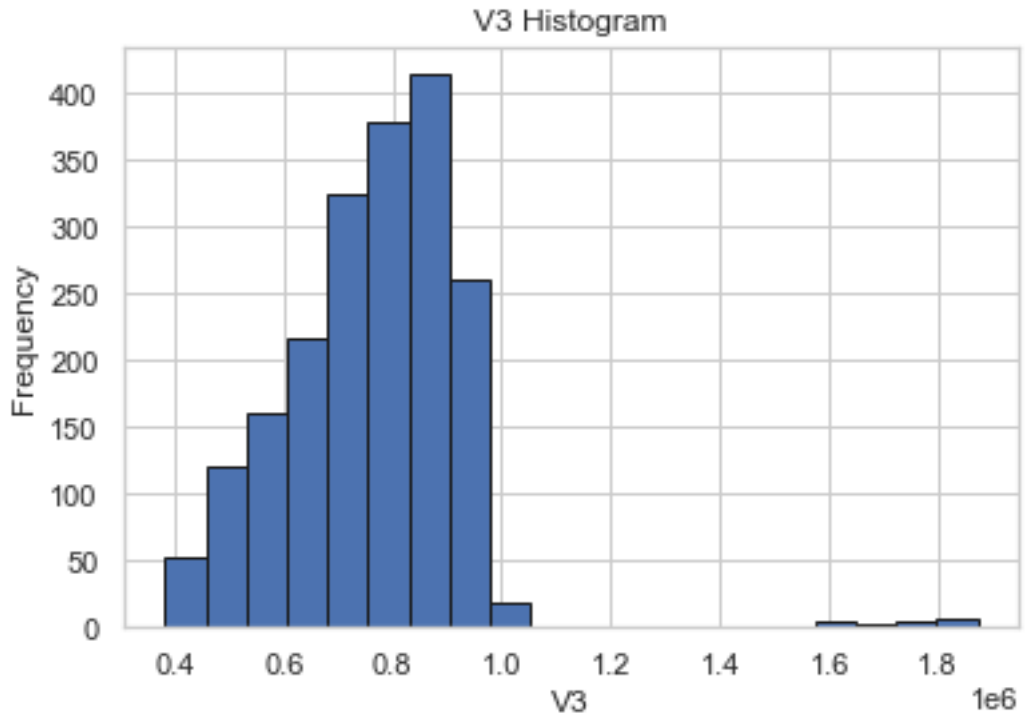


Figure 16: Histogram showing detailed view of data distribution for V3

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...

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):`

```

def plotBox(df, features):
    sns.set(style="darkgrid")
    plt.figure(figsize=(13,10))
```

```
    sns.boxplot(data=df[features])
    plt.title("Boxplot")
    plt.show()
```

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

```
plotBox(df, ['T1', 'T2', 'T3', 'T4'])
```

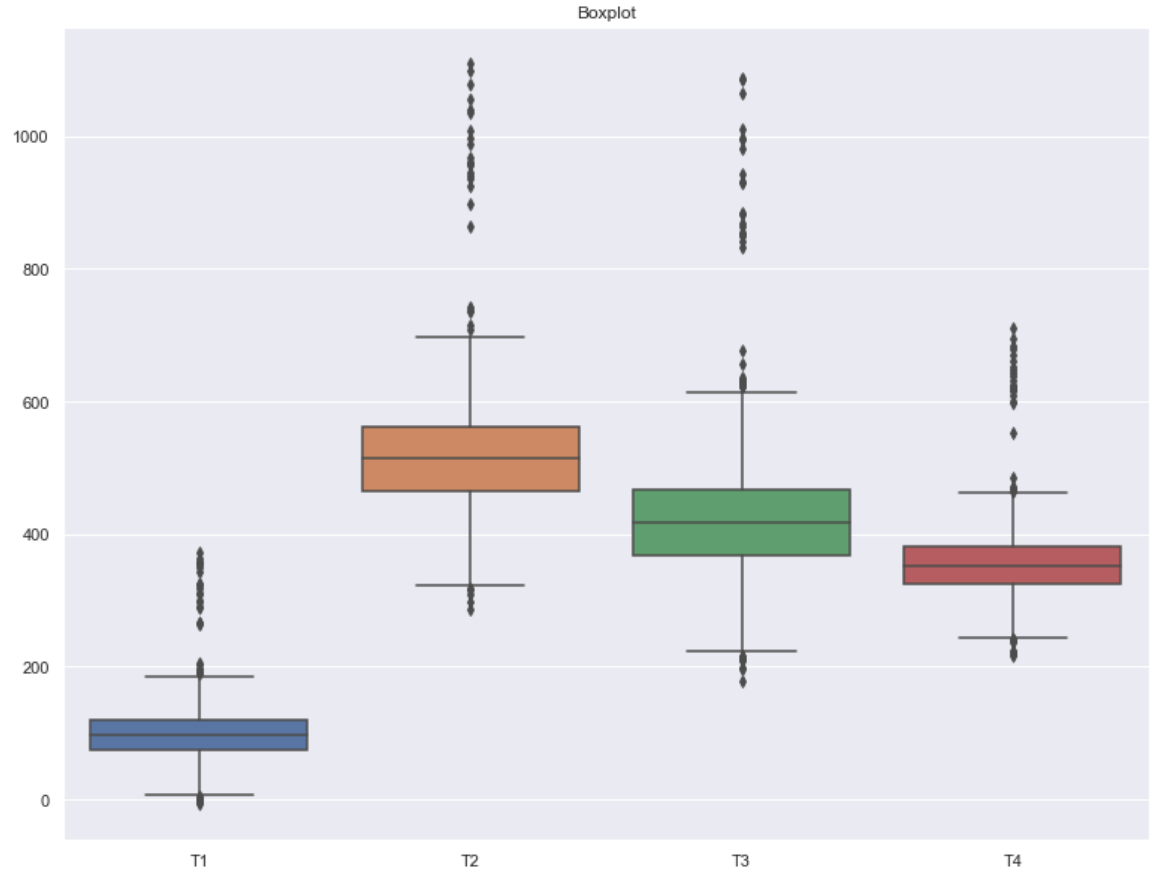


Figure 17: Box-plot showing relative distribution of features and outliers

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.

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

```
plotBox(df, ['V1', 'V2', 'V3', 'V4'])
plotBox(df, ['A1', 'A2', 'A3', 'A4'])
plotBox(df, ['P1', 'P2', 'P3', 'P4'])
```

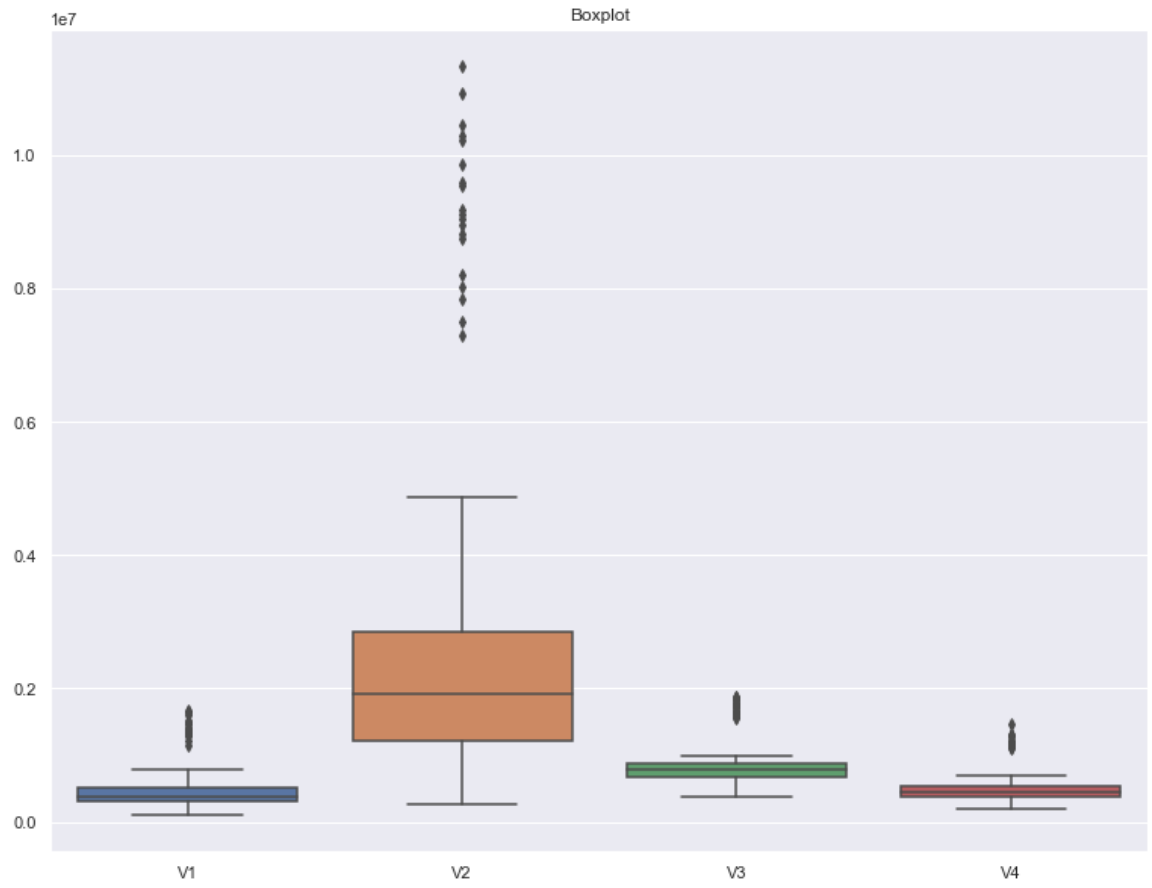
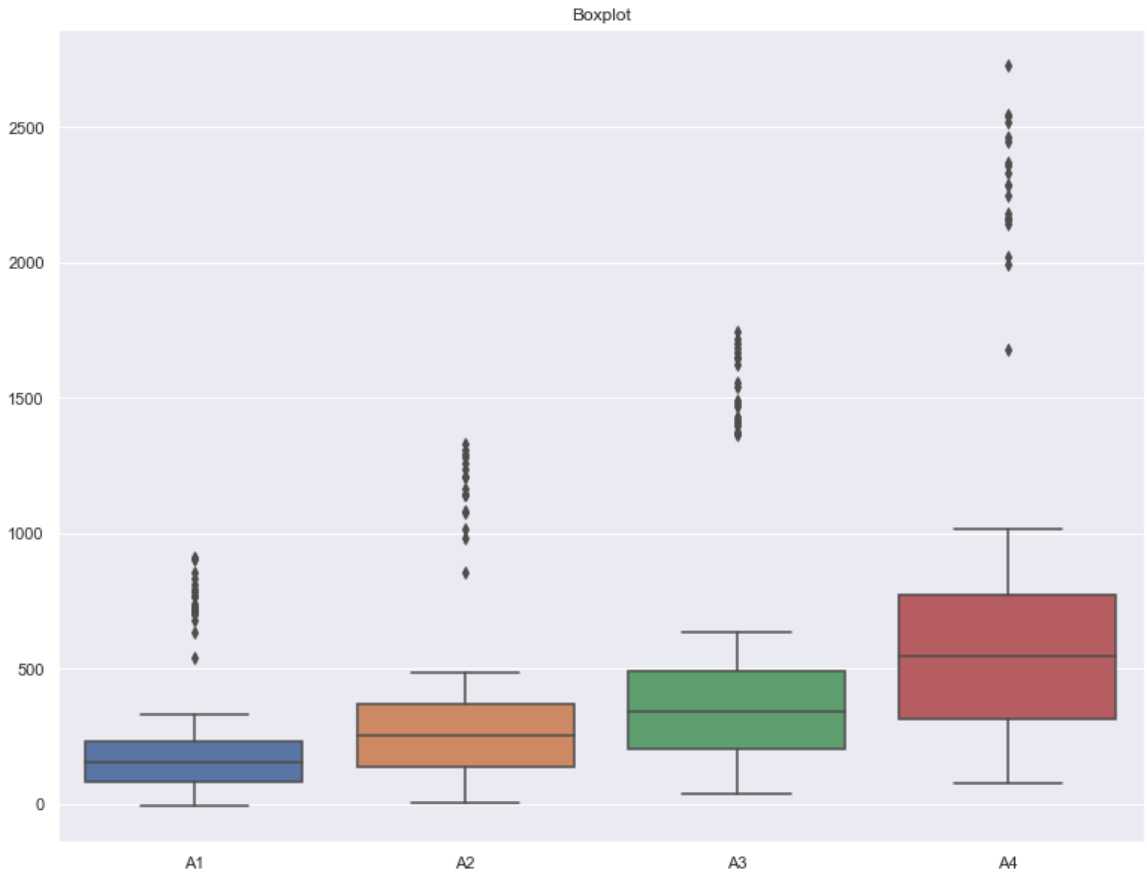
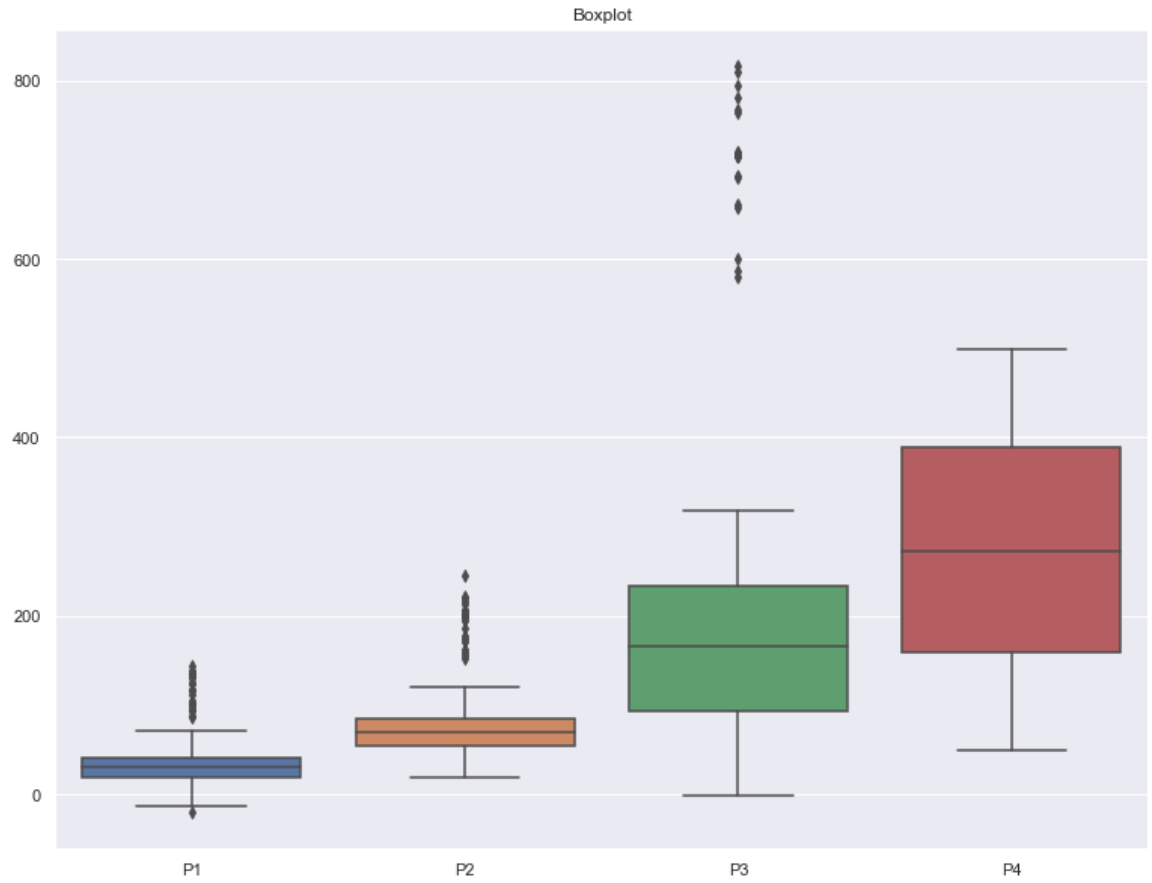


Figure 18: Box-plots showing relative distribution of features and outliers


Boxplot
2500
2000
1500
1000
500
0
A1
A2
A3
A4



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

```
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):
```

```
def plotViolin(df, features):
    sns.set(style="darkgrid")
    plt.figure(figsize=(10,10))
    sns.violinplot(data=df[features], orient = 'h')
    plt.title("Violin plot")
    plt.show()
```

Apply that function to each of the Temperature variables

```
plotViolin(df, ['T1', 'T2', 'T3', 'T4'])
plt.show()
```

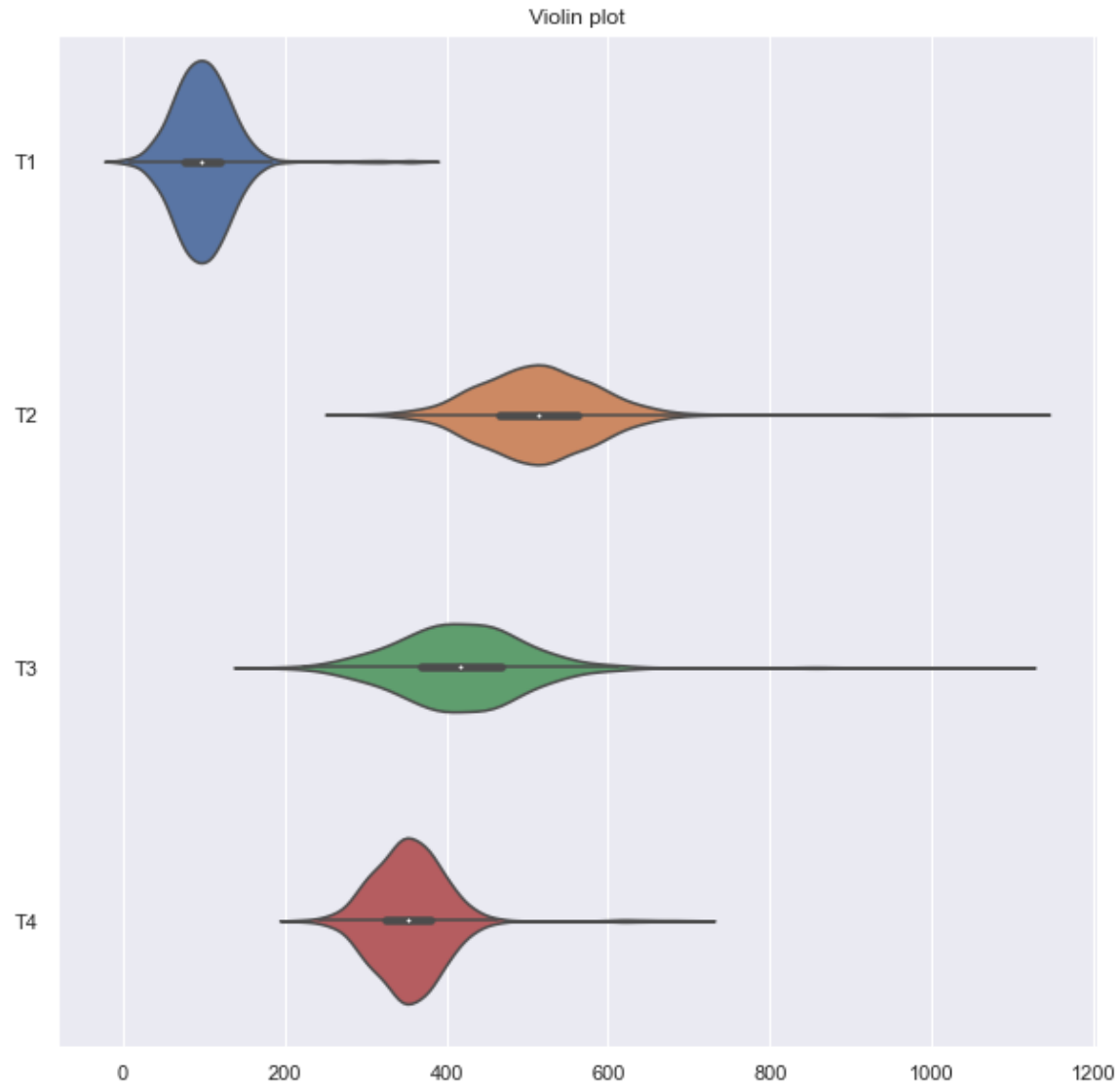


Figure 19: Violin plots: A different view of data distributions and outliers

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:

- 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()

```
def outliers_to_nan(df, features, n=3):
    # Annoying that pandas passes dataframes by reference
    # (I guess its quicker)
    df_to_clean = df.copy()
    for feature in features:
        mean_value = df_to_clean[feature].mean()
        std_dev = df_to_clean[feature].std()

        # Define the upper and lower bounds for outliers
        upper_bound = mean_value + n * std_dev
        lower_bound = mean_value - n * std_dev

        # Replace outliers with NaN values
        df_to_clean.loc[(df_to_clean[feature] > upper_bound)
            | (df_to_clean[feature] < lower_bound), feature] = np.NAN

    return df_to_clean
```

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.

```
outlier_features = ['T1', 'V1', 'A1', 'P1', 'B1','T2', 'V2', 'A2',
    'P2', 'B2','T3', 'V3', 'A3', 'P3', 'B3','T4', 'V4', 'A4', 'P4', 'B4' ]
df_no_outliers = outliers_to_nan(df, outlier_features, 3)
```

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

```
histogram(df_no_outliers,'V1',20)
```

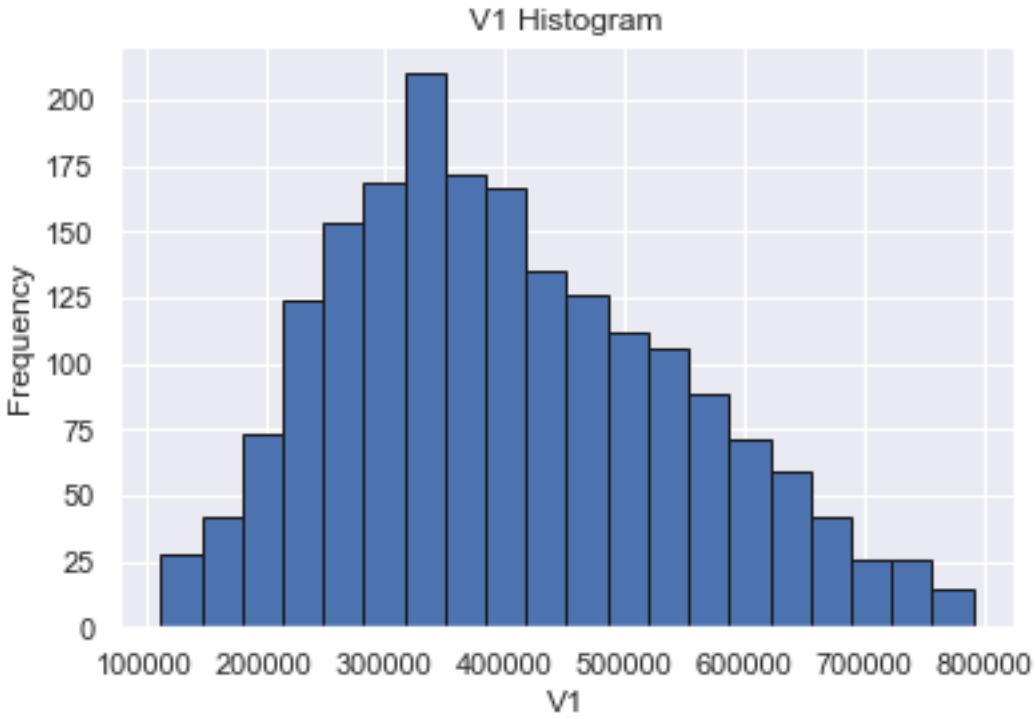


Figure 20: Histograms of data after removal of outliers

```
histogram(df_no_outliers,'A2',20)
```

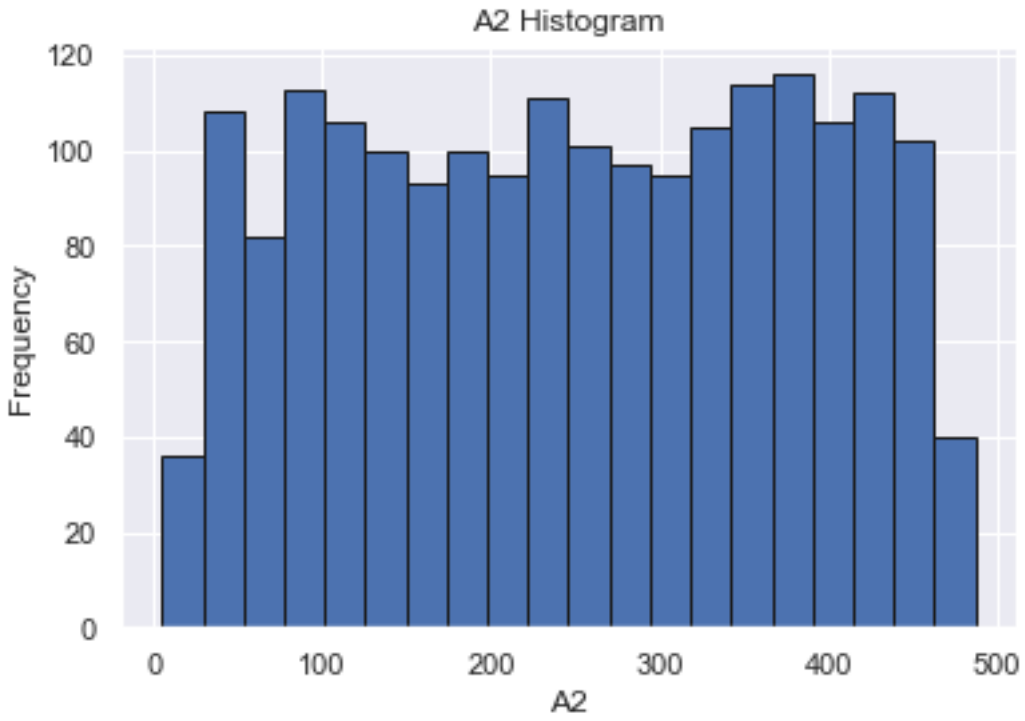


Figure 21: Histograms of data after removal of outliers

```
histogram(df_no_outliers,'V2',20)
```

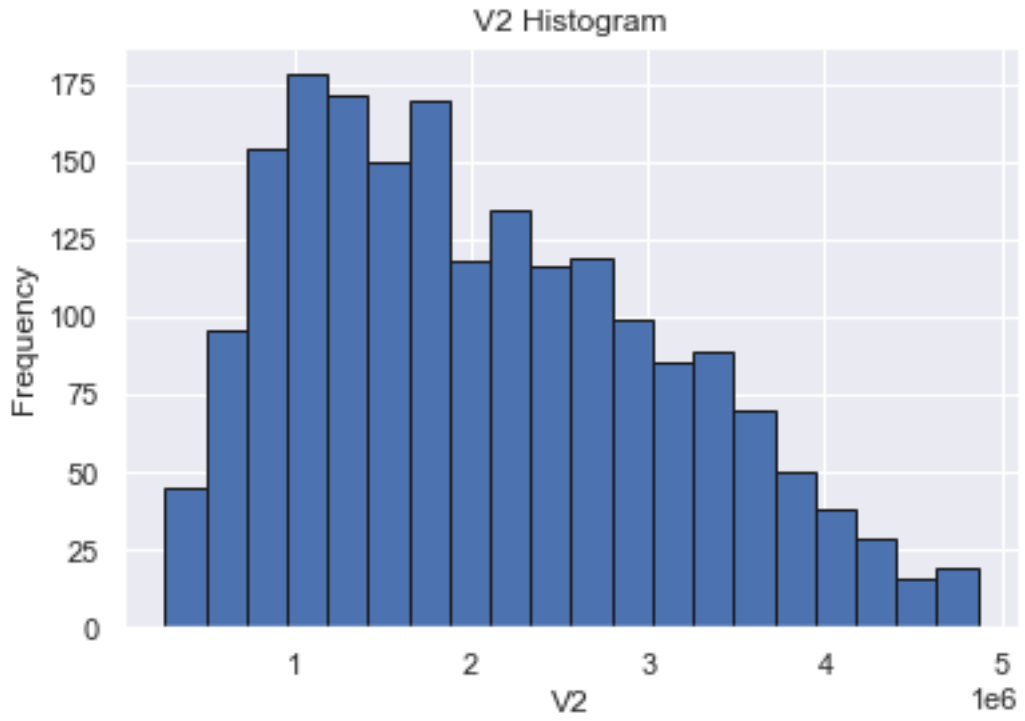


Figure 22: Histograms of data after removal of outliers

```
histogram(df_no_outliers,'V3',20)
```

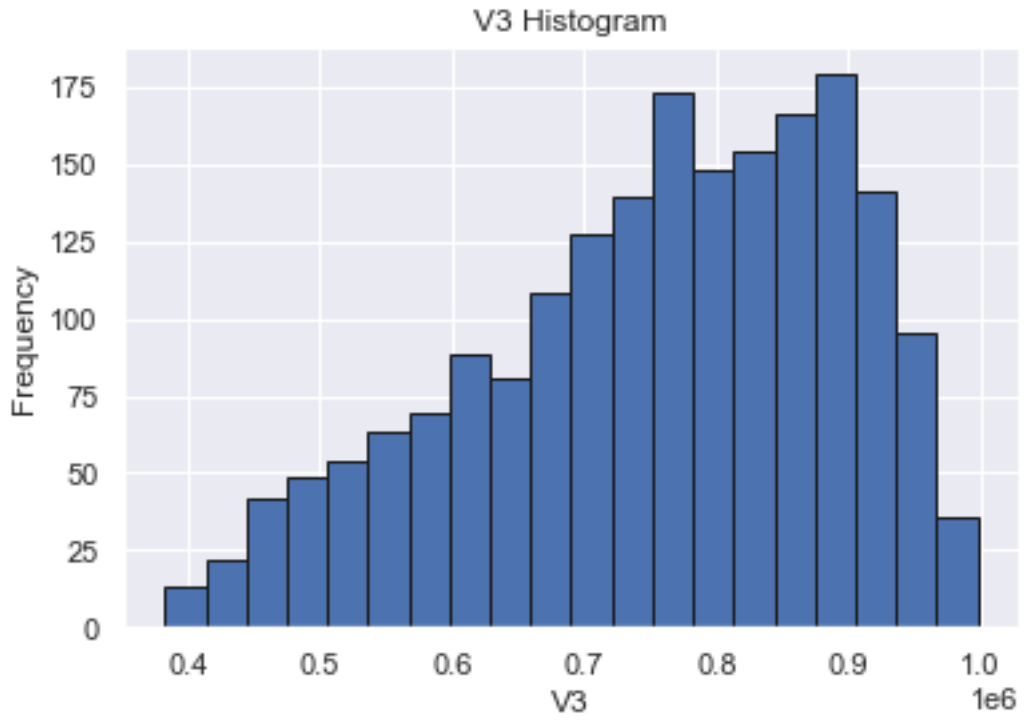


Figure 23: Histograms of data after removal of outliers

```
histogram(df_no_outliers,'V4',20)
```

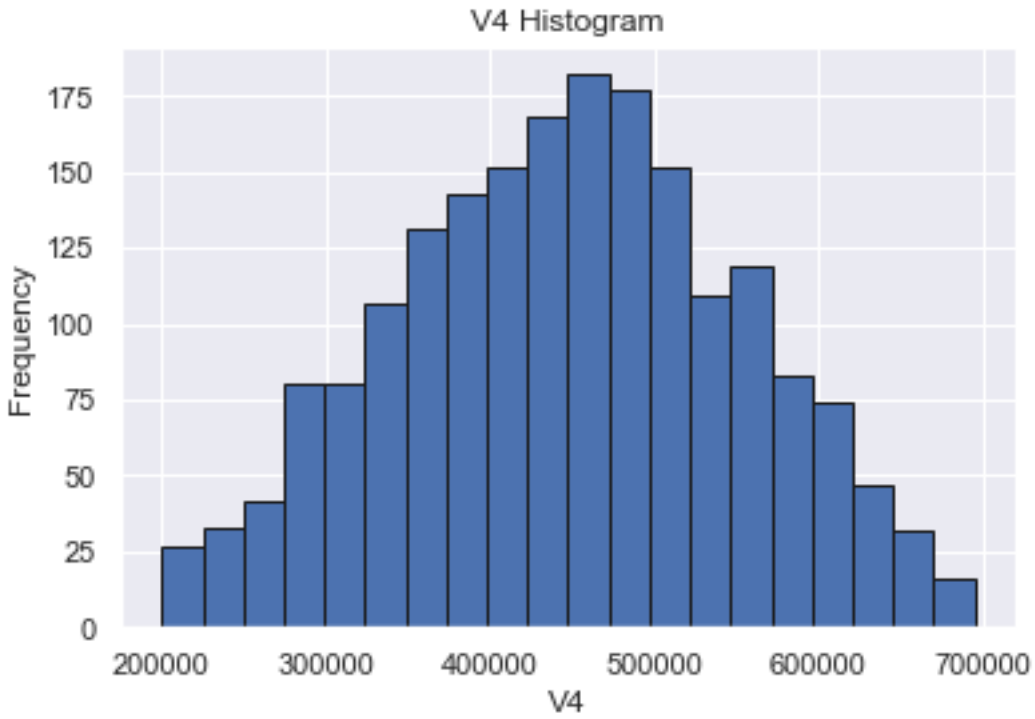


Figure 24: Histograms of data after removal of outliers

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

```
print("df.info =")
df.info()
```

```
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
 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
dtypes: float64(24), int64(4), object(2)
memory usage: 468.9+ KB
```

```
print("df_no_outliers.info() = ")
df_no_outliers.info()
```

```
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
```

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'):
```

```python
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

```
df_imputed = impute_nan_values(df_no_outliers, 'T1', 'mean')
```

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.

```
df_imputed = impute_nan_values(df_imputed, 'T2', 'mean')
df_imputed = impute_nan_values(df_imputed, 'T3', 'mean')
df_imputed = impute_nan_values(df_imputed, 'T4', 'mean')
```

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

```
msno.matrix(df_imputed)
plt.show()
```

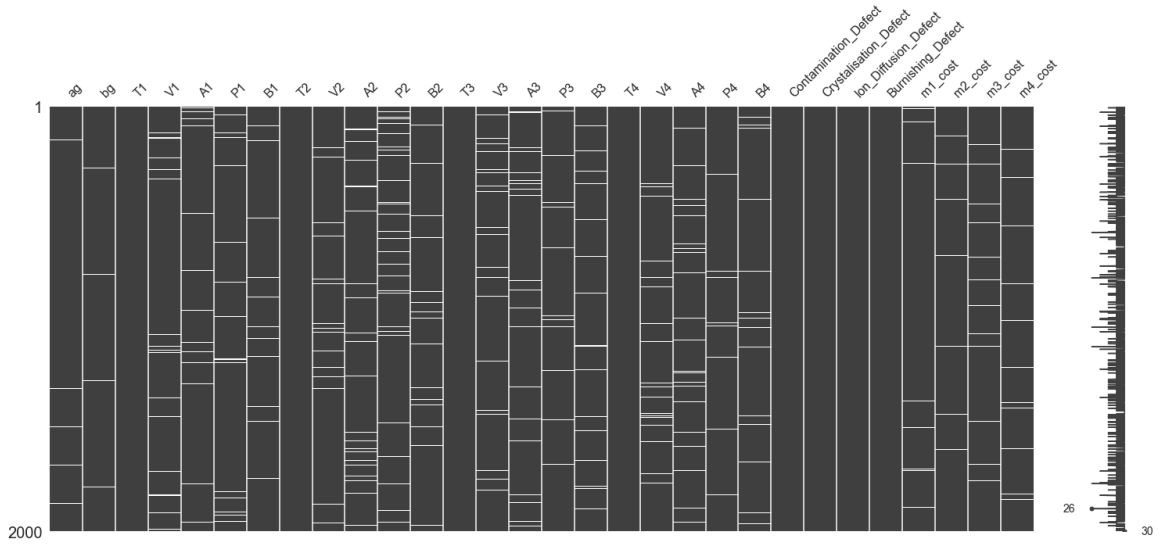


Figure 25: Results of imputing missing values of Tn

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

```
df_imputed = impute_nan_values(df_imputed, 'm1_cost', 'mean')
df_imputed = impute_nan_values(df_imputed, 'm2_cost', 'mean')
df_imputed = impute_nan_values(df_imputed, 'm3_cost', 'mean')
df_imputed = impute_nan_values(df_imputed, 'm4_cost', 'mean')
```

And check again using the missingno matrix function:

```
msno.matrix(df_imputed)
plt.show()
```

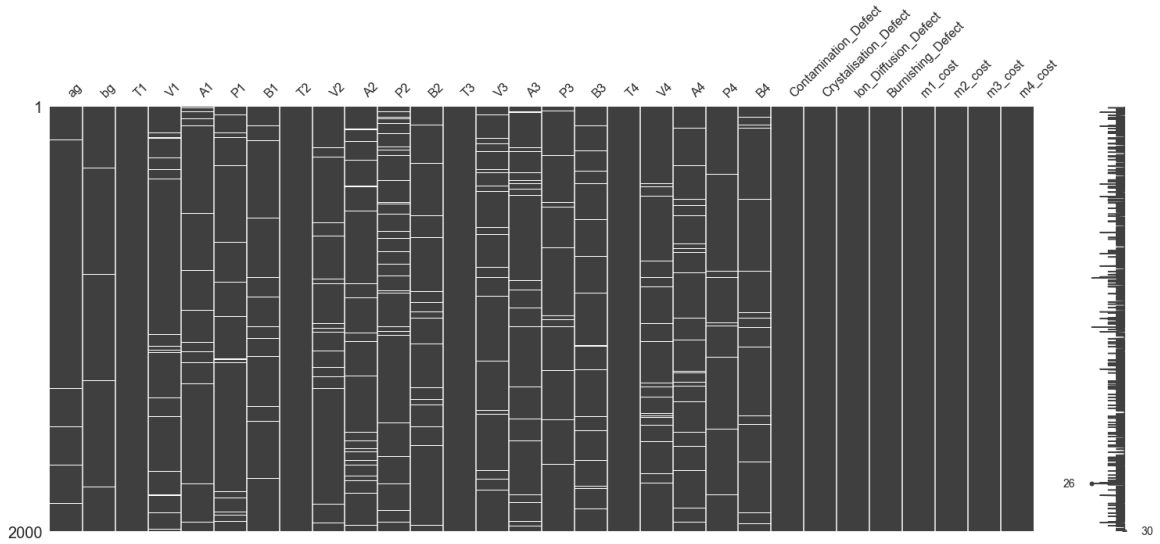


Figure 26: Results of imputing missing values for cost variables

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

```
median_impute_features =  ['V1','A1','P1','B1','V2','A2','P2','B2',
    'V3','A3','P3','B3','V4','A4','P4','B4']
for feature in median_impute_features:
    df_imputed = impute_nan_values(df_imputed, feature, 'median')
```

And display the results using missingno

```
msno.matrix(df_imputed)
plt.show()
```

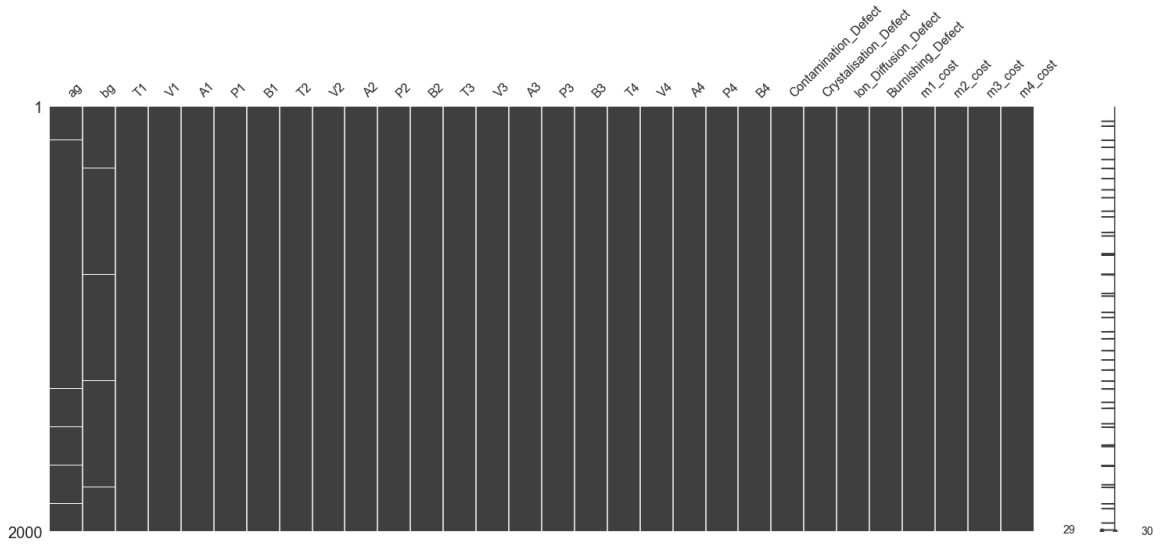


Figure 27: Final results of imputing missing values for all numerical variables

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', 'meadian' 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'

```
print("df_no_outliers.info() = ")
df_no_outliers.info()
```

```
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

```
1 imp = IterativeImputer(max_iter=50, random_state=0)
2 print(imp)
```

```
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'

```
imputed = imp.fit_transform(df_no_outliers.loc[:,'T1':])
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

```
df_imputed = pd.DataFrame(imputed, columns = df_no_outliers.loc[:,'T1':].columns )
```

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

```
df_imputed[['ag','bg']] = df[['ag','bg']]
```

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

```
msno.matrix(df_imputed)
plt.show()
```

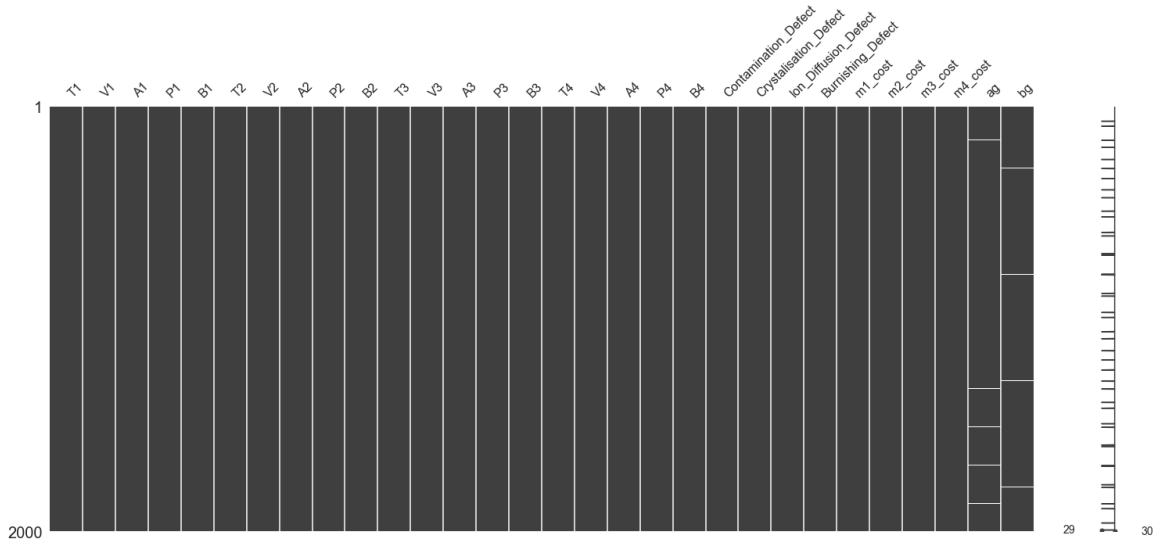


Figure 28: msno matrix of dataframe after imputing with IterativeImputer

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']])

```
imputer = SimpleImputer( strategy='most_frequent')
imputer.fit(df_imputed[['ag', 'bg']])
df_imputed[['ag', 'bg']] = imputer.transform(df_imputed[['ag', 'bg']])
```

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

```
msno.matrix(df_imputed)
plt.show()
```

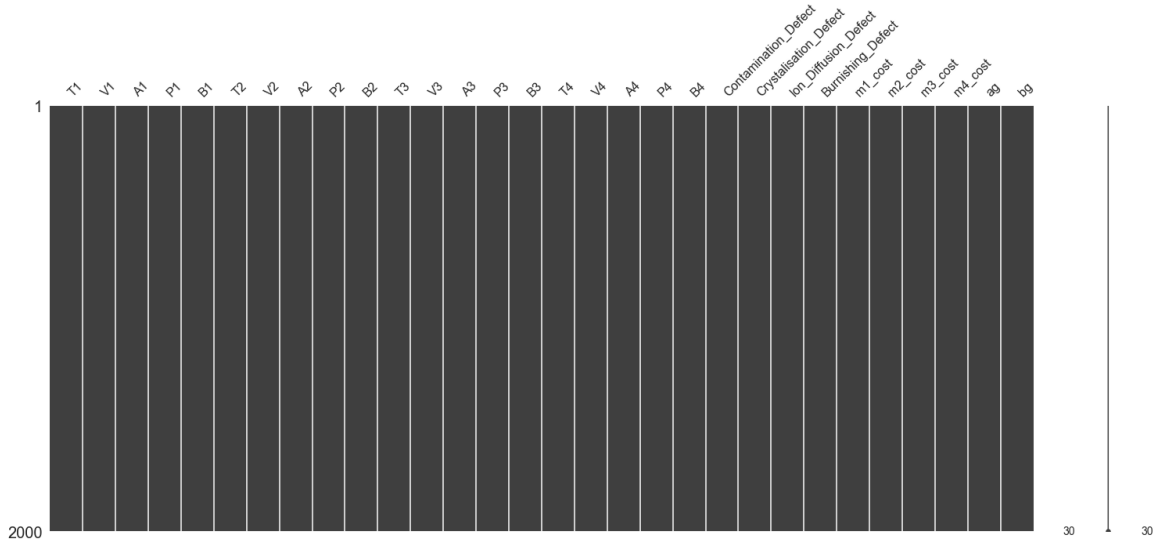


Figure 29: Final results of imputing all missing values

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

```
features_to_be_standardized = ['T1','V1','A1', 'B1','T2','V2','A2', 'B2',
                               'T3','V3','A3', 'B3','T4','V4','A4',
                               'B4']
print(f"features_to_be_standardized =\n[")

for feature in features_to_be_standardized:
    print(f"{feature},")
print("]")
```

```
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

```
plotBox(df_imputed, features_to_be_standardized)
plt.show()
```

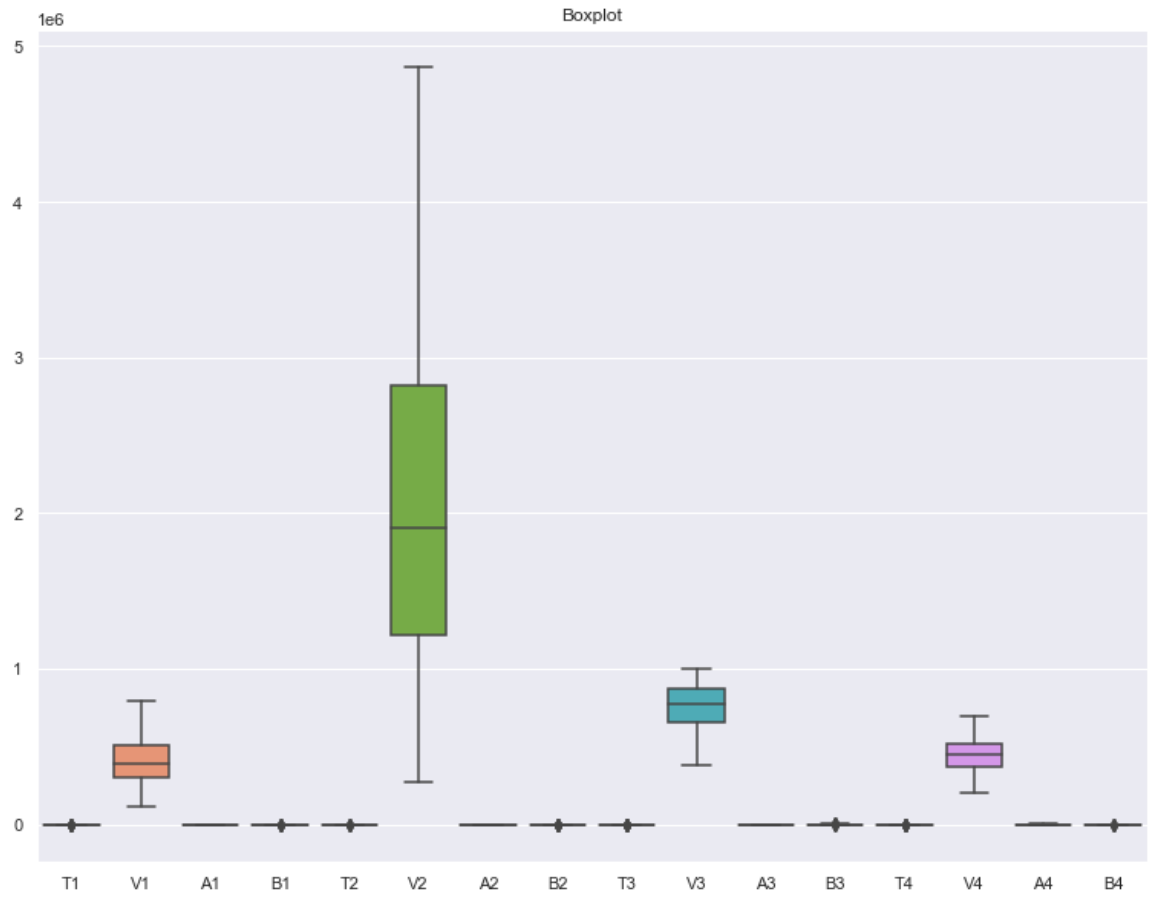


Figure 30: Box-plot of all variables showing significant scale differences

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

```
def feature_standardize(df, features_to_standardize):

    features_subset = df[features_to_standardize]
    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(features_subset)
    df_standardized = pd.DataFrame(scaled_features,
        columns=features_to_standardize)

    return(df_standardized)
```

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

```
df_standardized = feature_standardize(df_imputed,
                                      features_to_be_standardized)
```

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

```
plotBox( df_standardized, features_to_be_standardized)
plt.show()
```

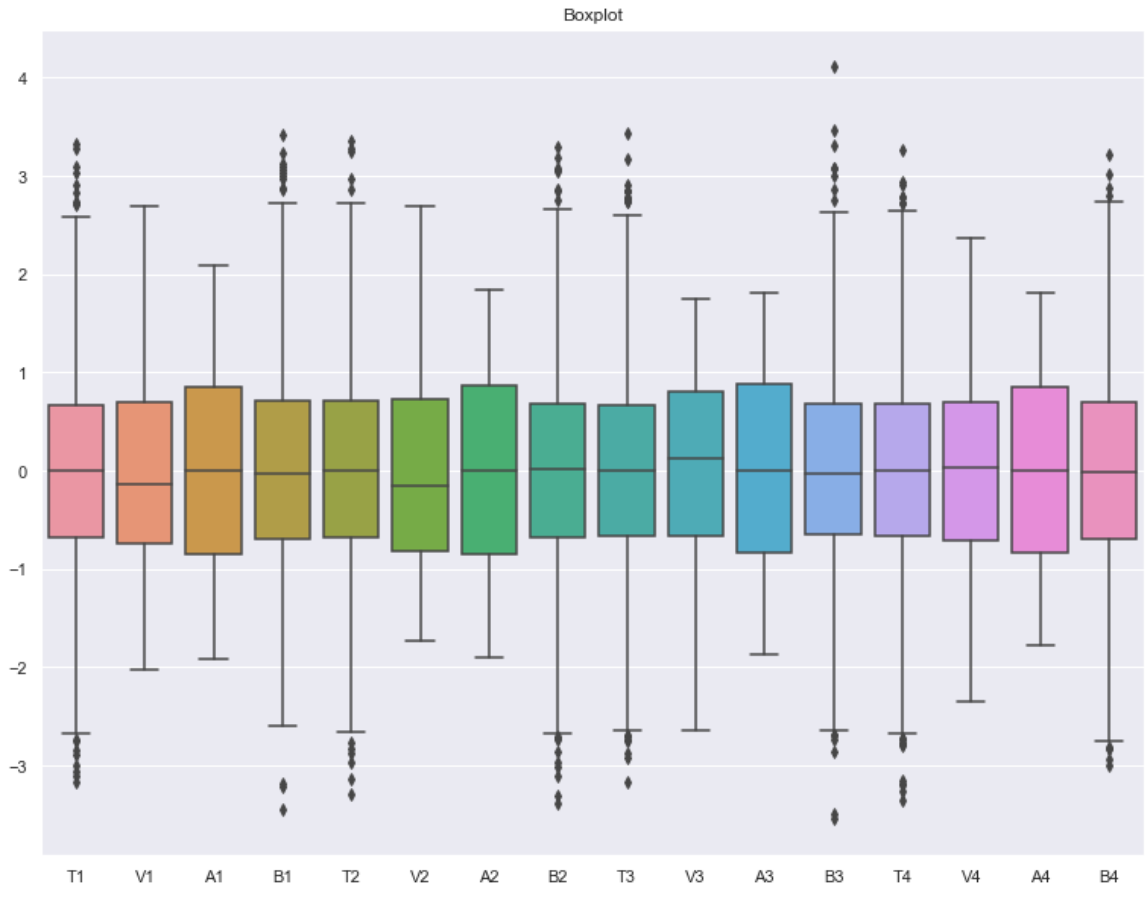


Figure 31: Box-plot of all variables after standardization

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 scalling 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.

```
def feature_standardize(df, features_to_standardize):

    features_subset = df[features_to_standardize]
    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(features_subset)
    df_standardized = pd.DataFrame(scaled_features,
        columns=features_to_standardize)

    return(df_standardized, scaler.scale_, scaler.mean_)
```

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

```
df_standardized, scales, means = feature_standardize(df_imputed,
                                        features_to_be_standardized)
print("List of scaling parameters for each feature:")
print(scales)

print("List of means for each feature")
print(means)
```

```
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
 3.51833919e+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)

```
features_to_copy = ['Contamination_Defect',
    'Crystalisation_Defect',
    'Ion_Diffusion_Defect',
    'Burnishing_Defect',
    'ag',
```

```
    'bg',
    'm1_cost',
    'm2_cost',
    'm3_cost',
    'm4_cost']
df_standardized[features_to_copy] = df_imputed[ features_to_copy]
df_standardized.loc[:, 'T1':'B2']
```

	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
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

```
df_standardized.loc[:, 'T3':'B4']
```

	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

```
df_standardized.loc[:, 'Contamination_Defect':'Burnishing_Defect']
```

	Contamination_Defect	Crystalisation_Defect	Ion_Diffusion_Defect	Burnishing_Defect
0	0.0	0.0	0.0	0.0

	Contamination_Defect	Crystalisation_Defect	Ion_Diffusion_Defect	Burnishing_Defect
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

```
df_standardized.loc[:, 'ag':'m4_cost']
```

	ag	bg	m1_cost	m2_cost	m3_cost	m4_cost
0	Low	Argon	1139.53	2064.62	1156.56	840.47
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	C02	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 out 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 difference 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}`

```
ag_mapping = {'Low': 1, 'Medium': 2, 'High': 3}
```

... 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.

```
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

```
df_ag_fixed = transform_ordinal_to_numeric(df_standardized, 'ag', ag_mapping)
df_ag_fixed.loc[:, 'ag':'ag']
```

	ag
0	1
1	1
2	2

	ag
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.

```
1 df_final = pd.get_dummies(df_ag_fixed, columns=['bg'])
```

Display those new features using the code:

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

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

	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.

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

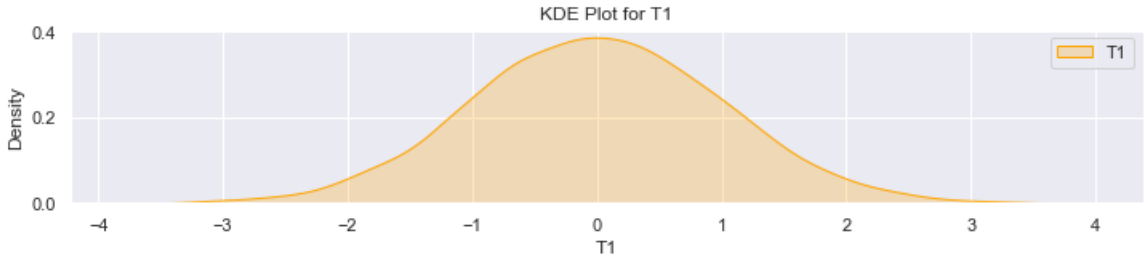
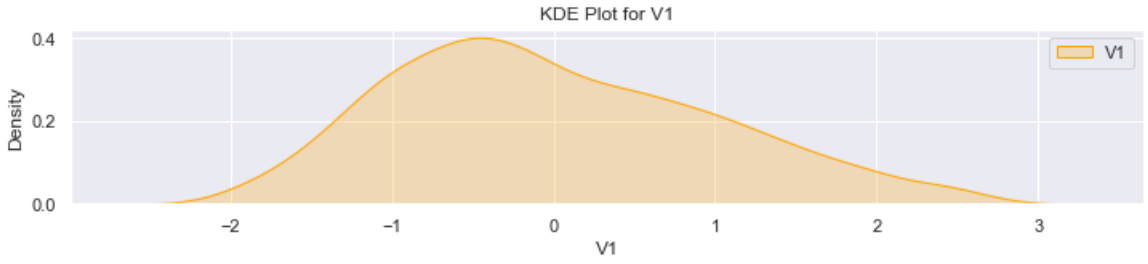
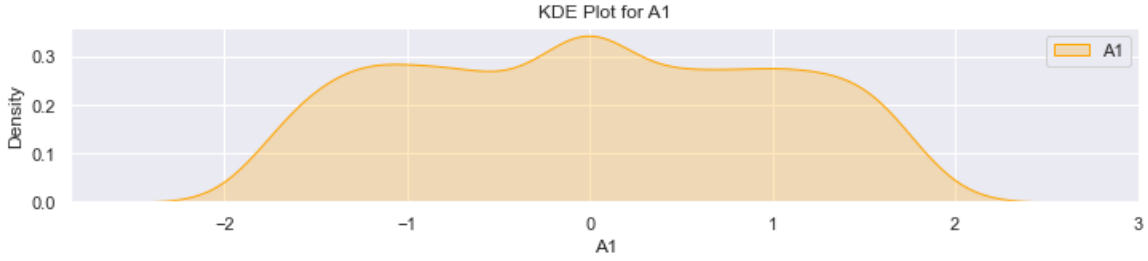
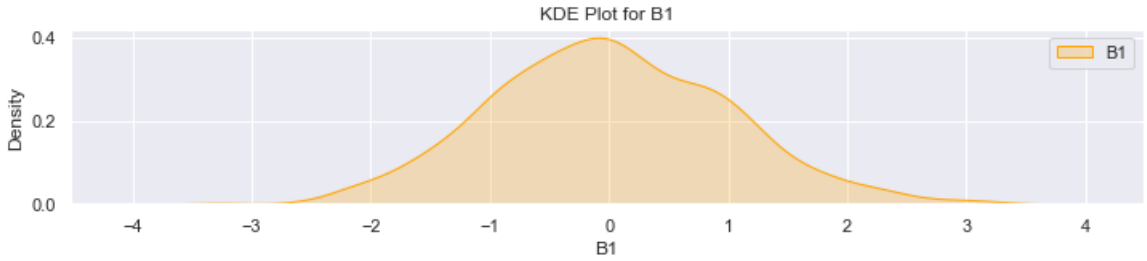
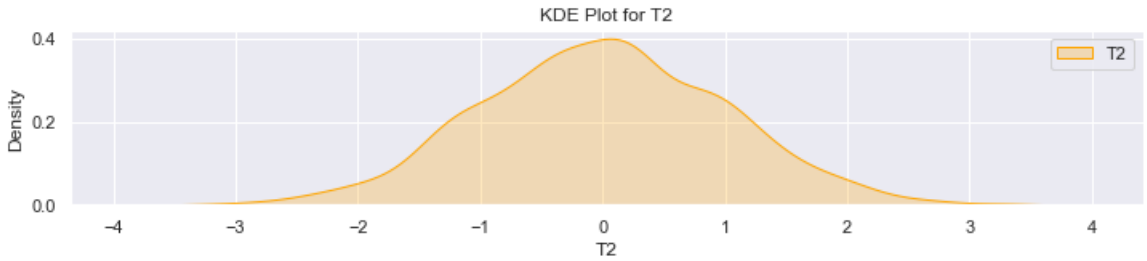


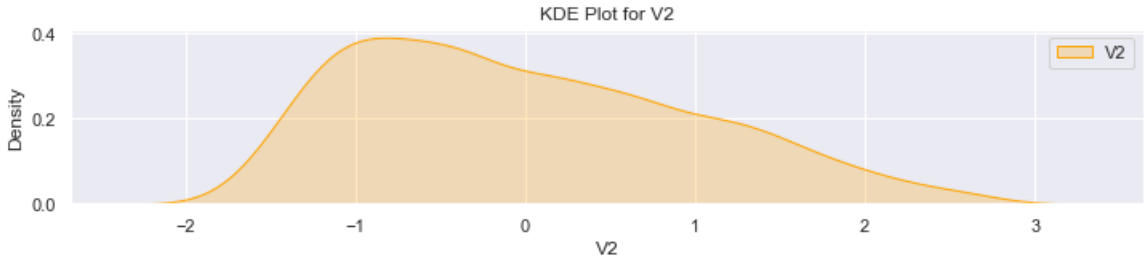
Figure 32: kde plot of the data

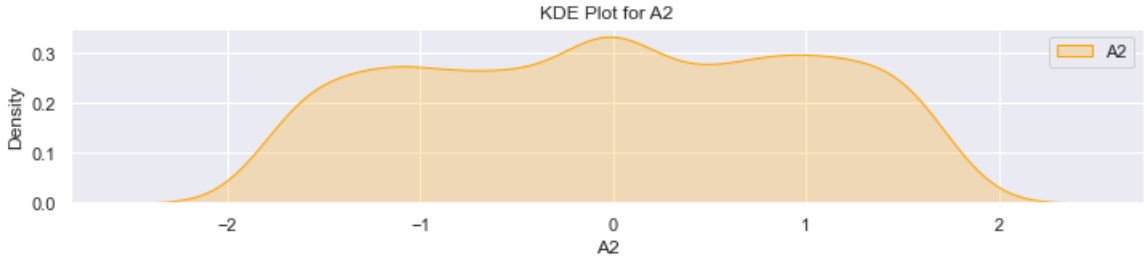


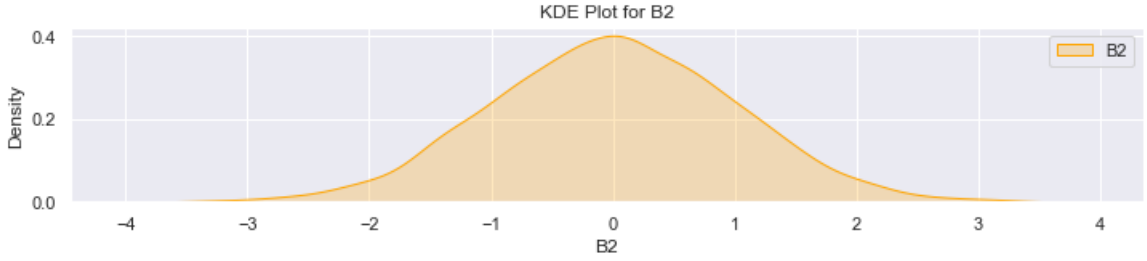


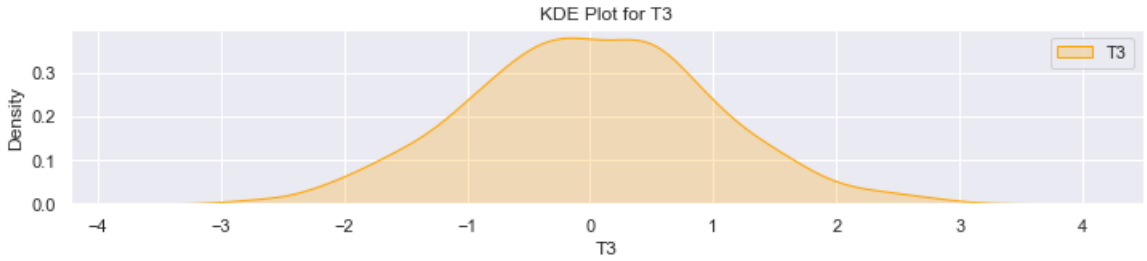

KDE Plot for B1
Density
B1
0.0
0.2
0.4
−4
−3
−2
−1
0
1
2
3
4

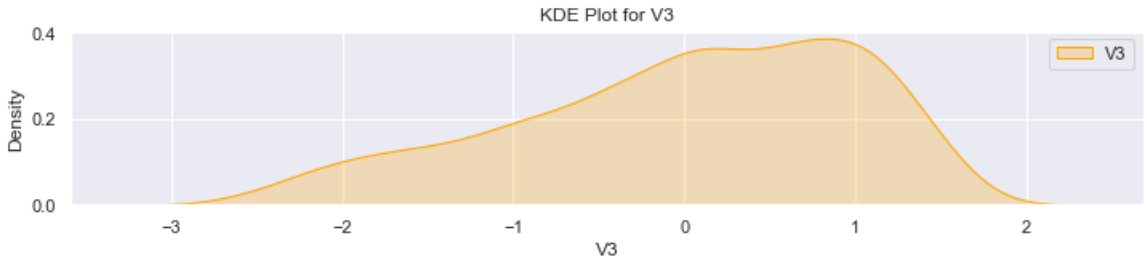

KDE Plot for T2
Density
T2
0.0
0.2
0.4
−4
−3
−2
−1
0
1
2
3
4

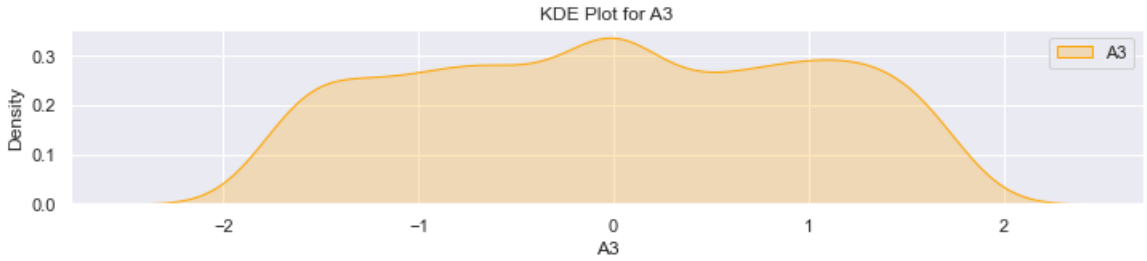

KDE Plot for V2
Density
V2
0.0
0.2
0.4
−2
−1
0
1
2
3

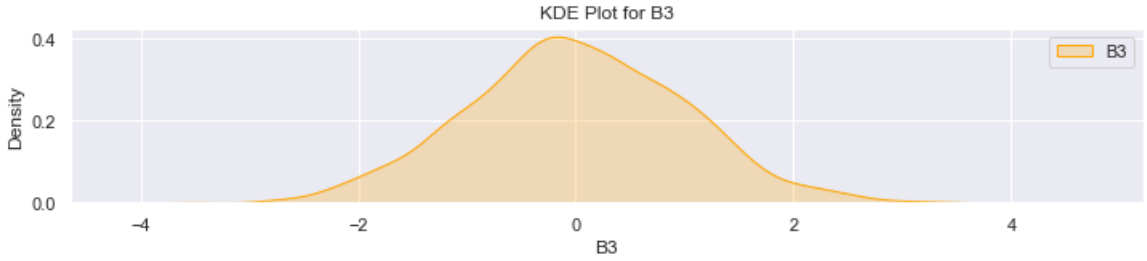

KDE Plot for A2
Density
A2
0.0
0.1
0.2
0.3
−2
−1
0
1
2

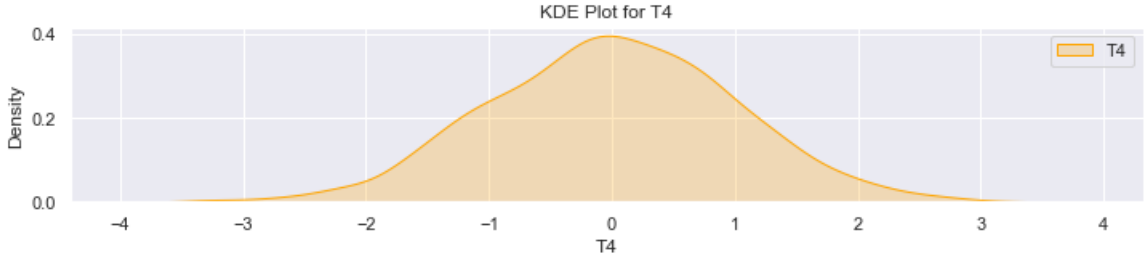

KDE Plot for B2
Density
B2
0.0
0.2
0.4
−4
−3
−2
−1
0
1
2
3
4

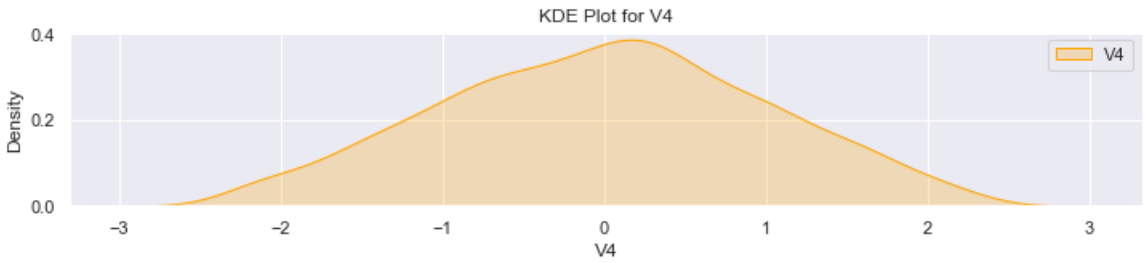

KDE Plot for T3
Density
T3
T3

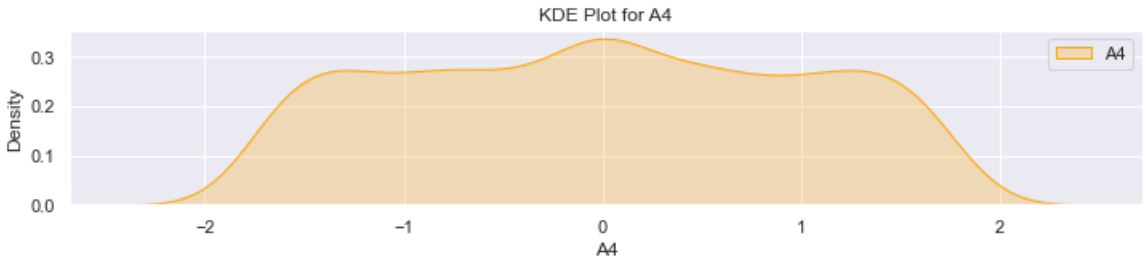

KDE Plot for V3
Density
V3
V3

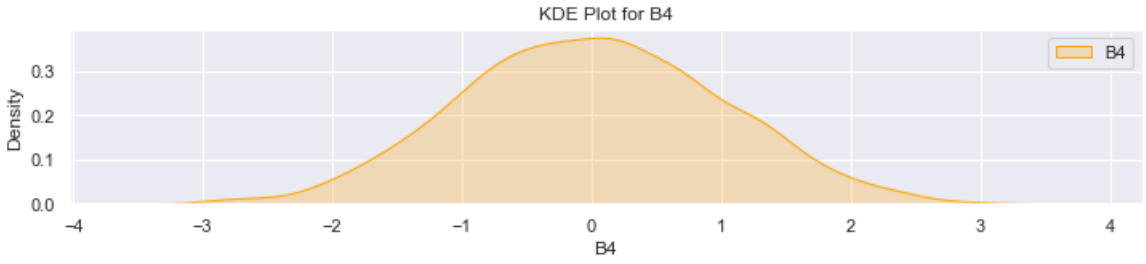

KDE Plot for A3
Density
A3
A3

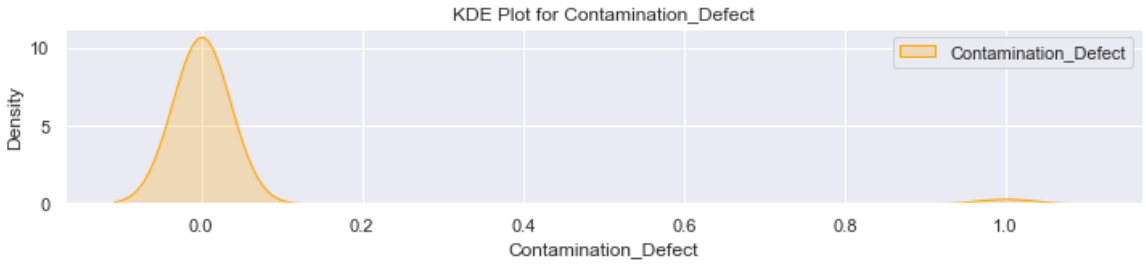

KDE Plot for B3
Density
B3
B3

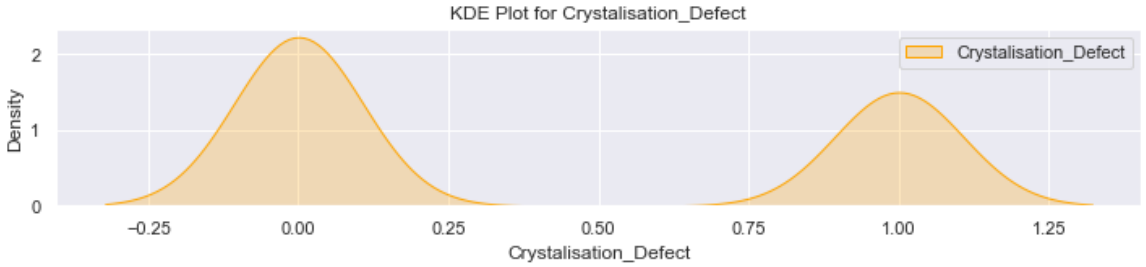

KDE Plot for T4
Density
T4
T4

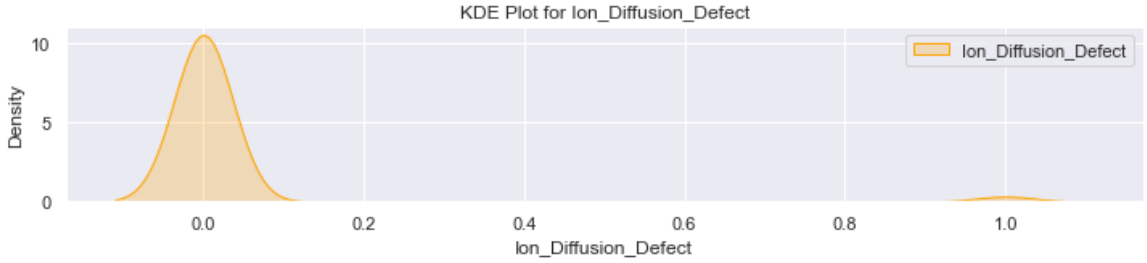

KDE Plot for V4
Density
V4
0.0
0.2
0.4
−3
−2
−1
0
1
2
3

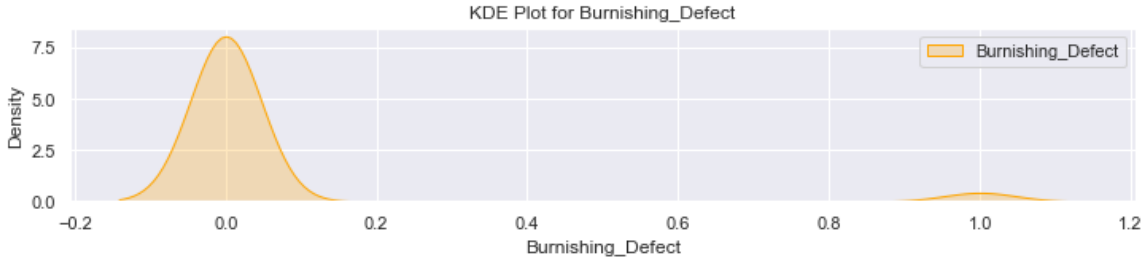

KDE Plot for A4
Density
A4
0.0
0.1
0.2
0.3
−2
−1
0
1
2

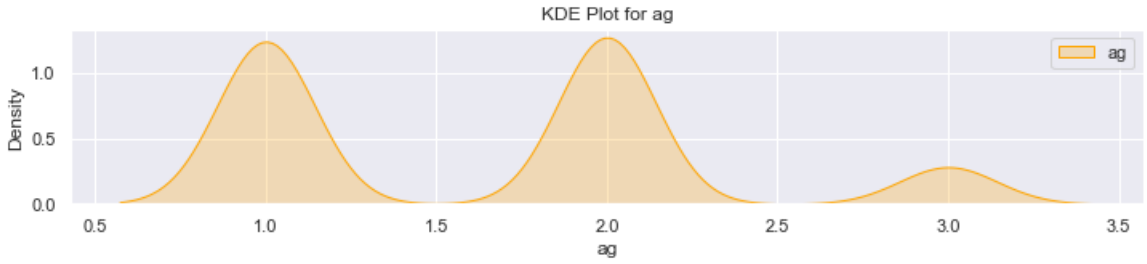

KDE Plot for B4
Density
B4
0.0
0.1
0.2
0.3
−4
−3
−2
−1
0
1
2
3
4


KDE Plot for Contamination_Defect
Density
Contamination_Defect
0
5
10
0.0
0.2
0.4
0.6
0.8
1.0

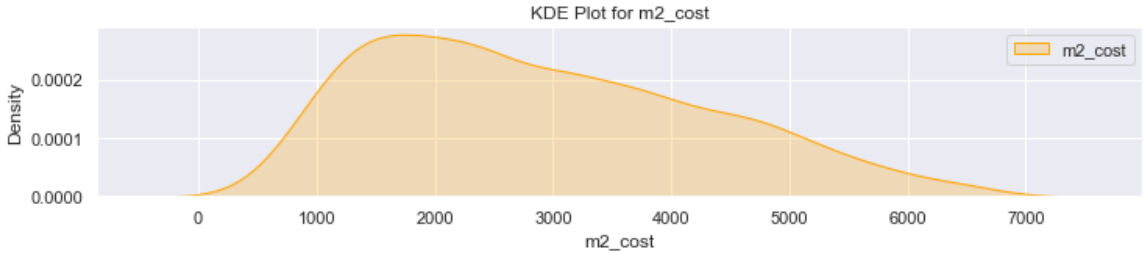

KDE Plot for Crystalisation_Defect
Density
Crystalisation_Defect
0
1
2
−0.25
0.00
0.25
0.50
0.75
1.00
1.25


KDE Plot for Ion_Diffusion_Defect
Ion_Diffusion_Defect
Density
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
Ion_Diffusion_Defect

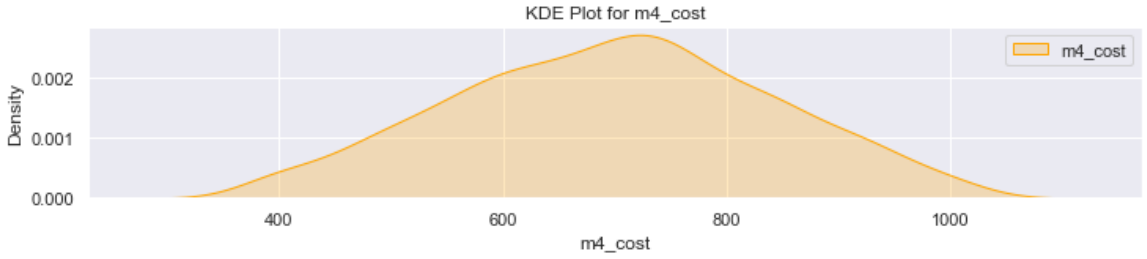

KDE Plot for Burnishing_Defect
Burnishing_Defect
Density
7.5
5.0
2.5
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Burnishing_Defect

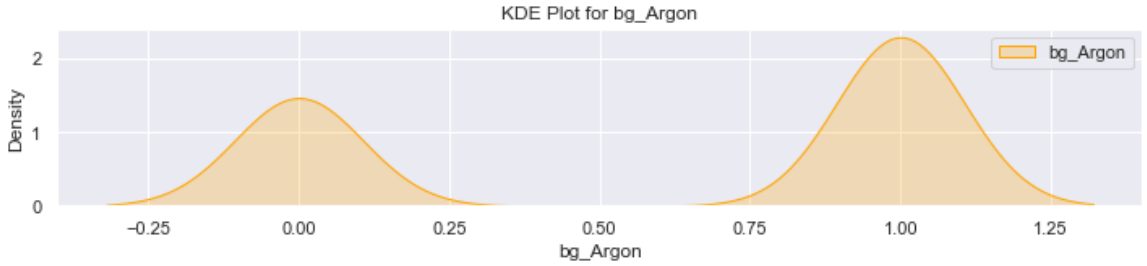

KDE Plot for ag
ag
Density
1.0
0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
ag

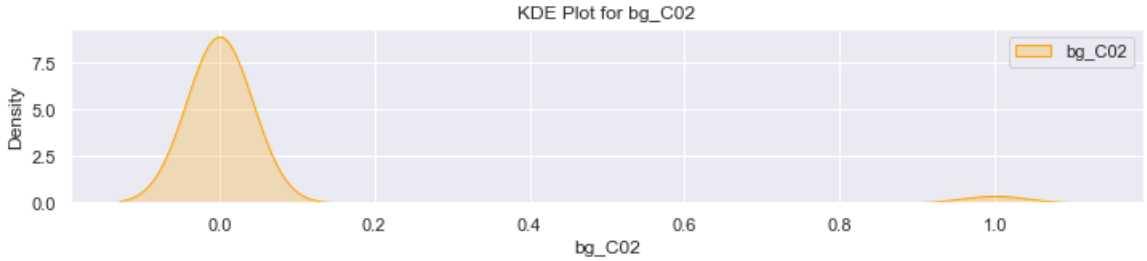

KDE Plot for m1_cost
m1_cost
Density
0.0010
0.0005
0.0000
250
500
750
1000
1250
1500
1750
2000
2250
m1_cost

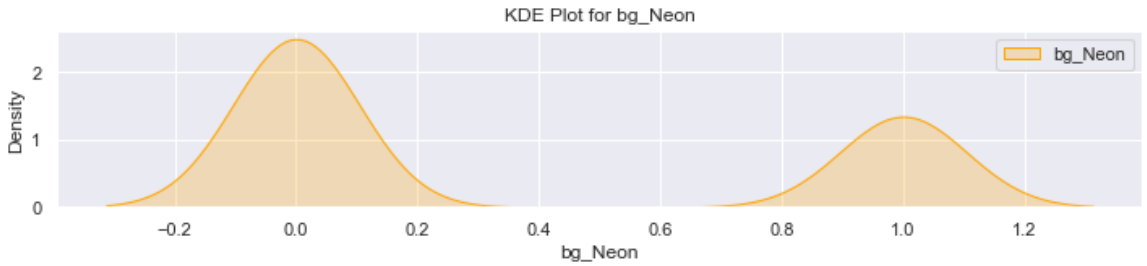

KDE Plot for m2_cost
m2_cost
Density
0.0002
0.0001
0.0000
0
1000
2000
3000
4000
5000
6000
7000
m2_cost


KDE Plot for m3_cost
m3_cost
Density
0.002
0.001
0.000
600
800
1000
1200
1400
1600
m3_cost


KDE Plot for m4_cost
m4_cost
Density
0.002
0.001
0.000
400
600
800
1000
m4_cost


KDE Plot for bg_Argon
bg_Argon
Density
2
1
0
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
bg_Argon


KDE Plot for bg_C02
bg_C02
Density
7.5
5.0
2.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
bg_C02


KDE Plot for bg_Neon
bg_Neon
Density
2
1
0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
bg_Neon

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.

```
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()

plt.show()
```

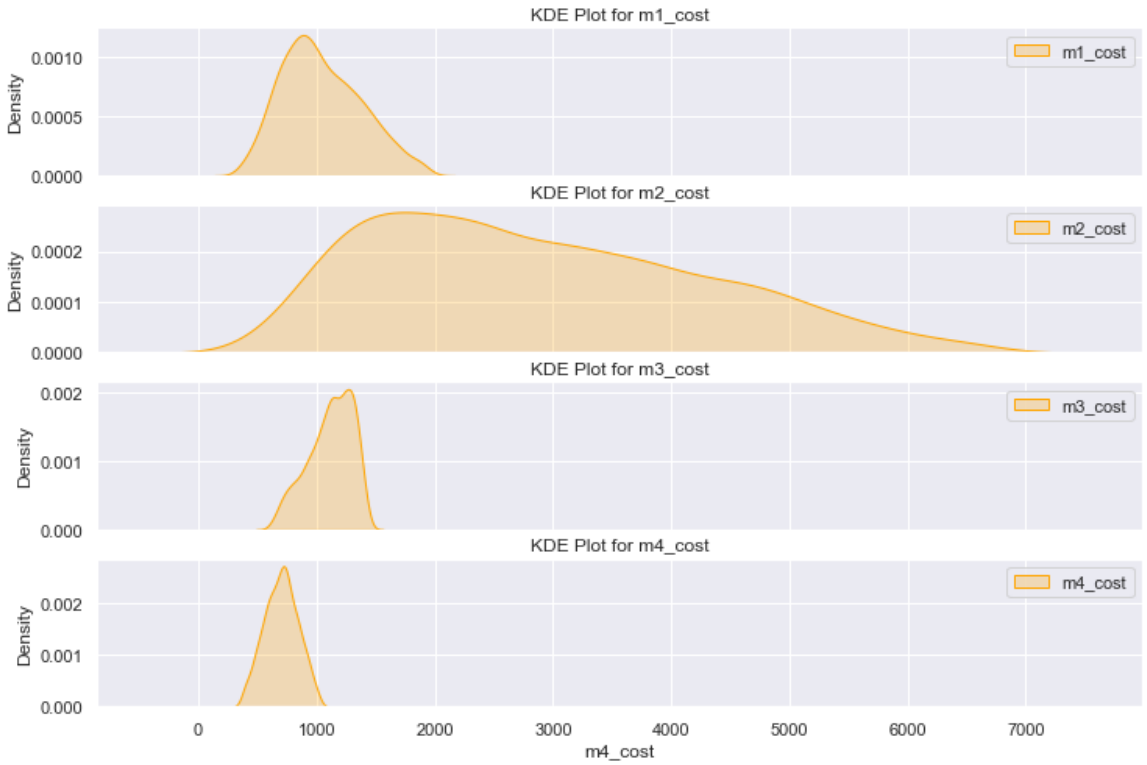


Figure 33: kde plots of selected features using a common x axis scale

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

```
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()
```

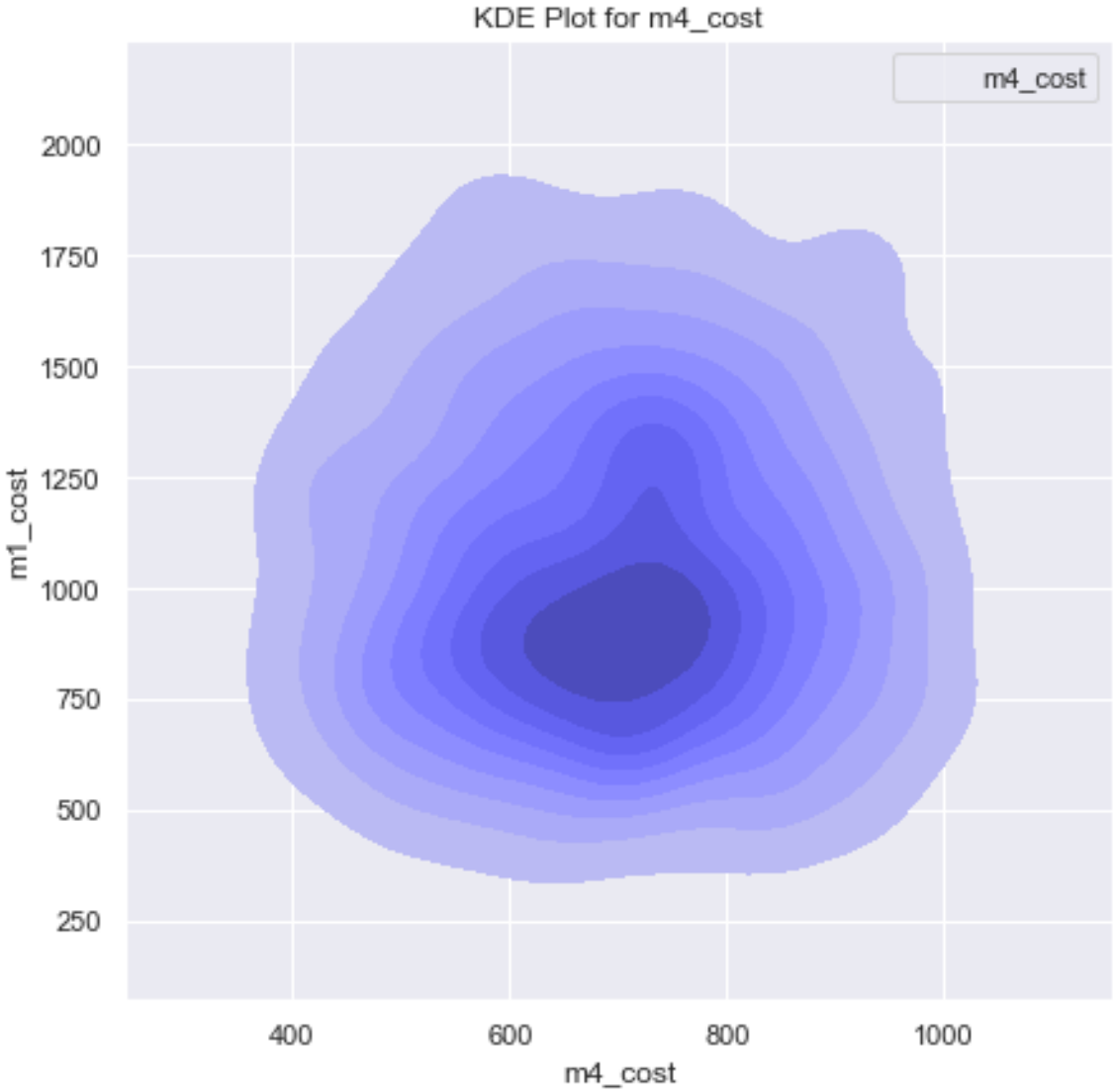


Figure 34: 2-Dimensional kde (surface contour) plot of selected features

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.

```
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()
```

```
diagnostic_plots(df_final, 'V1')
```

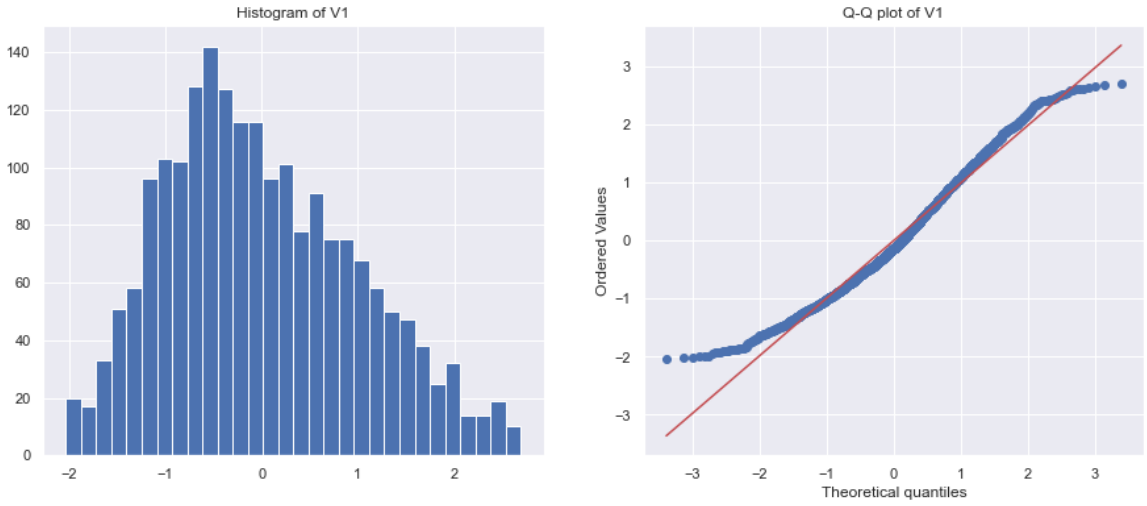


Figure 35: Q-Q plot to help show deviation from a Gaussian distribution