A Guide to Principal Component Analysis (IAPT)

Matthias Bartolo
B.Sc. IT (Hons) Artificial Intelligence Student
matthias.bartolo.21@um.edu.mt
University of Malta
Msida, Malta

1 INTRODUCTION

1.1 Definition and Practical Application of PCA

principal component analysis (PCA) is an incredibly useful, and widely used multivariate algorithm in Machine Learning. Moreover, such algorithm is also extremely helpful in the analysis of huge datasets, whilst effectively undertaking *dimensionality reduction* and *feature selection*. Furthermore, PCA is used to ensure that data scientists may load and utilise large datasets on less powerful machines, which could not support the size of the full dataset. Additionally, PCA also provides cleaner data visualisation through the envisioning of the key data features in the full dataset, which hold the largest degree of information. [1-2]

1.2 History

PCA has a long, and illustrious history that goes back more than a century. The algorithm was pioneered by Karl Pearson, who in 1901 launched this system with the aim of undertaking data analysis and dimensionality reduction. The current PCA's design was first pioneered by Harold Hotelling in the 1930s, who led the way for the method to truly take shape. Hotelling was instrumental in formulating the concept of variance maximization, and the use of orthogonal projections to find the *principal components*. [1-2]

Further improvements to the PCA algorithm were developed in the 1960s, in part due to the emergence of *singular value decomposition (SVD)*, which offered an alternate method for calculating the eigenvalues and vectors, necessary to perform the PCA algorithm. The growing adoption of PCA at that time was largely triggered by the need for *dimensionality reduction* and the widespread growth of computers. Consequently, the method gained a lot of popularity in the 1970s and later on when data scientists and researchers fully comprehended the effectiveness of such technique in dealing with enormous and complex datasets which were becoming more and more prevalent in industries such as banking, engineering, and medicine. [1-2]

1.3 Aims and Objectives

Nevertheless, such algorithm's behaviour may not always be comprehensible, thus triggering the need for the development of a visual tool, which allow users the possibility of visualising the algorithm's stages and data transformations, whilst offering a better understanding on the modified data. Consequently, the programmed solution also effectively portrays the PCA process as a simple convenient methodology which may be explained to students who have just completed a *linear algebra* or *AI numerical methods* course. Additionally, the developed Jupyter notebook which outlines the aforementioned process, conveys to the students the necessary information to understand better such algorithm, whilst providing them with essential tools, to experiment and expand their knowledge. Furthermore, the notebook's characteristics of being robust and responsive, allow students to interact with the visual plots through the plot's minimising and maximising tools. The Jupyter notebook which incorporates the developed solution, will be complemented with various famous datasets utilised by the machine learning community such as the Iris dataset, with the aim of making the students familiar with such datasets. One must note that the datasets were chosen for their distinct properties, to allow students to evaluate different experiments and infer new knowledge.

1.4 Summary of Results

The noteworthy key points attained from the creation of the Jupyter notebook are listed hereunder:

(1) *principal component analysis* is designed to be utilised on linearly separable data i.e., the variables in the dataset need to be linked together through a linear relationship. Consequently, *kernel PCA* can be used to resolve the above-mentioned limitation [3].
(2) The PCA algorithm is intended to be used on continuous data values. The approach taken to cater for discrete value features in the dataset, included changing the discrete values to continuous values, through different encoding techniques. Moreover, such discrete columns which could also be discarded before the calculation of the PCA, thus resulting in more accurate data projections, at the cost of losing the discrete data columns.
(3) There are different libraries which may be used to implement PCA, for example the *NumPy* and the *scikit-learn library*. Furthermore, sometimes the NumPy library does not support the singular value decomposition (SVD) of large datasets and would crash. On the other hand, the scikit-learn library enables the implementation of PCA without terminating abruptly as it utilises random singular value decomposition. Furthermore, *randomized SVD* can approximate the whole SVD with a substantially lower computation cost by randomly selecting a fraction of the matrix's rows or columns [4].
(4) It is imperative that the data which is fed to the PCA algorithm is normalized, so as to avoid the PCA from loading on the high variance data. [5]. Additionally, the created artefact utilises *z-score normalization*.
(5) Understanding the PCA algorithm mathematically can be quite a challenge, as the process includes multiple mathematical calculations such as SVD or *covariance matrix* evaluation. Nevertheless, the developed artefact provides the need-to-know basis for such algorithm, in a well explained format.

2 BACKGROUND AND METHODOLOGY

Mathematically, PCA enables the conversion of linear continuous data into a new coordinate system, characterized by new axis (principal components) which are sequentially placed in accordance with the features in the new coordinate system. This enables that the best principal components are plotted on different dimensional graphs, thus presenting a satisfactory visualisation of a large dataset. Unfortunately, such method may have some minimal data reduction. However, visualising an n dimensional feature dataset on a 3D plot is quite a benefit. The PCA's main characteristics of decreasing the dimensionality of data, whilst retaining salient information, accounts for to be classified as the most effectively ranked data analysis and machine learning technique [1-2].

It is worth pointing out that, the designed implementation provides an in-depth description of such algorithm, whilst categorizing the explanation in the following ordered sections:

2.1 Libraries Utilised

Figure 1 illustrates the various libraries which were used in the construction of the aforementioned notebook.

```
import os
import random
import numpy as np
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
import gensim
import collections
import nltk
from os.path import exists
from sklearn.decomposition import PCA
from numpy import linalg as LA
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize
```

Figure 1: Libraries utilised.

2.2 Loading the Data

A key step before the initiation of the PCA Algorithm, entails the selection of a relevant dataset which will be analysed by such algorithm. The designed implementation enables students interacting with the notebook, the choice to select any of the default datasets, and explore how the PCA algorithm will function on such datasets. Students are also given the option to load their preferred dataset. In addition, the selected default datasets are characterised by different attributes, so as to allow students to carry out different experiments, facilitating comparative analysis of the results obtained through varying the datasets.

The following are the default datasets (obtained from [6-12]):

(1) **country_wise_latest.csv** - This dataset has a small size, a large number of features, and a few numbers of discrete columns.
(2) **diabetes.csv** - This dataset has a small size, a small number of features, and no discrete columns.
(3) **FIFA-2014.csv** - This dataset has a small size, a small number of features, and one discrete column.
(4) **IRIS.csv** - This dataset has a small size, a large number of features, and one discrete column.
(5) **Salary_Dataset_with_Extra_Features.csv** - This dataset has a large size, a small number of features, and a reasonable number of discrete columns.
(6) **spotify.csv** - This dataset has a large size, a large number of features, and a reasonable number of discrete columns.
(7) **wine-quality-white-and-red.csv** - This dataset has a large size, a large number of features, and one discrete column.

The aforementioned functionality of loading the chosen dataset into a pandas data frame illustrated in Figure 2. The presented code snippet lists a Menu. Depending on the choice to load a preferred dataset or load a default dataset, the user will be given a relevant message to input the file path or name respectively. In case that the user selects an invalid option, the program will continue to loop, until the user has successfully inputted a valid input. In case that the user chooses to load a default dataset, the user is presented with the list of default csv files present in the datasets folder. Consequently, retrieving such dataset names is executed dynamically through the os.listdir function, thus rendering the addition of another default dataset to the datasets folder relatively simple. On the other hand, if the user decides to load a preferred dataset, a relevant check is run to ascertain that the specified file exists. Finally, the specified file name is loaded through the pd.read_csv function, and stored in a pandas data frame. In case that the specified csv file has less than 3 columns, a Warning message is displayed to the user, advising him that the dataset will present Errors in the data visualisations later on. Additionally, calculating the PCA algorithm on a dataset which has less than three columns presents unsatisfactory results. Furthermore, applying the concept of dimensionality reduction on a dataset which already has reduced size, is not permissible.

```
#Giving the user the option to choose either a default dataset, or to enter his/her own dataset path
filename=None
path=""
#Looping until the user enters a valid choice
while True:
    #Displaying Menu
    print("\033[1m \nChoosing From the following Options: \033[0m \n1.Load a Default Dataset\n2.Choose a Requested Dataset")
    choice=int(input())
    if(choice==1):
        #Showing a list of default datasets to the user, and awaiting valid user choice
        print("\033[1m \nChoose from the Following Default Datasets: \033[0m")
        validDataset=[]
        for dataset in os.listdir("Datasets"):
            validDataset.append(dataset)
            print(dataset)
            #Looping until user, enters a valid File Name
        while filename not in validDataset:
            print("\nPlease Input File Name:")
            filename=input()
        #Constructing File Path
        path="Datasets/"+filename
        break

    elif(choice==2):
        #Giving the user the option to load a preferred dataset
        print("\nPlease Input Requested File Path (Make sure that the file you wish to load is in the current directory, and is c
        filename=input()
        path=filename
        #Error Checking whether dataset exists
        if(exists(path)):
            break
        else:
            print("\033[91m \n\nError: Requested File Not Found \033[0m")

#Loading the data from a specified path, and storing csv contents in a dataframe, with correct Error Handling
if(exists(path)):
    dataframe = pd.read_csv(path)
    #Error Checking whether dataframe has the relevant number of columns
    if(len(dataframe.columns)<3):
        print("\033[91m \n\nWarning: The Dataset has less than 3 features/columns, and would present Errors later on in forthcomi
else:
    print("\033[91m \n\nError: Requested File Not Found \033[0m")
```

Figure 2: Code required for loading the chosen dataset into a pandas data frame.

2.3 Dataset Feature Selection

Another key step when performing a data analysis or a machine learning study, pertains to observing the type and number of different Genes/Features, which the dataset possesses. This step is highly critical, as sometimes processing a huge number of features in the dataset may cause memory allocation issues or prolong the processing time of algorithms. Consequently, in this stage students are given the option to choose which features to retain from the

dataset, through the form of a user input menu. Additionally, in the eventuality that the student's selection represents two columns or less, the program will automatically choose the first three columns which will be added to the filtered dataset. This was applied as a fail-safe measure, to ensure that the filtered dataset, would have enough features for visualisation in the upcoming sections.

The code snippet illustrated in Figure 3 portrays the aforementioned functionality, whereby the program first creates a new copy of original data frame, and then proceeds to loop through all of the new data frame's columns. For each column iteration, the program probes the user with the choice of keeping the current column or discarding it. In case that the user opts to drop the current column, the pd.drop method is actioned in removing such feature from the new data frame. At the end of such snippet, a while loop is being used to ensure that the new data frame has at least three columns. Notably, for every iteration in the while loop, the pd.insert method is being used to add a column from the old data frame.

```
#Dataframe which will hold the selected features
filteredDataframe=dataframe
#Displaying Menu to the User, so that the user will choose which features to keep
print("\033[1mChoose whether to keep the following Genes/Features, to work with: \033[0m ")
for colNum,column in enumerate(dataframe.columns):
    print("Do you wish to keep: ",colNum+1,"\b.",column,"? \033[1m(Enter 'Y' to accept, and 'N' to decline)\033[0m")
    choice=input()
    if(choice!="Y"):
        filteredDataframe=filteredDataframe.drop(columns=[column])

#Looping until size is smaller than 3 and adding the first 3 columns to the dataframe (Error Checking)
count=0
while(len(filteredDataframe.columns)<3):
    filteredDataframe.insert(count,dataframe.columns[count],dataframe.iloc[:,0])
    count+=1
```

Figure 3: Code required for feature selection.

2.4 Dealing with Discrete Data

As previously mentioned, PCA is designed to be utilised on continuous data [13]. This is a cardinal feature and dictates the need to transform discrete data into continuous data before using PCA on a dataset. This is critical, as discrete data lacks a continuous range of values and cannot be represented in the same way as continuous data for this cause. There are various ways how discrete data can be transformed/encoded to continuous data, in order to be examined by the PCA. Nevertheless, students might prefer to discard such columns, and focus only on the continuous data, they could opt to remove such discrete columns from the data frame in the previous section. Additionally proper error checking was implemented, in case that the data frame does not include any discrete columns, thus ensuring the robustness of the developed solution.

The following are different types of encoders, which were implemented in the artefact:

(1) **One-Hot Encoding**
(2) **Label Encoding**
(3) **Ordinal Encoding**
(4) **Count Encoding**
(5) **Word Embeddings Model**

2.4.1 One-Hot Encoding. One-Hot encoding is a data preparation technique used to transform discrete variables into a format that enables the examination by machine learning algorithms. Consequently, this encoding algorithm works by creating a binary vector for each possible category in the data. Additionally, each binary vector would have a value of 1 or 0 symbolising the presence or absence of each category respectively. [14-16]

The encoding technique featured in the program, is quite explosive, as the number of different features obtained after applying One-Hot encoding on a single column, will greatly increase the number of columns proportional to the number of distinct features in each column. For an algorithm which aims to reduce dimensionality, such approach to turn discrete data into continuous data is quite inefficient, notwithstanding the increase in memory and time complexity.

One might think whether this binary vector can be transformed back to decimal. Note that such encoding algorithm exists and is known as *binary to decimal decoding*. The aforementioned algorithm effectively transforms the binary vector back into a decimal value, thus reducing the size of the features to their original number [17]. Essentially such encoding would take relatively more time whilst achieving the same results as label encoding or ordinal encoding.

In this section students are presented with an application of such encoding algorithm on the filtered data frame, and given a detailed explanation, justifying the inefficiency of the combined use of One-Hot encoding with the PCA algorithm. As illustrated in the code snippet of Figure 4, the program loops through all the filtered data frame columns and proceeds to check whether the current column has an Object type i.e., is a discrete column. In case the current column has such type, then the pd.get_dummies function is applied to such column, and the result is stored inside a new data frame, whilst exiting the loop. Following this, the user is presented with a message displaying the difference in size between the encoded column and the original data frame.

```
#Looping through all the columns in the data frame and checking whether column has object type (i.e., contains discrete data),
#if so, applying one hot encoding on the first discrete column, and exiting
oneHotEncDataframe=pd.DataFrame()
columnName=None
for column in filteredDataframe.columns:
    if filteredDataframe[column].dtype=='O':#0 denotes type Object
        oneHotEncDataframe=pd.get_dummies(filteredDataframe[column])
        columnName=column
        break
#Showing one-hot encoding dataframe of first discrete column
display(oneHotEncDataframe)
```

Figure 4: Code required for applying One-Hot encoding on the first discrete column.

2.4.2 Label Encoding. Label encoding is another data preparation technique which facilitates the transformation of discrete variables into a format that is easily readable by machine learning algorithms. Such encoder works by giving each distinct category a unique numeric value or code [14,15,18]. For instance, taking the list of categories ["hat","apple","cap"], these will be encoded as "[3,1,2]" (as numeric values).

As can be depicted in code snippet of Figure 5, the program first creates a new copy of filtered data frame, and then proceeds to loop through all of the new data frame's columns. For every column iteration, the program checks whether the current column has an Object type. In the affirmative, the pd.factorise function is used to apply label encoding on such column. Furthermore, for this encoding implementation the sort flag was set to True in the pd.factorise function, as to enable the transformed data to be assigned numerical values based on the sorted strings.

```
#Looping through every colum, and applying the pd.factorise function with "sort=True" on the discrete data
labelEncDataframe=filteredDataframe.copy()
for column in labelEncDataframe.columns:
    if labelEncDataframe[column].dtype=='O':#O denotes type Object
        labelEncDataframe[column] = pd.factorize(filteredDataframe[column], sort=True)[0]
#Displaying Label Encoding DataFrame
display(labelEncDataframe)
```

Figure 5: Code required for applying label encoding on the filtered data frame.

2.4.3 Ordinal Encoding. A similar data preparation technique to label encoding is ordinal encoding. Such encoder works by giving each distinct category a unique numeric value or code, based on the order which the category first appeared [14,15,19]. For instance, taking the list of categories ["hat","apple","cap"], these will be encoded as "[1,2,3]" (as numeric values, and encoded in the order which they appeared).

As can be seen in the code snippet of Figure 6, ordinal encoding is being implemented similarly to the label encoding implementation depicted in Figure 5. Notably the difference between both figures is clearly denoted by the pd.factorise function used to apply ordinal encoding, which has the sort flag applied to False. This was intentionally programmed in such manner so as to facilitate the assignment of numerical values based on the order in which the words appeared. Through the implementation of label encoding and ordinal encoding, which utilise the same function with some minor tweaks, students are given further opportunities to test out different encoding techniques, on the same dataset.

```
#Looping through every colum, and applying the pd.factorise function on the discrete data
ordinalEncDataframe=filteredDataframe.copy()
for column in ordinalEncDataframe.columns:
    if ordinalEncDataframe[column].dtype=='O':#O denotes type Object
        ordinalEncDataframe[column] = pd.factorize(filteredDataframe[column])[0]
#Displaying Ordinal Encoding DataFrame
display(ordinalEncDataframe)
```

Figure 6: Code required for applying ordinal encoding on the filtered data frame.

2.4.4 Count Encoding. Another data preparation technique is count encoding. This encoder works by encoding each distinct category, with the number of times such category appeared [14-16]. For instance, if the category "hat" appeared 5 times, then "hat" will be encoded by the number 5.

The code snippet seen in Figure 7, depicts the implementation of the aforementioned count encoding technique, which utilises the same logic of realising discrete columns as the previously mentioned encoding techniques. Furthermore, the value_counts function, is being used to count the number of times, each discrete value appears in the current discrete column. Afterwards, the pd.map function is being used to map the acquired array of discrete value frequencies to the current column's discrete values, ultimately transforming the discrete values to the number of times which they have appeared in the discrete column.

2.4.5 Word Embeddings Model. A word embeddings model is a type of *natural language processing (NLP)* model which depict words as numerical vectors in a high-dimensional space. This model works by first training a neural network on a large corpus of text data, in order to represent words as dense, low-dimensional vectors. Each component of the word vector represents a specific aspect or characteristic of the word, such as its semantic meaning, part of speech,

```
#Looping through every colum, and applying the .value_counts and .map functions on the discrete data
countEncDataframe=filteredDataframe.copy()
for column in countEncDataframe.columns:
    if countEncDataframe[column].dtype=='O':#O denotes type Object
        colValueFreq=countEncDataframe[column].value_counts(dropna=False)
        countEncDataframe[column] =countEncDataframe[column].map(lambda x : colValueFreq[x])
#Displaying Count Encoding DataFrame
display(countEncDataframe)
```

Figure 7: Code required for applying count encoding on the filtered data frame.

or syntactic context [20]. Mainly, the developed artefact focuses on the use of *Word2vec*, this being one type of word embeddings model [20].

The code snippet at Figure 8, illustrates the implementation of the word embeddings model encoding technique, which utilises the same logic of realising discrete columns as the previously mentioned encoding techniques. Moreover, such technique was implemented through the use of the Word2vec and nltk word_tokenize libraries. In order to do so the program first proceeds to tokenise all the strings in the discrete column through the word_tokenize library, before saving the tokens in an array. Next the tokens are fed to the Word2Vec model with the min_count parameter set to 1, intentionally to capture words with a count of 1. Consequently, the resulting word vector obtained in the previous step is used in conjunction with the df.apply and np.mean methods, to assign a relevant value to the discrete values in the current column. Moreover, through this step, the resultant nan values from the word embeddings model classification, are transformed to 0 so as to avoid the featuring of errors later on. In view that the aforementioned process may take some time to attain completion, users are presented with a message highlighting the current column being processed, which in a way depicts the progress to the algorithm.

```
#Looping through every colum, and training the Word2Vec model on thge discrete data
wordEncDataframe=filteredDataframe.copy()
for column in wordEncDataframe.columns:
    if wordEncDataframe[column].dtype=='O':#O denotes type Object
        print("\033[1mCompleting Column:\033[0m",column)
        #Tokenising current Column
        tokens=wordEncDataframe[column].apply(lambda x: word_tokenize(str(x).lower()))
        #Feeding the model the tokens
        #min count refers to the number of words to consider, a count of 1 means
        #we are considering words with a count of 1
        wordEmbeddingsModel=Word2Vec(tokens,min_count=1)

        #Retrieving the mean of the vector of all the current tokens in the sentence, and checking that token is not nan
        wordEncDataframe[column] = wordEncDataframe[column].apply(lambda x:
            np.mean([wordEmbeddingsModel.wv[token] for token in word_tokenize(str(x).lower())], axis=0).tolist()[0]
            if str(x) != 'nan' else 0)
```

Figure 8: Code required for applying word embeddings model on the filtered data frame.

Intentionally, all of the aforementioned encoding techniques were designed with the sole purpose of enabling students to familiarise themselves with different encoding techniques. Moreover, the large stack of encoding techniques used, also provides students with the liberty to test out the different techniques on different datasets whilst, enabling them to identify the optimal encoding technique which would outperform the others for a specific dataset. Additionally, throughout the detailed explanation of all of the aforementioned encoding techniques which are highlighted in this section, students are also presented with visual representations of the resultant transformed data in the form of various data frames. After observing the results obtained from the various techniques, students are given the option of choosing which encoding technique to utilise in the prevailing sections of this project. The choice can be seen through the code snippet presented in Figure 9, whereby by default the label encoding technique was adopted.

```
print("\n\033[1mChoose which Encoding technique to utilise:\n1. Label Encoding\n2. Ordinal Encoding\n3. Count Encoding \n4. Word
userChoice=int(input())
filteredContinousData=labelEncDataframe
if(userChoice==2):
    filteredContinousData=ordinalEncDataframe
elif(userChoice==3):
    filteredContinousData=countEncDataframe
elif(userChoice==4):
    filteredContinousData=wordEncDataframe
```

Figure 9: Student choice for the encoding technique to utilise.

2.5 Filtered Dataset Visualisations

Visualisation is a useful tool, as it aids in the process of identifying data visual patterns and characteristics. It is a common fact that individuals find it simpler to spot patterns and trends when data is presented visually, rather than in numerical or written form. Unfortunately, not all features can be projected on screen, as visualised data is limited to three dimensions. Thus, individuals need to choose which features to visualise from a high-dimensional dataset with many features.

In this section, besides being presented with a list of features in the filtered dataset, students are given the option to choose their preferred features to plot. As illustrated in Figure 10, these will form the bases for the different dimensional visualisations in the forthcoming sections.

```
#Displaying list of features
print("\n\033[1mChoose 3 Features to represent the Data in 2D and 3D \033[0m")
for colNum,column in enumerate(filteredContinousData.columns):
    print(colNum,"\b.",column,)
#List which holds the valid Column ranges
validColumnRanges=range(0,len(filteredContinousData.columns))
#Variable Initialisation to null
x=None
y=None
z=None

#Looping until valid column index is entered for every Dimensional variable
while x not in validColumnRanges:
    x=int(input("\nPlease Input a Valid Column Index to represent the \033[1mX axis\033[0m attribute in the Plot:"))

while y not in validColumnRanges:
    y=int(input("\nPlease Input a Valid Column Index to represent the \033[1mY axis\033[0m attribute in the Plot:"))

while z not in validColumnRanges:
    z=int(input("\nPlease Input a Valid Column Index to represent the \033[1mZ axis\033[0m attribute in the Plot:"))
```

Figure 10: Student choice for the features to utilise for visualisation.

Principally, the plotly library was utilised in order to provide interactive plots and visualisations, allowing students to zoom in or out of the plots, whilst enabling them to recognize better certain data trends. The colour components utilised in the different visualisation, are not particularly linked to the variables, but serve as markers which compare graph axis between 2D and 3D plots. Unfortunately, the plotly library limits the number of maximum graphs to 10, For this reason some graphs would need to rerun the code cell in order to appear. Figure 11 depicts the code required to visualise a 3D plot.

```
#Creating a 3D scatter plot, from the respective user inputted columns
fig3D = go.Figure(data=go.Scatter3d(
    x=filteredContinousData.iloc[:,x],
    y=filteredContinousData.iloc[:,y],
    z=filteredContinousData.iloc[:,z],
    mode='markers',
    marker=dict(color = filteredContinousData.iloc[:,x],line=dict(width=2000, color='DarkSlateGrey'))
))

#Setting the respective title preferences
fig3D.update_layout(width=1000, height=1000, title = '3D Representation of Filtered DataSet from '+filename+':',
                    font_color="blue",font_family="verdana",
                    scene = dict(xaxis=dict(title=filteredContinousData.columns[x], titlefont_color='blue'),
                                 yaxis=dict(title=filteredContinousData.columns[y], titlefont_color='blue'),
                                 zaxis=dict(title=filteredContinousData.columns[z], titlefont_color='blue')))
#Displaying plot
fig3D.show()
```

Figure 11: Code required to visualise a 3D plot.

2.6 Normalizing Data

Normalization is the process of converting and scaling the numerical characteristics inside a dataset, with the aim of ensuring that the data is characterized by a uniform range and distribution. Normalization's primary objective is to guarantee that no feature dominates or has an excessively large impact on the model's performance. This process is critical in the calculation of the PCA, since if given unnormalized data, the PCA algorithm will load on the high variance data [5]. An example would be having two data variables, one having a value of 1 and the other having a value of 700, whereby the PCA algorithm will issue higher importance to the second value. The importance of normalizing data is highly significant especially when one considers that the previous encoding techniques, such as label encoding or ordinal encoding will provide transformed variables with an uneven distribution. Normalization addresses this issue once it converts the data values into a uniform range.

The developed implementation focuses on utilising **z-score normalization** or also known as **standardization** [21]. This normalization technique can be constructed through the formula presented in Figure 12.

$$norm = \frac{(x-\mu)}{\sigma}$$

where:

1. **x** - is the Original Value
2. μ - is the Mean of the Data
3. σ - is the Standard Deviation of the Data
4. **norm** - is the Normalized Data

Figure 12: z-score normalization Formula.

Additionally, the formula used to calculate **standard deviation** (σ) can be seen in Figure 13.

$$\sigma = \sqrt{\frac{\sum(x_i-\mu)^2}{N}}$$

where:

1. x_i - is a Single Value from the Whole Data
2. μ - is the Mean of the Data
3. **N** - is the Size of the Data
4. σ - is the Standard Deviation of the Data

Figure 13: standard deviation formula.

Figure 14 illustrates the implementation of the z-score normalization, whereby such method (NormalizeDF) utilises the mean() and std() functions for the calculation of the mean and standard deviation for each column respectively. Moreover, the formula presented in Figure 12 is applied on the current data frame, whereby every element in the data frame is subtracted by the mean, before being divided by the standard deviation. In the case that the standard deviation results display is a 0 figure, this automatically, changes the current value in the data frame to 0. The use of df.replace function, will trigger a process whereby nans are replaced by 0. This avoids that the program stops abruptly later on, whilst safeguarding against the loss of data.

2.7 Normalized Dataset Visualisations

The next step entails the plotting through the plotly library the normalized data frame attained from the previous sections, whilst

```python
#Function to Normalize a Dataframe
def NormalizeDF(inputDF):
    #Calculating the Mean for each Column through the .mean() function, and storing result in a list
    colMean=list(inputDF.mean())
    #Calculating the Standard Deviation for each Column through the .std() function, and storing result in a list
    colStandardDev=list(inputDF.std())
    #Dictionary which will hold the normalized data
    normalizedData=dict()
    counter=0
    #Looping through all the columns in the DataSet
    for column in inputDF:
        #Dictionary which will hold the normalized Column
        normalizedColumn=dict()
        rowCounter=0
        #Looping through every row in the current column
        for row in inputDF[column]:
            #Checking whether Standard Deviation is not 0, and if so normalizing current value
            if colStandardDev[counter]!=0:
                normalizedColumn[rowCounter]=(row-colMean[counter])/colStandardDev[counter]
            else:#Standard Deviation is 0, therefore setting value to 0
                normalizedColumn[rowCounter]=0
            #Incrementing row counter
            rowCounter+=1
        #appending normalized column to normalized data
        normalizedData[column]=normalizedColumn
        counter+=1

    #Converting data into pandas DataFrame
    normalizedDF=pd.DataFrame.from_dict(normalizedData)
    #Changing nan to 0
    normalizedDF=normalizedDF.replace(np.nan,0)
    #Returning normalized Dataframe
    return normalizedDF
```

Figure 14: z-score normalization implementation.

retaining the same features chosen by the student. This has the added benefit of enabling the student to draw up comparisons between graphs. Through comparisons with the original dataset visualisation and the normalized data visualisation, one might note that the normalized data is plotted on a smaller range of values, when compared to the original plots in the section above. Additionally, one can notice how in the normalized plots, the data values are centred around zero. With regards to the plots' colour points, there is a big tendency that the colour of the points in the normalized dataset plot changes from the ones displayed in the original plot. This is due to the fact that the data values are now centred around zero and the normalized plot is plotted on a smaller range of values.

2.8 Small note on Calculating PCA from first principles

For the following two PCA approaches, a small subset of the entire dataset is to be chosen. In case that the dataset has a larger size than a respective threshold, then the dataset is reduced to a tenth of its size, whilst maintaining the number of columns. This is of great assistance to the student to better understand the concept, and method of calculation. It is worth noting that the respective rows threshold is 10000 rows. Figure 15 illustrates the aforementioned process.

```python
#Taking a subset of the dataframe, normalizing it, and converting it to a numpy array, if the dataframe has a size more than 100
sizeThreshold=10000
smallerDF=NormalizeDF(filteredContinousData[:int(len(normalizedDF)/10)])
#Checking for the size
if(len(normalizedDF)<=sizeThreshold):
    smallerDF=NormalizeDF(filteredContinousData)
SmalledDFMatrix=smallerDF.to_numpy()
```

Figure 15: Taking a small subset of the dataset.

2.9 Understanding PCA - SVD Approach

The first step in Calculation of PCA via SVD Approach, was undertaken through the utilisation of the singular value decomposition (SVD), which is a decomposition method aimed at factorising a matrix of **m x n** size into three components. The resultant components include **U** and V^T, which are two orthonormal matrices, and Sigma (Σ) this being a diagonal matrix containing the singular values of the original matrix. Additionally, the size/magnitude of each singular value signifies the importance in explaining the data [22]. For example, a singular value of 10 will have a higher importance than a singular value of 5. Additionally, the calculation formula for SVD matrix decomposition, can be seen in Figure 16.

SVD Decomposition of matrix A with size m x n results in:

$$A = U.\Sigma.V^T$$

where:

1. **U** - is an Orthonormal Matrix of size **m x m**
2. **Σ** - is a Diagonal Matrix of size **m x n**
3. V^T - is an Orthonormal Matrix of size **n x n**

Figure 16: SVD calculation formula.

The second step in the calculation of PCA via SVD approach, involves multiplying the U matrix by the Σ matrix. This is performed as the multiplication of $U.\Sigma$ presents a matrix whose columns give the projections of the data points on each principal axis.

Subsequently the aforementioned algorithm was run through the NumPy library, in particular the svd() function, which upon a given a matrix as an input, the process would produce the required components. However, the Sigma matrix had to be developed, as the svd () function only returns the list of eigenvalues in descending order, which is far from the desired output matrix. Moreover construction of such matrix was facilitated through the np.zeros() and np.diag() functions respectively. The NumPy matrix multiplication (@) between the U and Sigma finally resulted in the creation of the required principal components. Figure 17 illustrates the aforesaid concept.

```python
#Calculating the svd, via np.linalg.svd function
U, s, VT = np.linalg.svd(SmalledDFMatrix)
#Constructing Sigma Matrix
Sigma = np.zeros((smallerDF.shape[0], smallerDF.shape[1]))
Sigma[:smallerDF.shape[1], :smallerDF.shape[0]] = np.diag(s)
#Multiplying U by Sigma to obtain Principal Axis
pcaSmallData1=U@Sigma
pd.DataFrame(pcaSmallData1)
```

Figure 17: Code required for PCA calculation via SVD approach.

2.10 Understanding PCA - Covariance Matrix Approach

Another approach which can be used to calculate PCA, is the covariance matrix method. Notably, the first step in the implementation of such approach pertains to the development of the covariance matrix or also known as the *covariance variance matrix*. The aforementioned matrix is a **n x n** symmetric matrix which is used to show the covariance values between adjacent pairs of items in a dataset of n attributes. Additionally, the diagonal elements of such matrix represent the variance of each element. [23]

The calculation formula for the covariance matrix, can be seen in Figure 18.

The principal components of this approach can be calculated through the multiplication of the normalised data frame with sorted eigenvectors of the covariance Matrix. The resultant will be a matrix whose columns give the projections of the data points on each principal axis.

Figure 19 illustrates the aforementioned process, through the input of the normalized data frame, whereby the covariance matrix

An example of a 3 x 3 Covariance Matrix for a dataset containing 3 variable x, y, z:

$$\begin{bmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{bmatrix}$$

Another notation for the Covariance Matrix above is as follows:

$$\begin{bmatrix} var(x) & cov(x,y) & cov(x,z) \\ cov(y,x) & var(y) & cov(y,z) \\ cov(z,x) & cov(z,y) & var(z) \end{bmatrix}$$

Calculation of Covariance for two variables x and y can be facilitate through the following Formula:

$$cov(x,y) = \frac{\sum(x_i-\bar{x})(y_i-\bar{y})}{N}$$

where:

1. $\bar{x}$ - is the mean value of the x attribute
2. $\bar{y}$ - is the mean value of the y attribute
3. x_i - is the current data value of the x attribute
4. y_i - is the current data value of the y attribute
5. N - is the number of data points

Figure 18: Covariance matrix calculation.

is calculated through the np.cov() function in the NumPy library. Additionally, the eigenvectors and eigenvalues of the covariance matrix, are computed through the NumPy.Linalg.eigh() function, and later sorted in descending order by eigenvalues. Finally, the principal components are drawn up by applying the NumPy matrix multiplication (@) between the normalized data frame and the sorted eigenvectors.

```
#Calculating covariance matrix through np.cov function
covMatrix=np.cov(SmalledDFMatrix.T)
pd.DataFrame(covMatrix)
#Computing eigen decomposition for covariance matrix
eigenvalues, eigenvectors = LA.eigh(covMatrix)
# Sort the eigenvectors by descending eigenvalues
sortKey = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sortKey]
eigenvectors = eigenvectors[:, sortKey]
#Multiplying the normalized Dataframe with the eigenvectors of the Covariance Matrix
pcaSmallData2=SmalledDFMatrix@eigenvectors
pd.DataFrame(pcaSmallData2)
#Ordering By EigenValues
sortIndices = np.argsort(eigenvalues)[::-1]
pcaSmallData2 = pcaSmallData2[:,sortIndices]

# Convert pcaSmallData2 to a pandas DataFrame and print it
pd.DataFrame(pcaSmallData2)
```

Figure 19: Code required for PCA calculation via covariance matrix approach.

2.11 Variance Ratio and Scree Plot

The use of *variance ratios* in PCA is carried out in order to compute the percentage of the overall variance in the data, associated with each principal component. Notably, each principal component in the PCA algorithm captures a specific amount of data variation. One can easily determine the percentage of the overall variation that each component contributes, by deriving the variance ratio. The variance ratio is a useful tool in identifying those principal components which are critical for explaining the variation in the data. [24]

The variance ratio is calculated through the following formula depicted in Figure 20 (obtained from [24-25]):

$$VarianceRatio = \frac{\lambda_i}{\sum\lambda_i}$$

where:

1. λ_i - is the sum of the squared distance for each of the principal component
2. $\sum\lambda_i$ - is the sum of all squared distances for all of the principal components

Figure 20: Variance ratio calculation.

The code snippet in Figure 21 depicts the calculation of the variance ratio. This method is initiated by the creation of an array which holds the square of each principal component. Subsequently, the program loops through all the principal components, whilst updating the square for each principal component for every iteration. The program then, proceeds to calculate the total sum of the calculated square values, whilst storing the result in a variable. Finally, the variance ratio is calculated by dividing the array of square values corresponding to each principal component with the total square value. Conclusively, the result is multiplied by 100, to derive a percentage value.

```
#Creating an array to hold the square of each principal component
square=[0]*len(pcaSmallData1[0])
#Looping through all the principal components, and calculating the square for each component
for row in range(len(pcaSmallData1)):
    for col in range(len(pcaSmallData1[row])):
        square[col]+=(pcaSmallData1[row][col])**2
#Calculating the total Square
totalSquare=sum(square)
#Calculating the Variance Ratio by dividing the square by the total square and multiplying by 100 to obtain a percentage
svdVarianceRatio=(square/totalSquare)*100
```

Figure 21: Code required for variance ratio calculation.

The *scree plot* is an important tool for displaying the calculated variance ratio. Through the use of such plot, one may easily visualise the variation percentage associated with each principal component. The variance ration displayed in the scree plot presents useful data for identifying a range of components to maximize the accuracy of the projected data, whilst minimizing data loss. [26]. Figure 22 illustrates a screen plot example.

The following methods can be used to determine the optimal number of principal components to retain [26]:

(1) **Elbow method** - This selective method is based on the premises of retaining all the principal components prior that to the curve plateaus in the Scree Plot. Moreover, this method derives the point on the Scree Plot where the curve plateaus, directly identifying those components before this point, which are ideal and thus should be maintained.
(2) **Kaiser rule** - This method of selection retains all the principal components with eigenvalues having at least a value of 1.
(3) **Proportion of variance plot** - This method bases its selection on desired percentage (%) amount of the variance. For example, one will take all principal components whose variance sum totals 60%.

2.12 Comparisons Between Approaches

Although both the SVD and covariance matrix approaches of computing the PCA algorithm essentially provide similar results, one

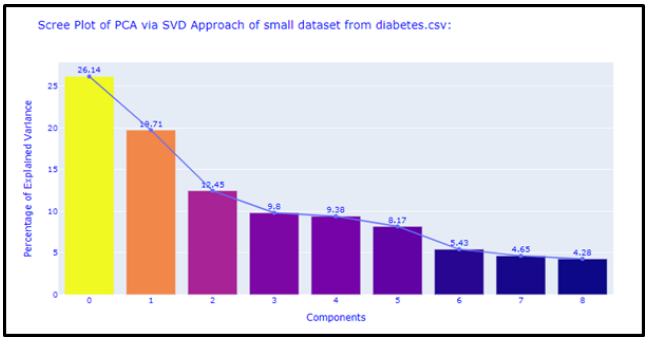


Figure 22: Scree plot example.

must acknowledge that both have their fair share of differences. For instance, in the SVD approach, computation of the principal components is done by directly applying SVD decomposition on the original matrix. In contrast in the covariance matrix approach, the covariance matrix needs to be computed, before applying eigen decomposition in order to compute the principal components. This makes the covariance matrix approach a lengthier process, which is more time consuming. Moreover, the covariance approach also tends to be quite memory inefficient, this is due to that fact that by its own nature this approach is structured on the computation of a large matrix, whereas the goal of the PCA is to reduce the dimensionality of data. Performance-wise, PCA with SVD not only outperforms PCA with covariance, but is often quicker, and has a higher numerical stability. However, in some circumstances, such as when the data includes missing values or when the data is not centred, PCA with covariance proves to be the ideal method. [22]

In this section students are presented with the different scree plots and 2D plots, for the respective approaches, displayed next to each other, in order to grant the student, the opportunity to visually compare the results obtained for each method. In some cases, the visualisation of both approaches may highlight discrepancies, depicting the graphs as inverted copies of each other. This is due to the difference in the direction of the eigenvectors. Figures 23 and 24 illustrate an example of such discrepancy.

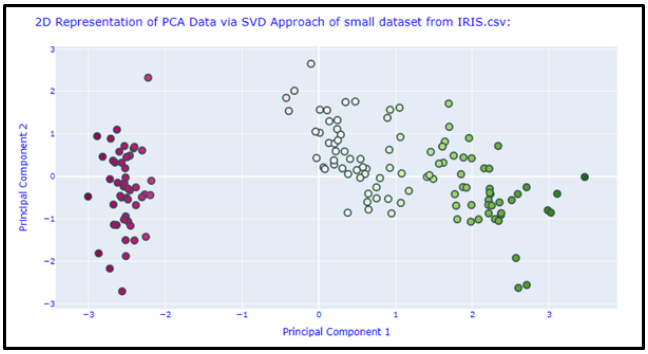


Figure 23: 2D plot of PCA utilising SVD approach.

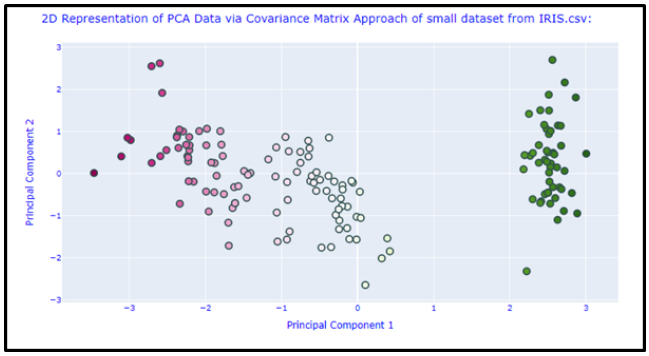


Figure 24: 2D plot of PCA utilising covariance matrix approach.

2.13 Working out PCA on the Entire Dataset

The PCA algorithm implementations in the previous sections, were undertaken with the sole purpose of educating students on how the algorithm functions. Nevertheless, utilisation of such algorithm does not require the lengthy implementation as outlined in the previous sections, as one can easily adopt to use the PCA function in the scikit-learn library through: "from sklearn.decomposition import PCA" import. The code snippet in Figure 25 clearly shows how the process could be compressed in a concise manner.

```
#Calling PCA algorithm from sklearn library, and feeding it the normalized Dataframe in its entirety
pca = PCA(n_components=len(normalizedDF.columns))
pca.fit(normalizedDF)
pcaData=pca.transform(normalizedDF)
#Transposing the matrix
pcaData=pcaData.transpose()
```

Figure 25: PCA algorithm through the scikit-learn library.

A discovery which arose through the creation of such notebook, entails that sometimes the calculation of PCA via the NumPy library would crash the notebook when running on large datasets, something which might be avoidable with the computation of PCA through the scikit-learn library. This feature is due to the fact that in the scikit-learn library, if the input dataset has a size larger than 500 x 500, and the number of components to extract is less than 80% of the smallest dimension of the data, then a randomized SVD proposed by Halko [4] is utilised [27]. Furthermore, utilisation of the NumPy library stores arrays in a contiguous block in memory, has a tendency to crash the notebook in the case that the computer has insufficient memory [28].

Notably, the randomized SVD proposed by Halko, can approximate the whole SVD with a substantially lower computation cost by randomly selecting a fraction of the matrix's rows or columns [4]. This is another attribute which highlights the superior efficiency of the scikit-learn approach, in comparison to the NumPy method.

2.14 PCA Visualisations

In this section students are presented with the relevant visualisations for the scree plot, 1D, 2D and 3D plots respectively. Furthermore, the required visualisations are plotted from the principal components obtained from applying the PCA algorithm of the

scikit-learn library, on the whole dataset. In addition, such visualisations are plotted through similar code illustrated in Figure 10, whilst keeping the same variable axis of the previous sections, with the aim of ensuring uniformity and permitting comparability.

2.15 Conclusions and Limitations of PCA

In this section students are presented with a brief summary of the concepts covered in the aforementioned notebook, whilst reflecting on the advantages and disadvantages inherent of the PCA algorithm. This provides a solid teaching ground for students to familiarize themselves with the different types of PCA algorithms, namely the kernel PCA and *robust PCA*. Notably, these two PCA techniques are fundamental for addressing some of the limitations presented by the PCA algorithm. Additionally, at the end of such section, students are also provided with a link [29] to access in-depth information about the different types of PCA, whilst also depicting the application of these types in python through the scikit-learn library.

3 EVALUATION

A run through of produced artefact can be seen in the following Figures 26-60, and this to verify that the project requirements have been met. The wine-quality-white-and-red.csv dataset, which signify a run through the whole notebook, was utilised.

Figures 26-27 display the user interface in the **Loading the Data** section presented in the notebook.

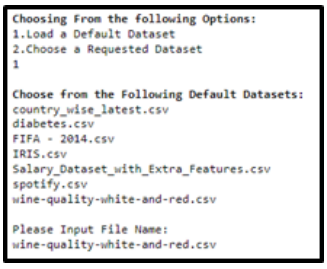
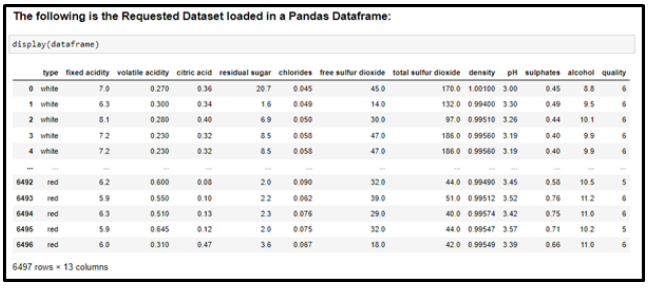


Figure 26: Showcasing the dataset selection menu.


The following is the Requested Dataset loaded in a Pandas Dataframe:

```
display(dataframe)
```

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	white	7.0	0.270	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.8	6
1	white	6.3	0.300	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.5	6
2	white	8.1	0.280	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.1	6
3	white	7.2	0.230	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
4	white	7.2	0.230	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...	...	...
6492	red	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
6493	red	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
6494	red	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
6495	red	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
6496	red	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

6497 rows × 13 columns

Figure 27: Displaying the contents of the requested dataset.

Figures 28-29 display the user interface in the **Dataset Feature Selection** section presented in the notebook.

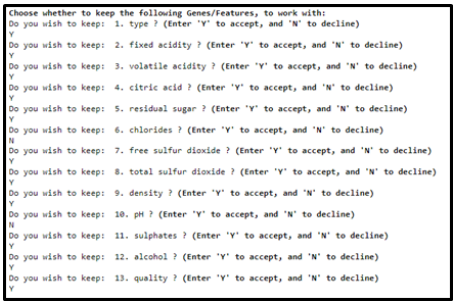
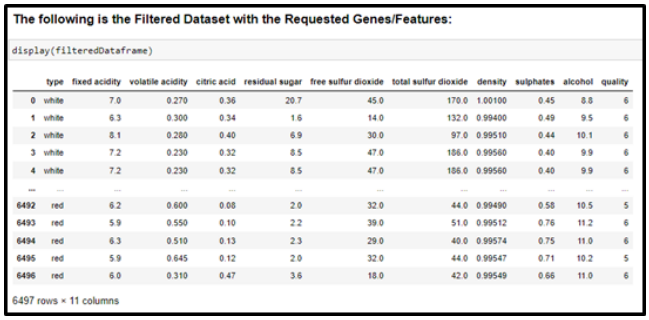


Figure 28: Showcasing the feature selection menu.


The following is the Filtered Dataset with the Requested Genes/Features:

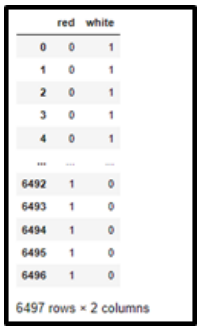
```
display(filteredDataframe)
```

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	white	7.0	0.270	0.36	20.7	45.0	170.0	1.00100	0.45	8.8	6
1	white	6.3	0.300	0.34	1.6	14.0	132.0	0.99400	0.49	9.5	6
2	white	8.1	0.280	0.40	6.9	30.0	97.0	0.99510	0.44	10.1	6
3	white	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
4	white	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...
6492	red	6.2	0.600	0.08	2.0	32.0	44.0	0.99490	0.58	10.5	5
6493	red	5.9	0.550	0.10	2.2	39.0	51.0	0.99512	0.76	11.2	6
6494	red	6.3	0.510	0.13	2.3	29.0	40.0	0.99574	0.75	11.0	6
6495	red	5.9	0.645	0.12	2.0	32.0	44.0	0.99547	0.71	10.2	5
6496	red	6.0	0.310	0.47	3.6	18.0	42.0	0.99549	0.66	11.0	6

6497 rows × 11 columns

Figure 29: Displaying the filtered dataset with the requested features.

Figures 30-36 display the user interface in **the Dealing with Discrete Data** section presented in the notebook.



	red	white
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1
...	...	...
6492	1	0
6493	1	0
6494	1	0
6495	1	0
6496	1	0

6497 rows × 2 columns

Figure 30: Displaying the result obtained after applying One-Hot encoding on the first discrete column.

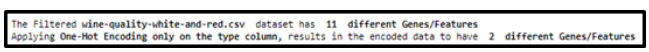


Figure 31: Displaying the comparison between the number of features between the filtered dataset and the One-Hot encoded Data.

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	1	7.0	0.270	0.36	20.7	45.0	170.0	1.00100	0.45	8.8	6
1	1	6.3	0.300	0.34	1.6	14.0	132.0	0.99400	0.49	9.5	6
2	1	8.1	0.280	0.40	6.9	30.0	97.0	0.99510	0.44	10.1	6
3	1	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
4	1	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...
6492	0	6.2	0.600	0.08	2.0	32.0	44.0	0.99490	0.58	10.5	5
6493	0	5.9	0.550	0.10	2.2	39.0	51.0	0.99512	0.76	11.2	6
6494	0	6.3	0.510	0.13	2.3	29.0	40.0	0.99574	0.75	11.0	6
6495	0	5.9	0.645	0.12	2.0	32.0	44.0	0.99547	0.71	10.2	5
6496	0	6.0	0.310	0.47	3.6	18.0	42.0	0.99549	0.66	11.0	6

6497 rows × 11 columns

Figure 32: Displaying the result obtained after applying label encoding.

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	0	7.0	0.270	0.36	20.7	45.0	170.0	1.00100	0.45	8.8	6
1	0	6.3	0.300	0.34	1.6	14.0	132.0	0.99400	0.49	9.5	6
2	0	8.1	0.280	0.40	6.9	30.0	97.0	0.99510	0.44	10.1	6
3	0	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
4	0	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...
6492	1	6.2	0.600	0.08	2.0	32.0	44.0	0.99490	0.58	10.5	5
6493	1	5.9	0.550	0.10	2.2	39.0	51.0	0.99512	0.76	11.2	6
6494	1	6.3	0.510	0.13	2.3	29.0	40.0	0.99574	0.75	11.0	6
6495	1	5.9	0.645	0.12	2.0	32.0	44.0	0.99547	0.71	10.2	5
6496	1	6.0	0.310	0.47	3.6	18.0	42.0	0.99549	0.66	11.0	6

6497 rows × 11 columns

Figure 33: Displaying the result obtained after applying ordinal encoding.

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	4898	7.0	0.270	0.36	20.7	45.0	170.0	1.00100	0.45	8.8	6
1	4898	6.3	0.300	0.34	1.6	14.0	132.0	0.99400	0.49	9.5	6
2	4898	8.1	0.280	0.40	6.9	30.0	97.0	0.99510	0.44	10.1	6
3	4898	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
4	4898	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...
6492	1599	6.2	0.600	0.08	2.0	32.0	44.0	0.99490	0.58	10.5	5
6493	1599	5.9	0.550	0.10	2.2	39.0	51.0	0.99512	0.76	11.2	6
6494	1599	6.3	0.510	0.13	2.3	29.0	40.0	0.99574	0.75	11.0	6
6495	1599	5.9	0.645	0.12	2.0	32.0	44.0	0.99547	0.71	10.2	5
6496	1599	6.0	0.310	0.47	3.6	18.0	42.0	0.99549	0.66	11.0	6

6497 rows × 11 columns

Figure 34: Displaying the result obtained after applying count encoding.

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	-0.000536	7.0	0.270	0.36	20.7	45.0	170.0	1.00100	0.45	8.8	6
1	-0.000536	6.3	0.300	0.34	1.6	14.0	132.0	0.99400	0.49	9.5	6
2	-0.000536	8.1	0.280	0.40	6.9	30.0	97.0	0.99510	0.44	10.1	6
3	-0.000536	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
4	-0.000536	7.2	0.230	0.32	8.5	47.0	186.0	0.99560	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...
6492	-0.008620	6.2	0.600	0.08	2.0	32.0	44.0	0.99490	0.58	10.5	5
6493	-0.008620	5.9	0.550	0.10	2.2	39.0	51.0	0.99512	0.76	11.2	6
6494	-0.008620	6.3	0.510	0.13	2.3	29.0	40.0	0.99574	0.75	11.0	6
6495	-0.008620	5.9	0.645	0.12	2.0	32.0	44.0	0.99547	0.71	10.2	5
6496	-0.008620	6.0	0.310	0.47	3.6	18.0	42.0	0.99549	0.66	11.0	6

6497 rows × 11 columns

Figure 35: Displaying the result obtained after utilising the word embeddings model.

```
Choose which Encoding technique to utilise:
1. Label Encoding
2. Ordinal Encoding
3. Count Encoding
4. Word Embeddings Model
2
```

Figure 36: Showcasing the encoding technique selection menu (in this case ordinal encoding was selected).

Figures 37-39 display the user interface in the **Filtered Dataset Visualisations** section presented in the notebook.

```
Choose 3 Features to represent the Data in 2D and 3D
0. type
1. fixed acidity
2. volatile acidity
3. citric acid
4. residual sugar
5. free sulfur dioxide
6. total sulfur dioxide
7. density
8. sulphates
9. alcohol
10. quality

Please Input a Valid Column Index to represent the X axis attribute in the Plot:2

Please Input a Valid Column Index to represent the Y axis attribute in the Plot:3

Please Input a Valid Column Index to represent the Z axis attribute in the Plot:4
```

Figure 37: Showcasing the feature visualisation selection menu (in this case features 2,3 and 4 were chosen as the features to be plotted on the respective axis).

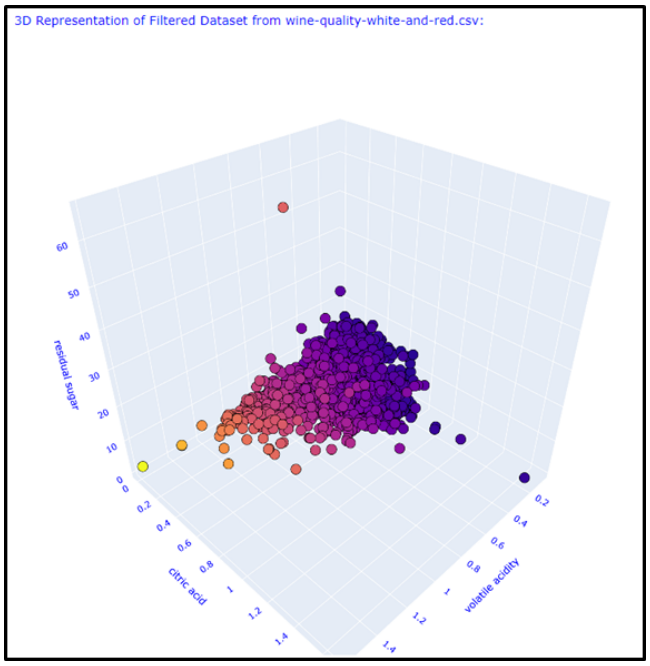


Figure 38: Displaying the 3D representation of the filtered dataset, with the chosen features.

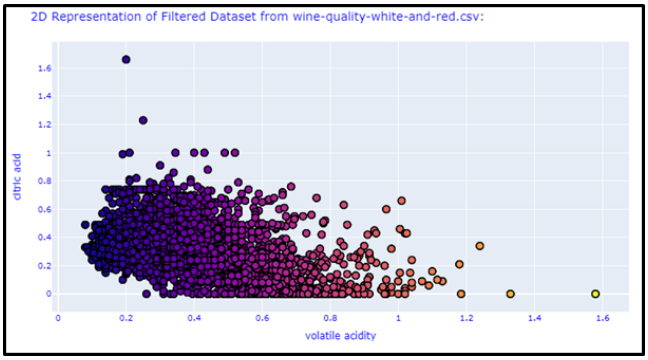


Figure 39: Displaying the 2D representation of the filtered dataset, with the chosen features.

Figures 40-42 display the user interface in the **Normalizing the Data** section presented in the notebook.

```
Column:  1      Mean:  0.2461135908881022
Column:  2      Mean:  7.215307864799138
Column:  3      Mean:  0.3396659996921656
Column:  4      Mean:  0.3186332153301524
Column:  5      Mean:  5.443235339387489
Column:  6      Mean:  30.52533937817544
Column:  7      Mean:  115.7445744189626
Column:  8      Mean:  0.9946966338309989
Column:  9      Mean:  0.5312682776666153
Column:  10     Mean:  10.491800831149455
Column:  11     Mean:  5.818377712790519
```

Figure 40: Displaying the calculated mean for each column.

```
Column:  1      Standard Deviation:  0.4307786597787076
Column:  2      Standard Deviation:  1.2964337577998155
Column:  3      Standard Deviation:  0.1646364748646784
Column:  4      Standard Deviation:  0.1453178648975915
Column:  5      Standard Deviation:  4.757803743147418
Column:  6      Standard Deviation:  17.7493997720025
Column:  7      Standard Deviation:  56.52185452263028
Column:  8      Standard Deviation:  0.0029867300379039
Column:  9      Standard Deviation:  0.1488058736144902
Column:  10     Standard Deviation:  1.1927117488868981
Column:  11     Standard Deviation:  0.8732552715311248
```

Figure 41: Displaying the calculated standard deviation for each column.

The following is the Normalized Dataframe:

```
display(normalizedDF)
```

	type	fixed acidity	volatile acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	sulphates	alcohol	quality
0	-0.571323	-0.166076	-0.423150	0.284664	3.206682	0.815503	0.959902	2.102052	-0.546136	-1.418449	0.207983
1	-0.571323	-0.706019	-0.240931	0.147035	-0.807775	-0.931035	0.287595	-0.232314	-0.277330	-0.831551	0.207983
2	-0.571323	0.682405	-0.362411	0.559923	0.306184	-0.029596	-0.331634	0.134515	-0.613338	-0.328496	0.207983
3	-0.571323	-0.011807	-0.666110	0.009405	0.642474	0.928182	1.242978	0.301255	-0.882144	-0.496181	0.207983
4	-0.571323	-0.011807	-0.666110	0.009405	0.642474	0.928182	1.242978	0.301255	-0.882144	-0.496181	0.207983
...	...	...	...	...	...	...	...	...	...	...	...
6492	1.750055	-0.783154	1.581266	-1.642146	-0.723703	0.083083	-1.269324	0.067819	0.327485	0.006874	-0.937157
6493	1.750055	-1.014558	1.277566	-1.504517	-0.681666	0.477463	-1.145479	0.141185	1.537115	0.593772	0.207983
6494	1.750055	-0.706019	1.034607	-1.298073	-0.660648	-0.085936	-1.340094	0.347943	1.469913	0.426087	0.207983
6495	1.750055	-1.014558	1.854595	-1.366888	-0.723703	0.083083	-1.269324	0.257903	1.201107	-0.244553	-0.937157
6496	1.750055	-0.937423	-0.180191	1.041625	-0.387413	-0.705676	-1.304709	0.264572	0.865098	0.426087	0.207983

6497 rows × 11 columns

Figure 42: Displaying the normalized data frame.

Figures 43-44 display the user interface in the **Normalized Dataset Visualisations** section presented in the notebook.

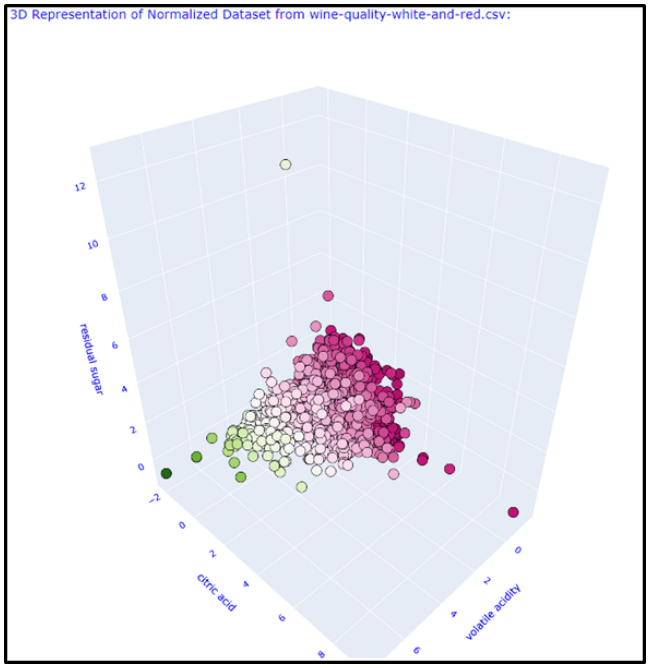


Figure 43: Displaying the 3D representation of the normalized dataset, with the chosen features.

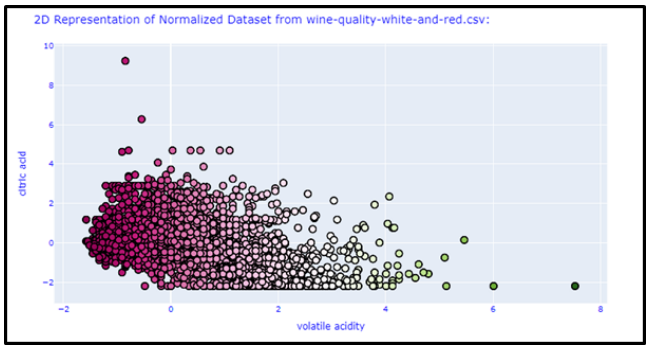
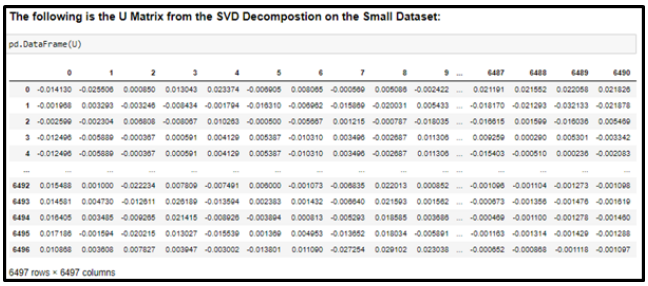


Figure 44: Displaying the 2D representation of the normalized dataset, with the chosen features.

Figures 45-51 display the user interface in the **Understanding PCA-SVD Approach** section presented in the notebook.

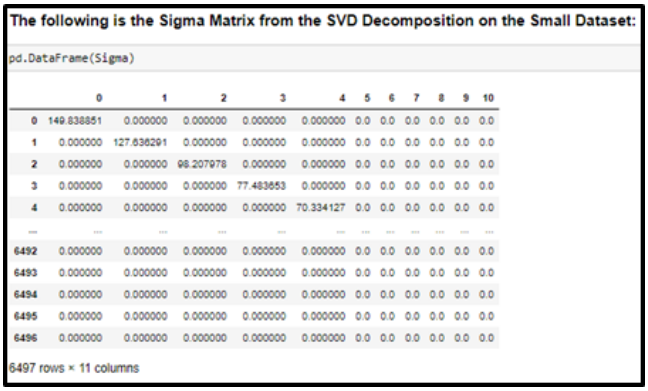

The following is the U Matrix from the SVD Decompostion on the Small Dataset:

```
pd.DataFrame(U)
```

	0	1	2	3	4	5	6	7	8	9	...	6487	6488	6489	6490
0	-0.014130	-0.025606	0.000850	0.013043	0.023374	-0.009905	0.008065	-0.000569	0.005086	-0.002422	...	0.021191	0.021552	0.022058	0.021826
1	-0.001968	0.003293	-0.003246	-0.008434	-0.001794	-0.016310	-0.006962	-0.015859	-0.020031	0.005433	...	-0.016170	-0.021293	-0.032133	-0.021878
2	-0.002599	-0.002304	0.005808	-0.008067	0.010263	-0.000500	-0.005667	0.001215	-0.000787	-0.018035	...	-0.016515	0.001599	-0.016036	0.005469
3	-0.012496	-0.005889	-0.000367	0.000591	0.004129	0.005387	-0.010310	0.003496	-0.002687	0.011306	...	0.009259	0.000290	0.005301	-0.003342
4	-0.012496	-0.005889	-0.000367	0.000591	0.004129	0.005387	-0.010310	0.003496	-0.002687	0.011306	...	-0.015403	-0.000510	0.000236	-0.002083
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
6492	0.015488	0.001000	-0.022234	0.007809	-0.007491	0.006000	-0.001073	-0.006835	0.022013	0.000852	...	-0.001096	-0.001104	-0.001273	-0.001096
6493	0.014581	0.004730	-0.012611	0.026189	-0.013594	0.002383	0.001432	-0.006640	0.021593	0.001562	...	-0.000673	-0.001356	-0.001476	-0.001619
6494	0.016405	0.003485	-0.009265	0.021415	-0.008926	-0.003894	0.000813	-0.005293	0.019585	0.003686	...	-0.000469	-0.001100	-0.001278	-0.001460
6495	0.017186	-0.001594	-0.020215	0.013027	-0.015539	0.001369	0.004953	-0.013652	0.018034	-0.005891	...	-0.001163	-0.001314	-0.001429	-0.001288
6496	0.010868	0.003608	0.007827	0.003947	-0.003002	-0.013801	0.011090	-0.027254	0.029102	0.023038	...	-0.000652	-0.000868	-0.001118	-0.001097

6497 rows × 6497 columns

Figure 45: Displaying the resultant U matrix obtained after the SVD decomposition.

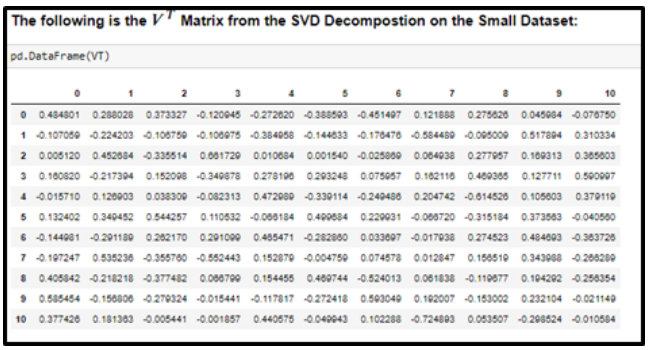

The following is the Sigma Matrix from the SVD Decomposition on the Small Dataset:

```
pd.DataFrame(Sigma)
```

	0	1	2	3	4	5	6	7	8	9	10
0	149.838851	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
1	0.000000	127.636291	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
2	0.000000	0.000000	98.207978	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
3	0.000000	0.000000	0.000000	77.483653	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
4	0.000000	0.000000	0.000000	0.000000	70.334127	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...
6492	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
6493	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
6494	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
6495	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
6496	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.0

6497 rows × 11 columns

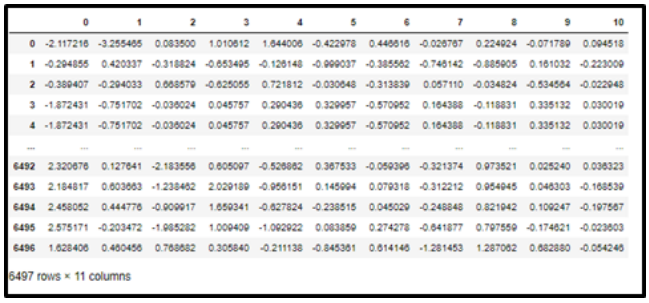
Figure 46: Displaying the resultant Sigma matrix obtained after the SVD decomposition.


The following is the V^T Matrix from the SVD Decompostion on the Small Dataset:

```
pd.DataFrame(VT)
```

	0	1	2	3	4	5	6	7	8	9	10
0	0.484801	0.288028	0.373327	-0.120945	-0.272620	-0.388593	-0.451497	0.121888	0.275626	0.045984	-0.076750
1	-0.107059	-0.224203	-0.106759	-0.106975	-0.384958	-0.144633	-0.176476	-0.584489	-0.095009	0.517894	0.310334
2	0.005120	0.452684	-0.335514	0.661729	0.010684	0.001540	-0.025869	0.064938	0.277957	0.169313	0.365603
3	0.160820	-0.217394	0.152098	-0.349878	0.278196	0.293248	0.075957	0.162116	0.469365	0.127711	0.590997
4	-0.015710	0.126903	0.038309	-0.082313	0.472989	-0.339114	-0.249486	0.204742	-0.614526	0.105603	0.379119
5	0.132402	0.349452	0.544257	0.110532	-0.066184	0.499684	0.229931	-0.069720	-0.315184	0.373563	-0.040560
6	-0.144981	-0.291189	0.262170	0.291099	0.465471	-0.282960	0.033697	-0.017938	0.274523	0.484693	-0.363726
7	-0.197247	0.535236	-0.355760	-0.552443	0.152879	-0.004759	0.074578	0.012847	0.156519	0.343988	-0.266289
8	0.405842	-0.218218	-0.377482	0.066799	0.154455	0.469744	-0.524013	0.061838	-0.119577	0.194292	-0.256354
9	0.585454	-0.156806	-0.279324	-0.015441	-0.117817	-0.272418	0.593049	0.192007	-0.153002	0.232104	-0.021149
10	0.377426	0.181363	-0.005441	-0.001857	0.440575	-0.049943	0.102288	-0.724893	0.053507	-0.298524	-0.010584

Figure 47: Displaying the resultant V transpose matrix obtained after the SVD decomposition.



	0	1	2	3	4	5	6	7	8	9	10
0	-2.117216	-3.255465	0.083500	1.010612	1.644006	-0.422978	0.446616	-0.026787	0.224924	-0.071789	0.094518
1	-0.294855	0.420337	-0.318824	-0.653495	-0.126148	-0.999037	-0.385562	-0.746142	-0.885905	0.161032	-0.223009
2	-0.389407	-0.294033	0.668579	-0.625055	0.721812	-0.030648	-0.313839	0.057110	-0.034824	-0.534564	-0.022948
3	-1.872431	-0.751702	-0.036024	0.045757	0.290436	0.329957	-0.570952	0.164388	-0.118831	0.335132	0.030019
4	-1.872431	-0.751702	-0.036024	0.045757	0.290436	0.329957	-0.570952	0.164388	-0.118831	0.335132	0.030019
...	...	...	...	...	...	...	...	...	...	...	...
6492	2.320676	0.127641	-2.183556	0.605097	-0.526862	0.367533	-0.059396	-0.321374	0.973521	0.025240	0.036323
6493	2.184817	0.603663	-1.238462	2.029189	-0.956151	0.145994	0.079318	-0.312212	0.954945	0.046303	-0.168539
6494	2.458052	0.444776	-0.909917	1.659341	-0.627824	-0.238515	0.045029	-0.248848	0.821942	0.109247	-0.197567
6495	2.575171	-0.203472	-1.985282	1.009409	-1.092922	0.083859	0.274278	-0.641877	0.797559	-0.174621	-0.023603
6496	1.628406	0.460456	0.768682	0.305840	-0.211138	-0.845361	0.614146	-1.281453	1.287062	0.682880	-0.054246

6497 rows × 11 columns

Figure 48: Displaying the principal components obtained through the matrix multiplication of U by Sigma.

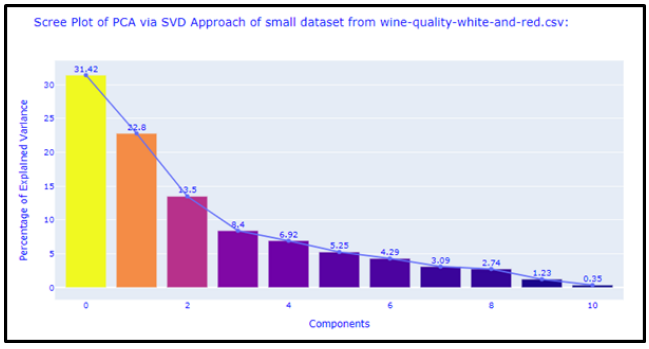


Figure 49: Displaying scree plot in the Understanding PCA - SVD Approach section.

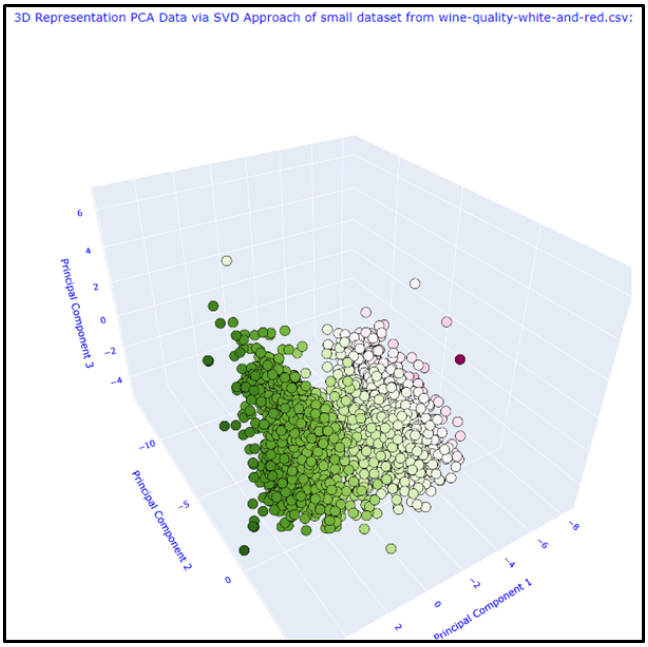


Figure 50: Displaying the 3D representation of the plotted principal components in the PCA - SVD approach section.

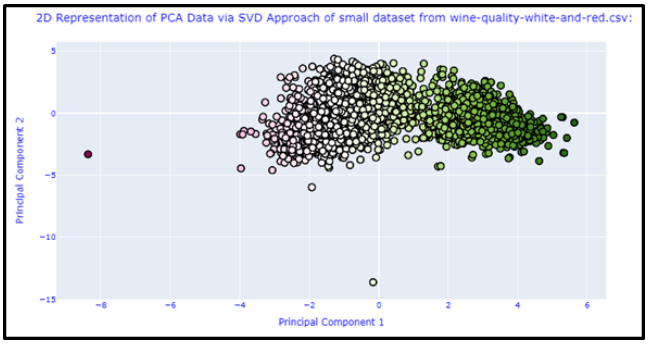


Figure 51: Displaying the 2D representation of the plotted principal components in the PCA - SVD approach section.

Figures 52-56 display the user interface in the **Understanding PCA-Covariance Matrix Approach** section presented in the notebook.

	0	1	2	3	4	5	6	7	8	9	10
0	1.000000	0.486740	0.653036	-0.187397	-0.348821	-0.471644	-0.700357	0.390645	0.487218	-0.032970	-0.119323
1	0.486740	1.000000	0.219008	0.324436	-0.111981	-0.282735	-0.329054	0.458910	0.299568	-0.095452	-0.076743
2	0.653036	0.219008	1.000000	-0.377981	-0.196011	-0.352557	-0.414476	0.271296	0.225984	-0.037640	-0.265699
3	-0.187397	0.324436	-0.377981	1.000000	0.142451	0.133126	0.195242	0.096154	0.056197	-0.010493	0.085532
4	-0.348821	-0.111981	-0.196011	0.142451	1.000000	0.402871	0.495482	0.552517	-0.185927	-0.359415	-0.036980
5	-0.471644	-0.282735	-0.352557	0.133126	0.402871	1.000000	0.720934	0.025717	-0.188457	-0.179838	0.055463
6	-0.700357	-0.329054	-0.414476	0.195242	0.495482	0.720934	1.000000	0.032395	-0.275727	-0.265740	-0.041385
7	0.390645	0.458910	0.271296	0.096154	0.552517	0.025717	0.032395	1.000000	0.259478	-0.686745	-0.305858
8	0.487218	0.299568	0.225984	0.056197	-0.185927	-0.188457	-0.275727	0.259478	1.000000	-0.003029	0.038485
9	-0.032970	-0.095452	-0.037640	-0.010493	-0.359415	-0.179838	-0.265740	-0.686745	-0.003029	1.000000	0.444319
10	-0.119323	-0.076743	-0.265699	0.085532	-0.036980	0.055463	-0.041385	-0.305858	0.038485	0.444319	1.000000

Figure 52: Displaying the calculated covariance matrix.

	0	1	2	3	4	5	6	7	8	9	10
0	2.117216	3.255465	0.083500	-1.010812	-1.644006	-0.422978	0.446616	-0.026767	0.224924	0.071789	-0.094518
1	0.294855	-0.420337	-0.318824	0.653495	0.126148	-0.999037	-0.385562	-0.746142	-0.885905	-0.161032	0.223009
2	0.389407	0.294033	0.668579	0.625055	-0.721812	-0.030648	-0.313839	0.057110	-0.034824	0.534564	0.022948
3	1.872431	0.751702	-0.036024	-0.045757	-0.290436	0.329957	-0.570952	0.164388	-0.118831	-0.335132	-0.030019
4	1.872431	0.751702	-0.036024	-0.045757	-0.290436	0.329957	-0.570952	0.164388	-0.118831	-0.335132	-0.030019
...	...	...	...	...	...	...	...	...	...	...	...
6492	-2.320676	-0.127641	-2.183556	-0.605097	0.526862	0.367533	-0.059396	-0.321374	0.973521	-0.025240	-0.036323
6493	-2.184817	-0.603663	-1.238462	-2.029189	0.956151	0.145994	0.079318	-0.312212	0.954945	-0.046303	0.168539
6494	-2.458052	-0.444776	-0.909917	-1.659341	0.627824	-0.238515	0.045029	-0.248848	0.821942	-0.109247	0.197567
6495	-2.575171	0.203472	-1.985282	-1.009409	1.092922	0.083859	0.274278	-0.641877	0.797559	0.174621	0.023603
6496	-1.628406	-0.460456	0.768682	-0.305840	0.211138	-0.845361	0.614146	-1.281453	1.287062	-0.682880	0.054246

6497 rows × 11 columns

Figure 53: Displaying the principal components obtained through the matrix multiplication of the normalized data frame with the sorted eigenvectors of the covariance matrix.

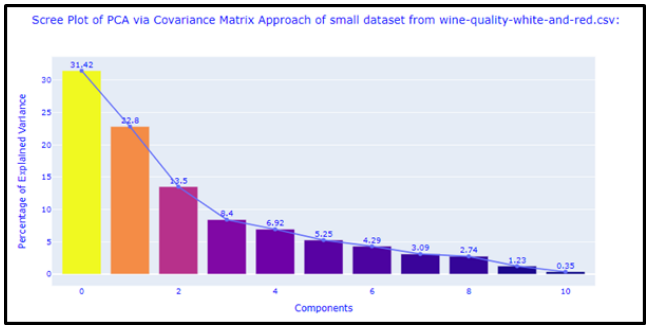


Figure 54: Displaying scree plot in the Understanding PCA – Covariance Matrix Approach section.

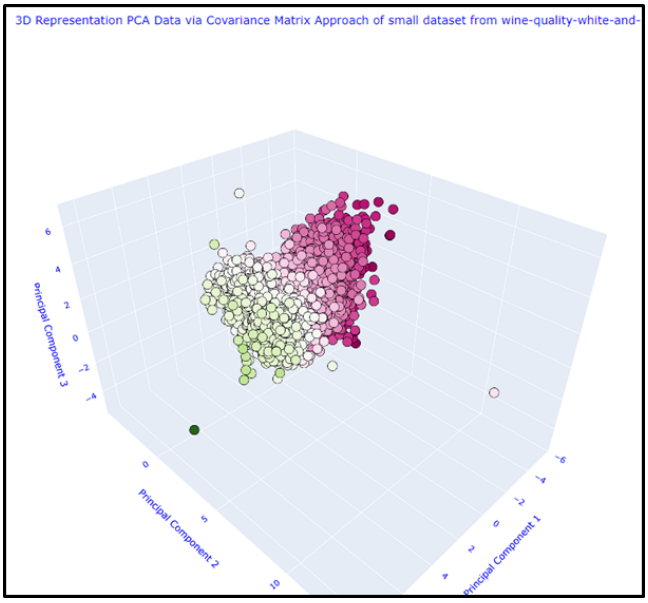


Figure 55: Displaying the 3D Representation of the plotted principal components in the Understanding PCA – Covariance Matrix Approach section.

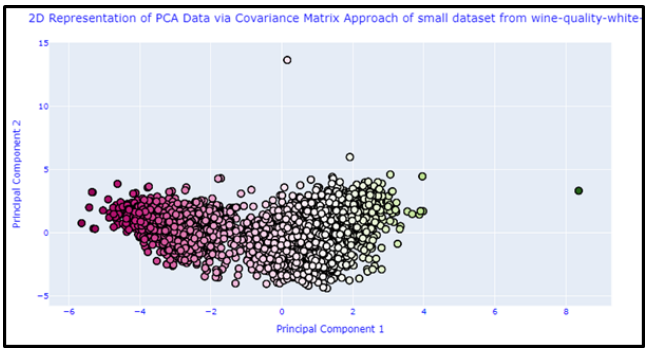


Figure 56: Displaying the 2D representation of the plotted principal components in the Understanding PCA – Covariance Matrix Approach section.

Figures 57-60 display the user interface in the **PCA Visualisations** section presented in the notebook. Note that the visualisations presented in the aforementioned figures utilise the calculations accomplished in the **Working out PCA on the Entire Dataset** section.

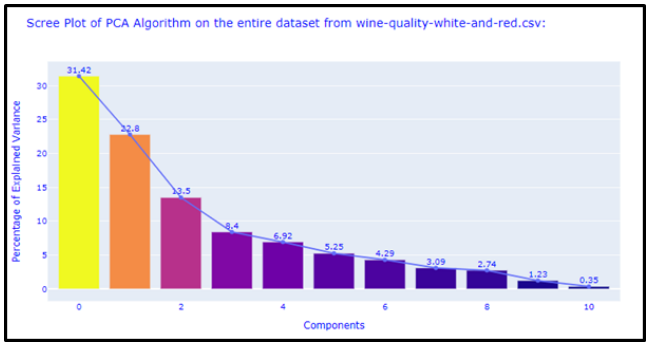


Figure 57: Displaying scree plot pertaining to the principal components obtained by utilising the entire dataset.

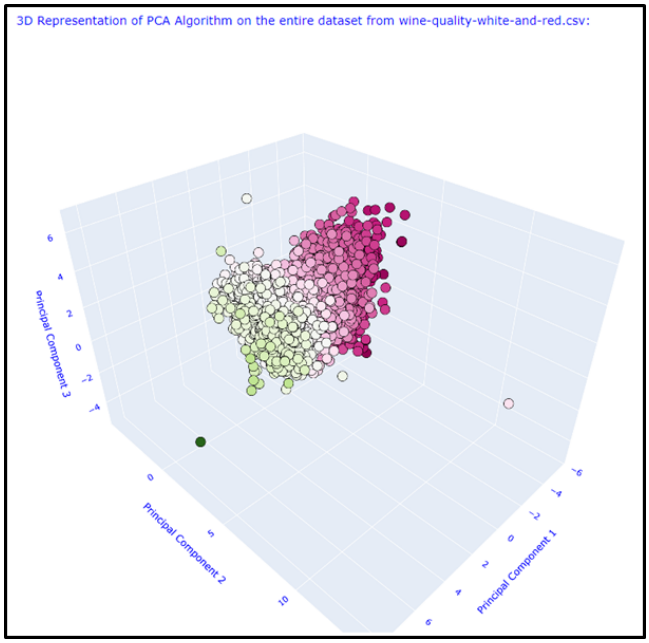


Figure 58: Displaying the 3D representation of the plotted principal components obtained by utilising the entire dataset.

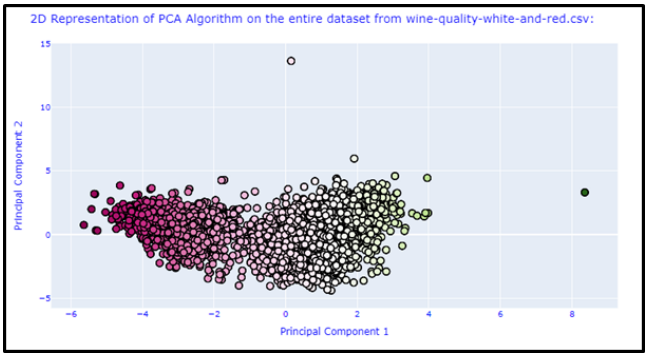


Figure 59: Displaying the 2D representation of the plotted principal components obtained by utilising the entire dataset.

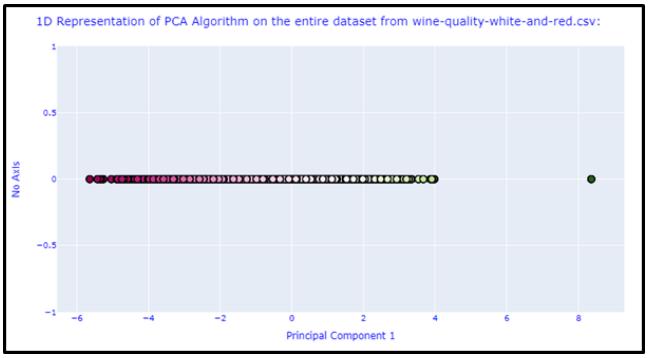


Figure 60: Displaying the 1D representation of the plotted principal components obtained by utilising the entire dataset.

4 CONCLUSION

The developed project explains the fundamentals of the principal component analysis as a dimensionality reduction tool, whilst presenting adequate and realistic applications of such algorithm. Throughout the course of this project, the different steps which compose the PCA algorithm were evaluated. In summary, the data which is first loaded into a required data structure, is followed by a thorough feature selection in order to discard any features which are not requested in the analysis. Afterwards, the discrete data in the filtered dataset is then transformed to Continuous data through various different encoding techniques, which can be properly analysed by the PCA algorithm. The cleaned data is then normalized, in order to ensure that the data to be fed to the PCA algorithm has a uniform variance. Moreover, the PCA algorithm can be evaluated through the singular value decomposition (SVD) approach or the covariance matrix approach, by utilising the normalized data as an input to the aforementioned approaches. Conclusively, the project's goals were met as the produced artefact outlines the aforementioned process through detailed explanations accompanied by code applications, and visualisation tools which grant students a better understanding of such concept.

5 VIDEO DEMONSTRATION

A video demonstration showcasing the developed notebook, can be accessed through the following links:

(1) https://youtu.be/IWYkN0AeK0o
(2) https://drive.google.com/drive/folders/1Ms7sojfVKI_BvzDhBag98KOfX34OoKwK

6 REFERENCES

[1] S. Mishra et al., "Multivariate Statistical Data Analysis-Principal Component Analysis," Int. J. Livest. Res., vol. 1, pp. 1-6, 2017. [Online]. Available: https://www.researchgate.net/publication/316652806_Principal_Component_Analysis. [Accessed: 18-Apr-2023].

[2] D. Li and S. Liu, "4.2.3.1 Principal Component Analysis," in Water Quality Monitoring and Management: Basis, Technology and Case Studies, 1st ed., S. K. Gupta and R. Kumar, Eds. Amsterdam, Netherlands: Elsevier, 2019. [Online]. Available: https://www.sciencedirect.com/topics/agricultural-and-biological-sciences/principal-component-analysis. [Accessed: 18-Apr-2023].

[3] N. B. Subramanian, "Types of PCA", aiaspirant.com. [Online]. Available: https://aiaspirant.com/types-of-pca/. [Accessed: 18-Apr-2023].

[4] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," arXiv preprint arXiv:0909.4061, 2009. [Online]. Available: https://arxiv.org/abs/0909.4061. [Accessed: 18-Apr-2023].

[5] Stack Exchange. "Why do we need to normalize data before Principal Component Analysis (PCA)?", Cross Validated, May 26, 2014. [Online]. Available: https://stats.stackexchange.com/questions/69157/why-do-we-need-to-normalize-data-before-principal-component-analysis-pca. [Accessed: 18-Apr-2023].

[6] DEVAKUMAR K. P., "COVID-19 Dataset", Kaggle, 2020. [Online]. Available: https://www.kaggle.com/datasets/imdevskp/corona-virus-report?select=country_wise_latest.csv. [Accessed: 18-Apr-2023].

[7] UCI MACHINE LEARNING, "Pima Indians Diabetes Database", Kaggle, 2016. [Online]. Available: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database. [Accessed: 18-Apr-2023].

[8] S. BANERJEE, "FIFA - Football World Cup Dataset", Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/iamsouravbanerjee/fifa-football-world-cup-dataset?select=FIFA+-+2014.csv. [Accessed: 18-Apr-2023].

[9] MATHNERD, "Iris Flower Dataset", Kaggle, 2018. [Online]. Available: https://www.kaggle.com/datasets/arshid/iris-flower-dataset. [Accessed: 18-Apr-2023].

[10] S. BANERJEE, "Software Industry Salary Dataset - 2022", Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/iamsouravbanerjee/software-professional-salaries-2022. [Accessed: 18-Apr-2023].

[11] R. Holbrook and A. Cook, "Principal Component Analysis, spotify.csv", Kaggle. [Online]. Available: https://www.kaggle.com/code/ryanholbrook/principal-component-analysis/data?select=spotify.csv. [Accessed: 18-Apr-2023].

[12] RUTHGN, "Wine Quality Data Set (Red & White Wine)", Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/ruthgn/wine-quality-data-set-red-white-wine. [Accessed: 18-Apr-2023].

[13] V. Karthik, "PCA for categorical features", Stack Overflow, Dec. 2016. [Online]. Available: https://stackoverflow.com/questions/40795141/pca-for-categorical-features#: :text=PCA%20is%20designed%20for%20continuous,yes%2C%20you%20can%20use%20PCA. [Accessed: 18-Apr-2023].

[14] Datagy. "Pandas get_dummies (One-Hot Encoding) Explained," Datagy.io, Feb. 2021. [Online]. Available: https://datagy.io/pandas-get-dummies/. [Accessed: 18-Apr-2023].

[15] DataCamp. "Dealing with Categorical Data". DataCamp, 2021. [Online]. Available: https://www.datacamp.com/tutorial/categorical-data. [Accessed: 18-Apr-2023].

[16] B. Roy, "All about Categorical Variable Encoding," Towards Data Science, Jul. 2, 2019. [Online]. Available: https://towardsdatascience.com/all-about-categorical-variable-encoding-305f3361fd02.[Accessed: 18-Apr-2023].

[17] T. Crosley, "What is the binary to decimal decoder?", Quora, May 8, 2018. [Online]. Available: https://www.quora.com/What-is-the-binary-to-decimal-decoder. [Accessed: 18-Apr-2023].

[18] Pandas. "pandas.factorize()". pandas 1.4.0 documentation, Jan. 07, 2022. [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.factorize.html. [Accessed: 18-Apr-2023].

[19] J. Brownlee, "One-Hot Encoding for Categorical Data," Machine Learning Mastery, Aug. 17, 2020. [Online]. Available: https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/. [Accessed: 18-Apr-2023].

[20] Vatsal, "Word2Vec Explained", Towards Data Science, Jul. 29, 2021. [Online]. Available: https://towardsdatascience.com/word2vec-explained-49c52b4ccb71. [Accessed: 18-Apr-2023].

[21] R. Sharma. "What is Normalization in Data Mining and How to Do It?", UpGrad, Sep. 22, 2022. [Online]. Available: https://www.upgrad.com/blog/normalization-in-data-mining/#: :text=Project%20Ideas%20%26%20Topics-,Z%2DScore%20Normalization,up%20to%20%2B3%20standard%20deviation. [Accessed: 18-Apr-2023].

[22] M. E. Wall, A. Rechtsteiner, and L. M. Rocha, "Singular Value Decomposition and Principal Component Analysis," in Learning from Data: Concepts, Theory, and Methods, vol. 2, Springer, Boston, MA, 2007, pp. 151-176, doi: 10.1007/0-306-47815-3_5. [Online]. Available: https://www.researchgate.net/publication/2167923_Singular_Value_Decomposition_and_Principal_Component_Analysis. [Accessed: 18-Apr-2023].

[23] CUEMATH, "Covariance Matrix", CUEMATH. [Online]. Available: https://www.cuemath.com/algebra/covariance-matrix/. [Accessed: 18-Apr-2023].

[24] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," in The Data Deluge: Can Libraries Cope with E-Science? Proceedings of a Conference Held at the Royal Society, London, UK, 4-5 November 2004, vol. 463, Royal Society Publishing, 2016, pp. 21-36. doi: 10.1098/rsta.2015.0202.[Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0202. [Accessed: 18-Apr-2023].

[25] K. Guillaumier, "Linear Algebra in Data Science and PCA"

[26] S. Mangale, "Scree Plot," Medium, Aug. 28, 2020. [Online]. Available: https://sanchitamangale12.medium.com/scree-plot-733ed72c8608. [Accessed: 18-Apr-2023].

[27] Scikit-learn, "sklearn.decomposition.PCA", scikit-learn.org. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. [Accessed: 18-Apr-2023].

[28] M. Kumar, "Memory error in NumPy SVD," in IEEE, 2014. [Online]. Available: https://stackoverflow.com/questions/21180298/memory-error-in-numpy-svd. [Accessed: 18-Apr-2023].

[29] N. B. Subramanian, "Types of PCA", aiaspirant.com. [Online]. Available: https://aiaspirant.com/types-of-pca/. [Accessed: 18-Apr-2023].