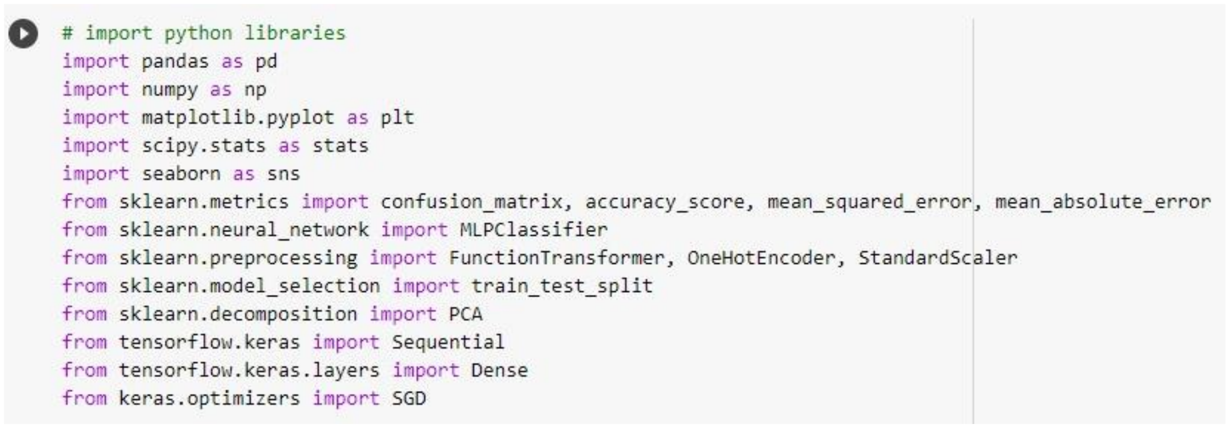
Prediction of client subscription of term deposit using MLP

Table of Contents

1. Data Loading and Importing Libraries 2
2. Data Preprocessing 4
   i. Handle Missing Values 4
   ii. Dropping ‘duration’ column 5
   iii. Handling outliers 6
3. Data transformation 8
   i. Q-Q Plots and Histograms of the features 8
   ii. Applying feature encoding technique 11
4. Splitting the dataset 12
5. Scale and/or standardized the feature 13
6. Perform Feature Engineering 14
   i. Identifying significant and independent features 14
   ii. PCA (Principal Component Analysis) for feature reduction 15
7. Applying Multi-Layer Perception 16

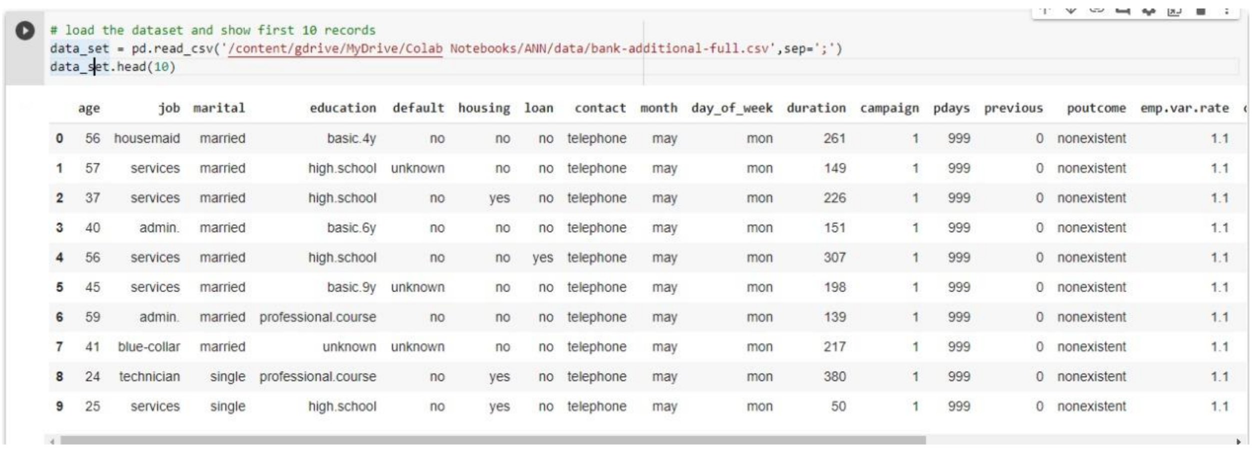
1. Data Loading and Importing Libraries

First of all, the following libraries which are necessary to the data mining process are imported before starting the preprocessing of the dataset



```python
# import python libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score, mean_squared_error, mean_absolute_error
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import FunctionTransformer, OneHotEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from keras.optimizers import SGD
```

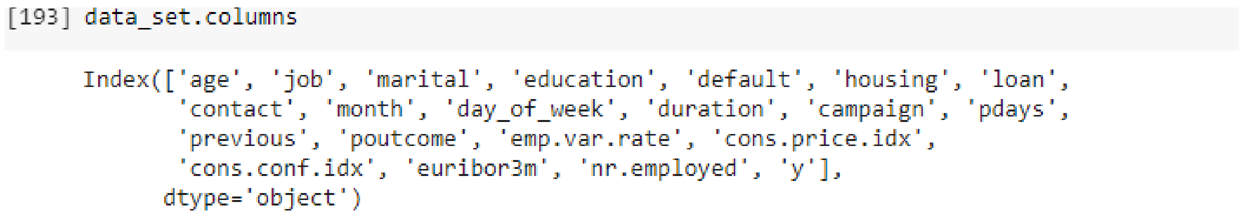
The following code was used to import the dataset, which was then confirmed using the following outputs.



```python
# load the dataset and show first 10 records
data_set = pd.read_csv('/content/gdrive/MyDrive/Colab Notebooks/ANN/data/bank-additional-full.csv',sep=';')
data_set.head(10)
```

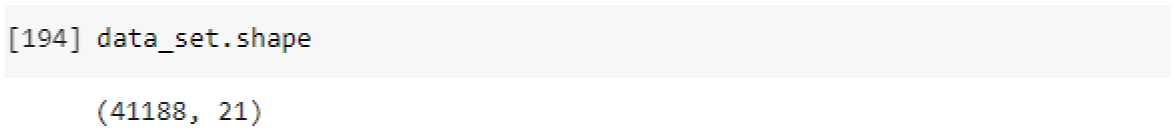
	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	emp.var.rate
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexistent	1.1
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	149	1	999	0	nonexistent	1.1
2	37	services	married	high.school	no	yes	no	telephone	may	mon	226	1	999	0	nonexistent	1.1
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	151	1	999	0	nonexistent	1.1
4	56	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexistent	1.1
5	45	services	married	basic.9y	unknown	no	no	telephone	may	mon	198	1	999	0	nonexistent	1.1
6	59	admin.	married	professional.course	no	no	no	telephone	may	mon	139	1	999	0	nonexistent	1.1
7	41	blue-collar	married	unknown	unknown	no	no	telephone	may	mon	217	1	999	0	nonexistent	1.1
8	24	technician	single	professional.course	no	yes	no	telephone	may	mon	380	1	999	0	nonexistent	1.1
9	25	services	single	high.school	no	yes	no	telephone	may	mon	50	1	999	0	nonexistent	1.1

Columns in the dataset;



```
[193] data_set.columns

      Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
             'contact', 'month', 'day_of_week', 'duration', 'campaign', 'pdays',
             'previous', 'poutcome', 'emp.var.rate', 'cons.price.idx',
             'cons.conf.idx', 'euribor3m', 'nr.employed', 'y'],
            dtype='object')
```

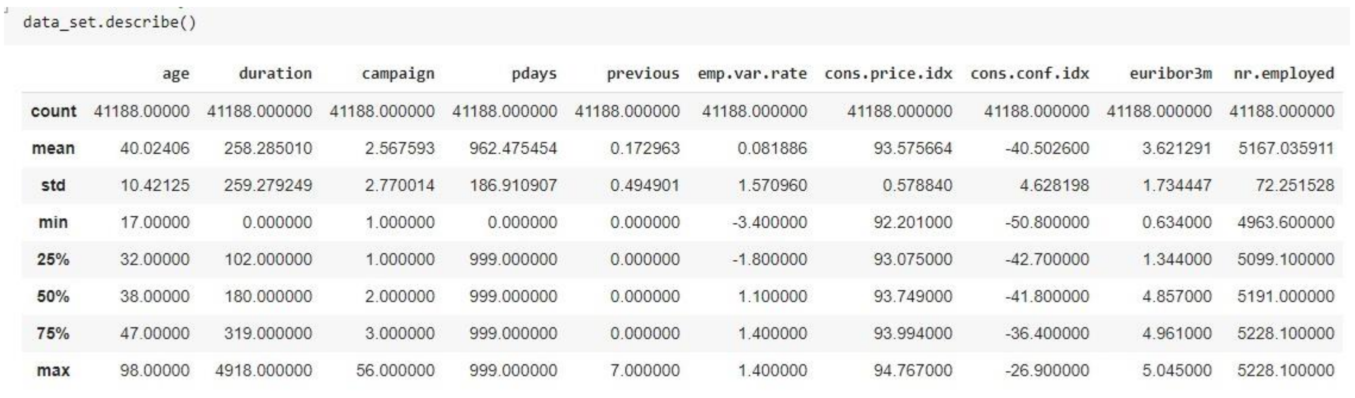
The shape of the dataset:



```
[194] data_set.shape

       (41188, 21)
```

As it can be seen from the output data set has 41188 rows and 21 columns.

Description of the dataset:



```
data_set.describe()
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.00000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000
mean	40.02406	258.285010	2.567593	962.475454	0.172963	0.081886	93.575664	-40.502600	3.621291	5167.035911
std	10.42125	259.279249	2.770014	186.910907	0.494901	1.570960	0.578840	4.628198	1.734447	72.251528
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.634000	4963.600000
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.800000	93.075000	-42.700000	1.344000	5099.100000
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	5191.000000
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.994000	-36.400000	4.961000	5228.100000
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.767000	-26.900000	5.045000	5228.100000

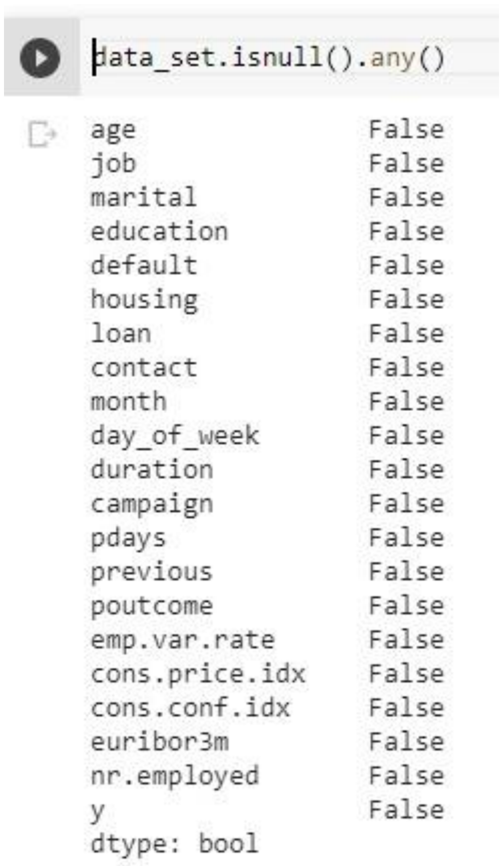
Count of each column, mean and standard deviation, minimum value, maximum value, 1st quantile, 2nd quantile, 3rd quantile values of each column are shown here. By considering each column a clear idea of how the data is distributed can be obtained.

2. Data Preprocessing

The majority of real-world data has noise, missing values, and is in an inadequate format that cannot be directly utilized for the models. Preprocessing data is important for cleaning and preparing data for a model, which enhances the model's accuracy and efficiency.

i. Handle Missing Values

The following code is used to see if the data set has any missing values.



```
data_set.isnull().any()
```

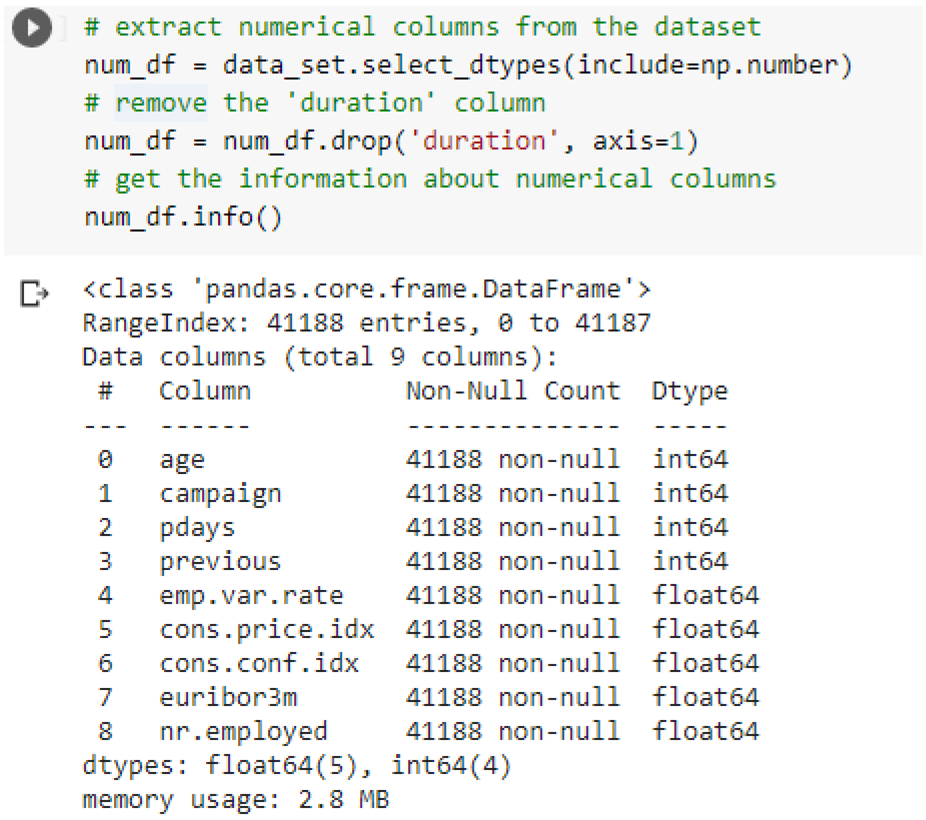
```
age               False
job               False
marital           False
education         False
default           False
housing           False
loan              False
contact           False
month             False
day_of_week       False
duration          False
campaign          False
pdays             False
previous          False
poutcome          False
emp.var.rate      False
cons.price.idx    False
cons.conf.idx     False
euribor3m         False
nr.employed       False
y                 False
dtype: bool
```

As it can be seen as the output there are no missing values in the dataset.

ii. Dropping ‘duration’ column

When considering the duration attribute, it has a significant impact on the output target (for example, if duration=0, y='no'). However, the duration is unknown before a call is made. Also, y is known at the end of the call. Since the goal is to create a realistic predictive model, the duration column has been removed.

Hence dataset has more than 20 columns I have divided it into categorical columns and numerical columns so that data preprocessing will be easier to handle.

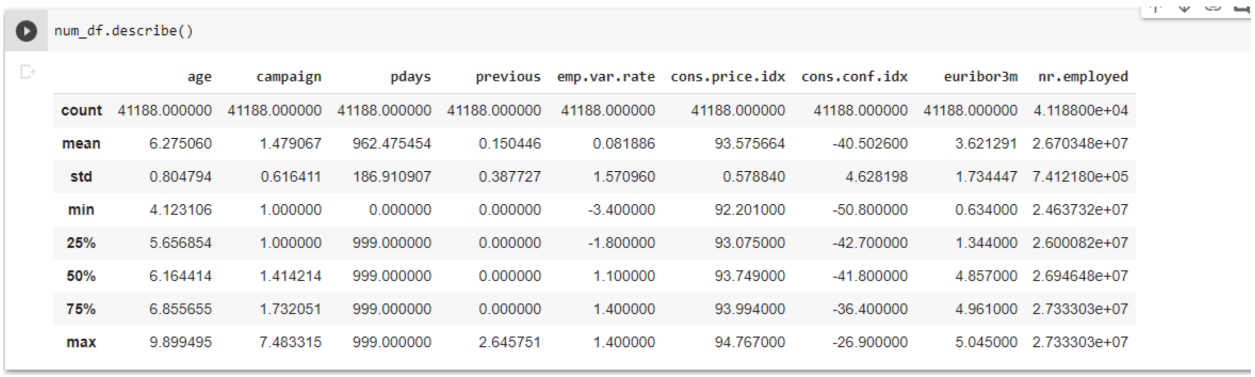


```python
# extract numerical columns from the dataset
num_df = data_set.select_dtypes(include=np.number)
# remove the 'duration' column
num_df = num_df.drop('duration', axis=1)
# get the information about numerical columns
num_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   age             41188 non-null  int64  
 1   campaign        41188 non-null  int64  
 2   pdays           41188 non-null  int64  
 3   previous        41188 non-null  int64  
 4   emp.var.rate    41188 non-null  float64
 5   cons.price.idx  41188 non-null  float64
 6   cons.conf.idx   41188 non-null  float64
 7   euribor3m       41188 non-null  float64
 8   nr.employed     41188 non-null  float64
dtypes: float64(5), int64(4)
memory usage: 2.8 MB
```

This num_df (numerical columns) data frame consists of age, campaign, pdays, previous, emp.var.rate ,cons.price.idx, cons.conf.idx, euribor3m, nr.employed columns.

Description of the num_df data frame is as follows:

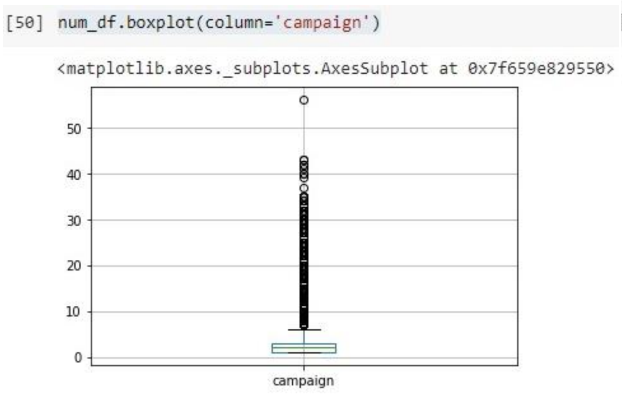


```
num_df.describe()
```

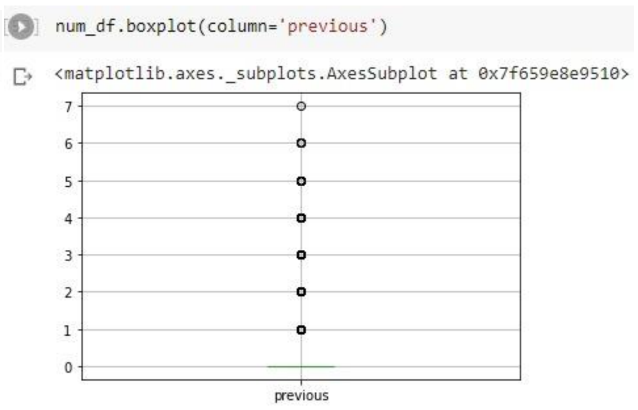
	age	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	4.118800e+04
mean	6.275060	1.479067	962.475454	0.150446	0.081886	93.575664	-40.502600	3.621291	2.670348e+07
std	0.804794	0.616411	186.910907	0.387727	1.570960	0.578840	4.628198	1.734447	7.412180e+05
min	4.123106	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.634000	2.463732e+07
25%	5.656854	1.000000	999.000000	0.000000	-1.800000	93.075000	-42.700000	1.344000	2.600082e+07
50%	6.164414	1.414214	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	2.694648e+07
75%	6.855655	1.732051	999.000000	0.000000	1.400000	93.994000	-36.400000	4.961000	2.733303e+07
max	9.899495	7.483315	999.000000	2.645751	1.400000	94.767000	-26.900000	5.045000	2.733303e+07

iii. Handling outliers

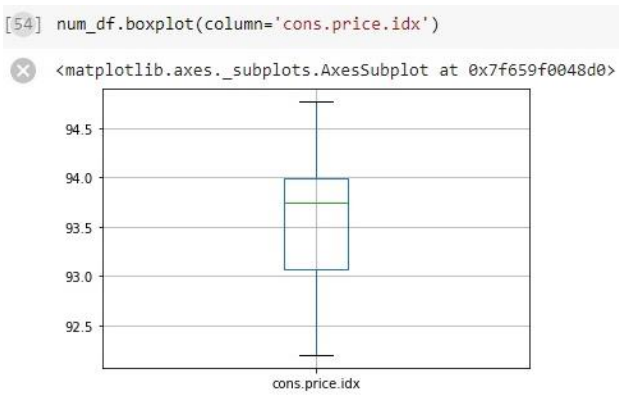
The next step is to see if the dataset contains any outliers. The boxplot tool can be used to visualize the dataset and look for outliers.



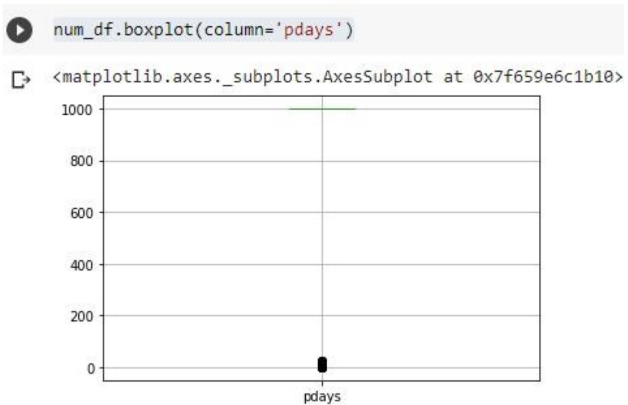
```
[50] num_df.boxplot(column='campaign')

<matplotlib.axes._subplots.AxesSubplot at 0x7f659e829550>
```



```
num_df.boxplot(column='previous')

<matplotlib.axes._subplots.AxesSubplot at 0x7f659e8e9510>
```



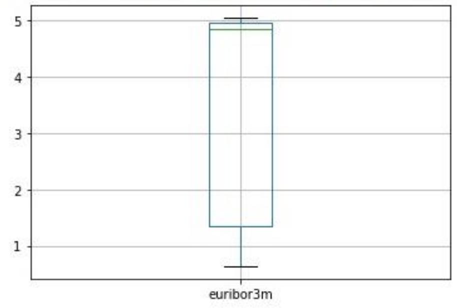
```
[54] num_df.boxplot(column='cons.price.idx')

<matplotlib.axes._subplots.AxesSubplot at 0x7f659f0048d0>
```



```
num_df.boxplot(column='pdays')

<matplotlib.axes._subplots.AxesSubplot at 0x7f659e6c1b10>
```

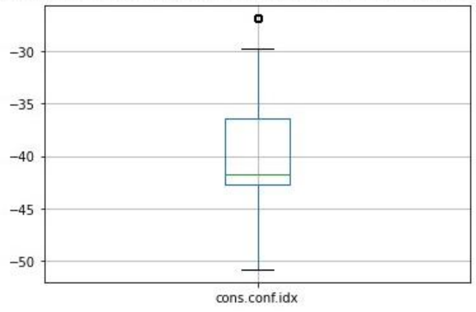
```
[56] num_df.boxplot(column='euribor3m')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f65a78cf7d0>
```



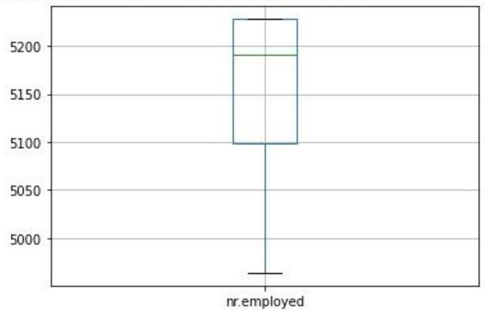
```
[55] num_df.boxplot(column='cons.conf.idx')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f659efdc210>
```



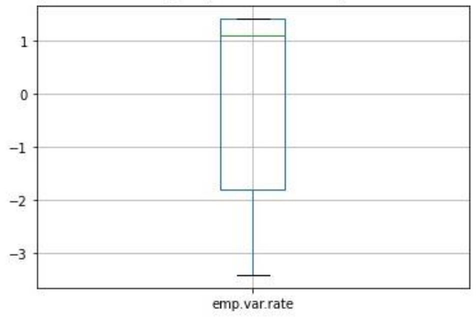
```
[57] num_df.boxplot(column='nr.employed')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f659ebf5b50>
```



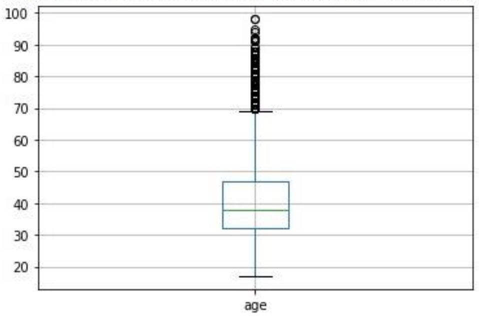
```
num_df.boxplot(column='emp.var.rate')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f659ebaad90>
```



```
[49] num_df.boxplot(column='age')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f65a7998910>
```

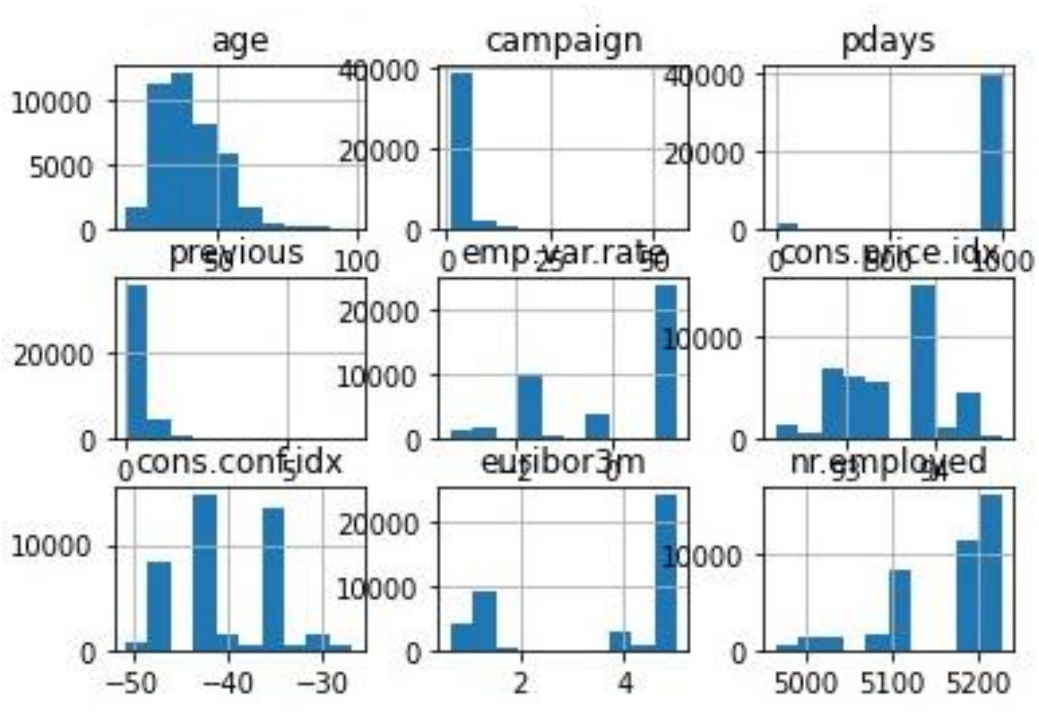


Since some of the data are skewed the data points outside the boxplots are not considered as outliers. Therefore, as per the above boxplots, it can be concluded that there are no outliers in the dataset.

3. Data transformation

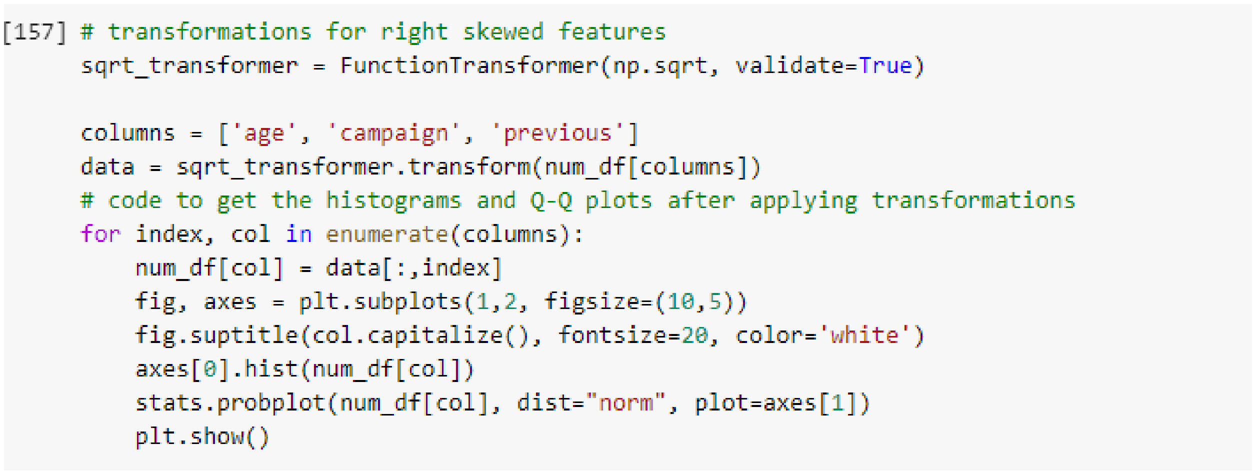
i. Q-Q Plots and Histograms of the features

Histogram is constructed to observe the skewness of the data. The output is as follows.



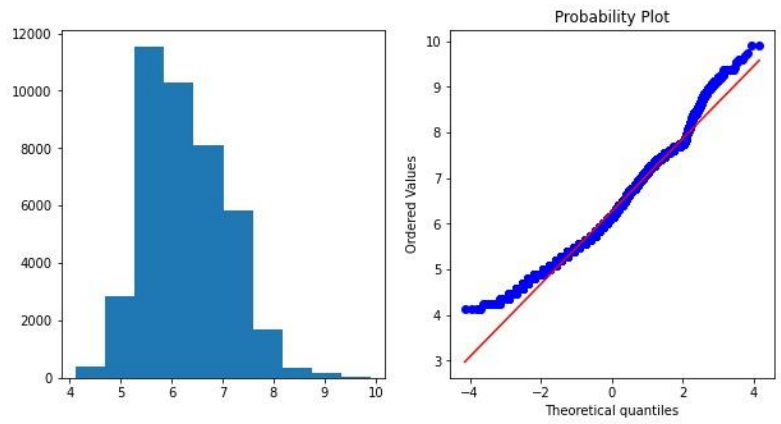
Age, Campaign, Previous and nr.employed columns data are skewed. Therefore, it is a must to transform them into a normal distribution. As per the histograms of each feature, Age, Campaign, and Previous columns are left-skewed while the nr.employed feature is right-skewed. Therefore, Logarithmic transformation is applied to the Age, Campaign, and Previous features and Exponential or power transformation is applied to the nr.employed column. I have applied transformation to both testing and training sets. Code is as below.

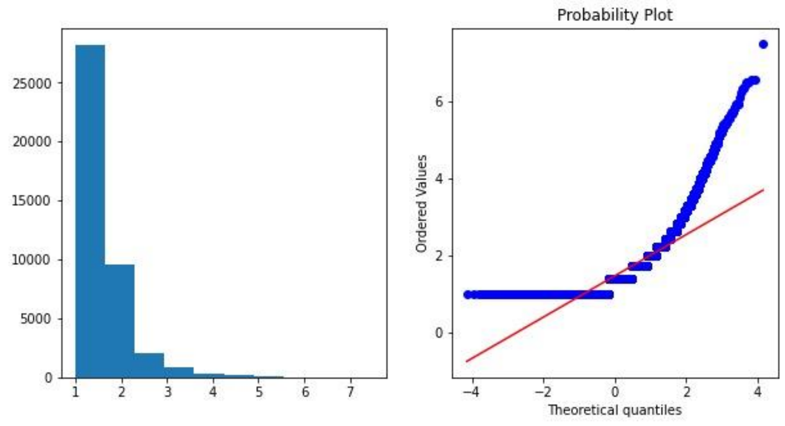
Logarithmic transformation:



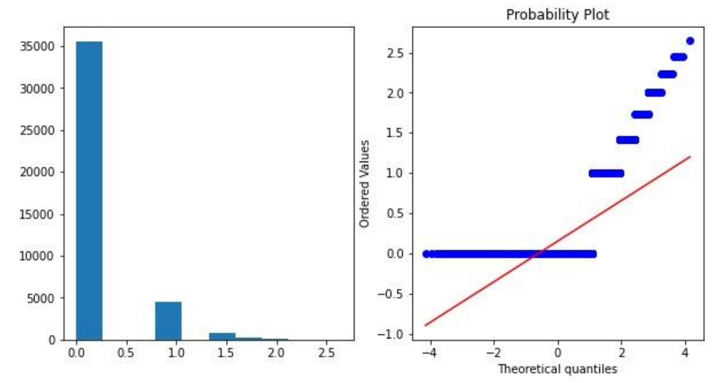
```
[157] # transformations for right skewed features
      sqrt_transformer = FunctionTransformer(np.sqrt, validate=True)

      columns = ['age', 'campaign', 'previous']
      data = sqrt_transformer.transform(num_df[columns])
      # code to get the histograms and Q-Q plots after applying transformations
      for index, col in enumerate(columns):
          num_df[col] = data[:,index]
          fig, axes = plt.subplots(1,2, figsize=(10,5))
          fig.suptitle(col.capitalize(), fontsize=20, color='white')
          axes[0].hist(num_df[col])
          stats.probplot(num_df[col], dist="norm", plot=axes[1])
          plt.show()
```

Q-Q Plots and the histogram of the age, campaign and previous columns after transformation are as follows

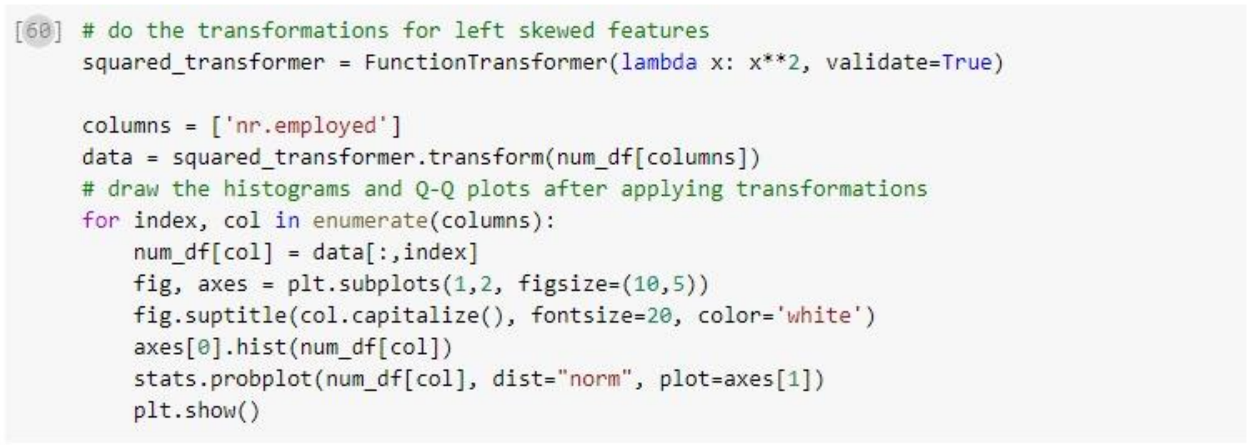




[59]

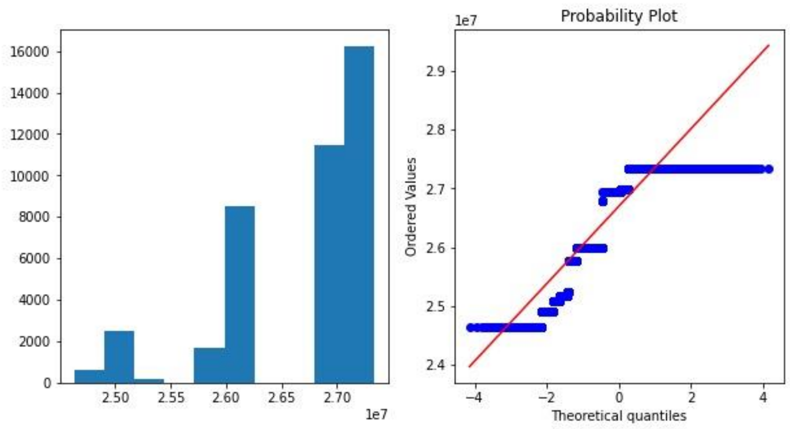


Exponential or power transformation:



```
[60] # do the transformations for left skewed features
     squared_transformer = FunctionTransformer(lambda x: x**2, validate=True)

     columns = ['nr.employed']
     data = squared_transformer.transform(num_df[columns])
     # draw the histograms and Q-Q plots after applying transformations
     for index, col in enumerate(columns):
         num_df[col] = data[:,index]
         fig, axes = plt.subplots(1,2, figsize=(10,5))
         fig.suptitle(col.capitalize(), fontsize=20, color='white')
         axes[0].hist(num_df[col])
         stats.probplot(num_df[col], dist="norm", plot=axes[1])
         plt.show()
```

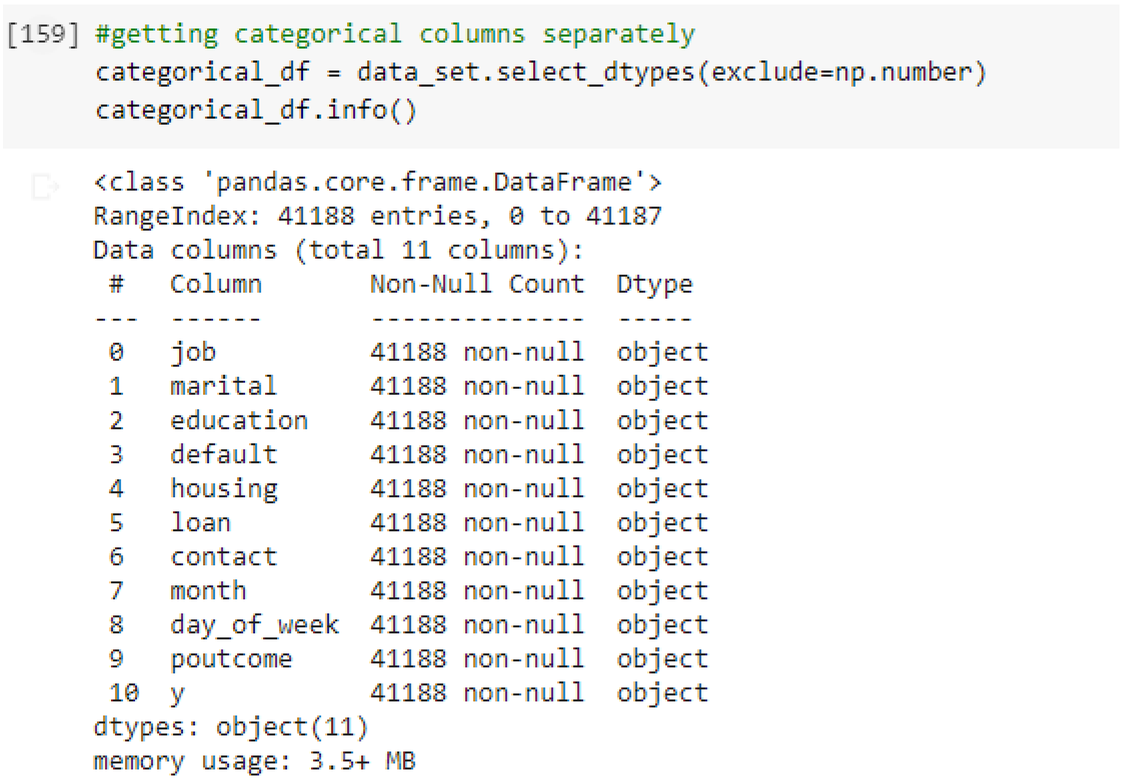
Q-Q plot and the Histogram of the nr.employed column is as follows:



Now all the features including age, campaign, previous, nr.em;poyed features data have a normal distribution.

ii. Applying feature encoding technique

Same as the numerical columns I have separated categorical column and added them to a new data frame as follows.



```
[159] #getting categorical columns separately
      categorical_df = data_set.select_dtypes(exclude=np.number)
      categorical_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   job          41188 non-null  object
 1   marital      41188 non-null  object
 2   education    41188 non-null  object
 3   default      41188 non-null  object
 4   housing      41188 non-null  object
 5   loan         41188 non-null  object
 6   contact      41188 non-null  object
 7   month        41188 non-null  object
 8   day_of_week  41188 non-null  object
 9   poutcome     41188 non-null  object
 10  y            41188 non-null  object
dtypes: object(11)
memory usage: 3.5+ MB
```

These categorical data is in an object format. In order to convert them to numerical values feature encoding is used.

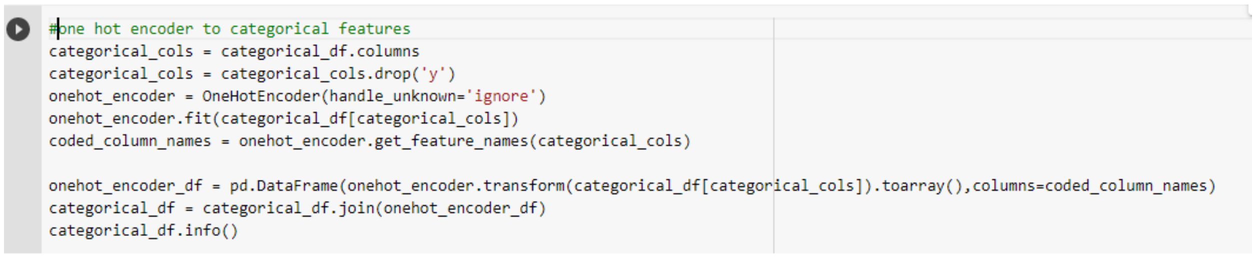
There are 2 types of feature encoding techniques.

a. Label Encoding
b. One hot encoding

The Label Encoder encodes classes with values ranging from 0 to n-1, where n is the number of distinct classes. Label encoding will lead the model to believe that a column contains data in some sort of order or hierarchy. To solve this problem One hot encoding can be used.

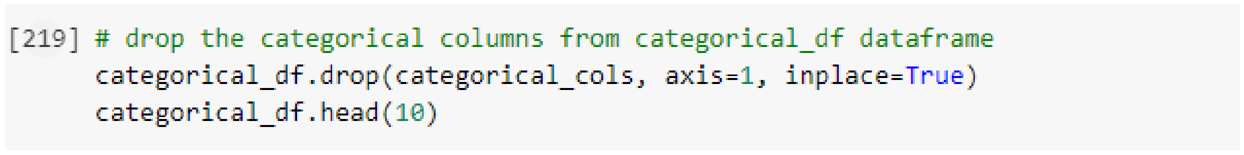
One Hot Encoding takes a column with categorical data and divides it into multiple columns. Depending on which column has a particular class, the classes are replaced by binaries (1s and 0s).

To apply one hot encoding following code is used



```
#one hot encoder to categorical features
categorical_cols = categorical_df.columns
categorical_cols = categorical_cols.drop('y')
onehot_encoder = OneHotEncoder(handle_unknown='ignore')
onehot_encoder.fit(categorical_df[categorical_cols])
coded_column_names = onehot_encoder.get_feature_names(categorical_cols)

onehot_encoder_df = pd.DataFrame(onehot_encoder.transform(categorical_df[categorical_cols]).toarray(),columns=coded_column_names)
categorical_df = categorical_df.join(onehot_encoder_df)
categorical_df.info()
```

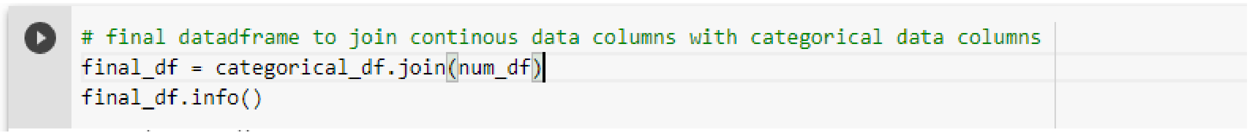
Hence now the categorical column are encoded and I have joined it with the dataset, I have dropped them from the dataframe.



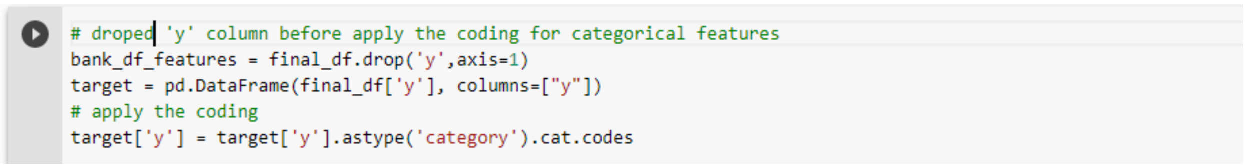
```
[219] # drop the categorical columns from categorical_df dataframe
      categorical_df.drop(categorical_cols, axis=1, inplace=True)
      categorical_df.head(10)
```

4. Splitting the dataset

Before splitting the dataset, I have created a new data frame to combine num_df and categorical_df. Then created another data frame as bank_df_features and assigned feature to it and added y column to a new dataset as target.



```
# final datadframe to join continous data columns with categorical data columns
final_df = categorical_df.join(num_df)
final_df.info()
```

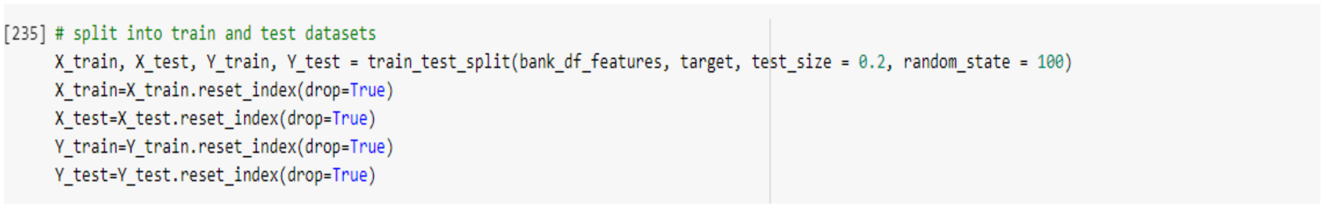


```
# droped 'y' column before apply the coding for categorical features
bank_df_features = final_df.drop('y',axis=1)
target = pd.DataFrame(final_df['y'], columns=["y"])
# apply the coding
target['y'] = target['y'].astype('category').cat.codes
```

As above snapshot I have applied encoding to y column using categorical coding because y column contain only two classes.

The training set is where we train and fit our model to fit the parameters, whilst test data is only used to evaluate the model's performance. The output of training data is accessible to model, but testing data is unobserved data for which predictions must be generated.

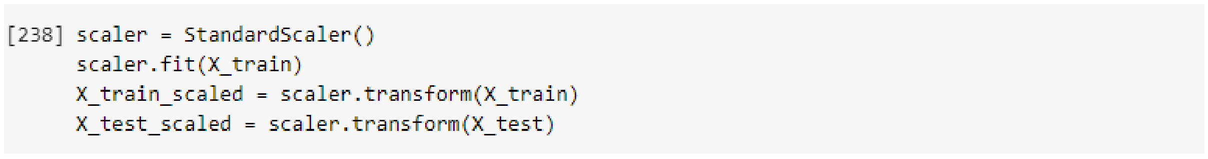
I have split my training and testing data to the 80% and 20% ratio. bank_df_features includes features to predict the y and the target column includes y and the indexes are reset as follows in order to avoid further confusion.



```
[235] # split into train and test datasets
      X_train, X_test, Y_train, Y_test = train_test_split(bank_df_features, target, test_size = 0.2, random_state = 100)
      X_train=X_train.reset_index(drop=True)
      X_test=X_test.reset_index(drop=True)
      Y_train=Y_train.reset_index(drop=True)
      Y_test=Y_test.reset_index(drop=True)
```

5. Scale and/or standardized the feature

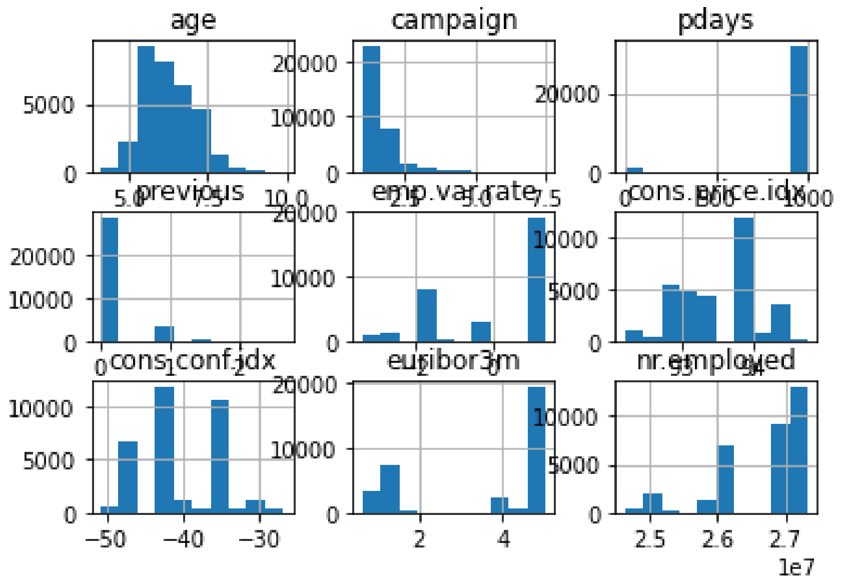
As the scaling method standardization is used. Data is centered using standardization. Since only continuous data can be standardized, StandardScaler() must be only used for continuous data columns. I have already dropped categorical data columns. Therefore, I have straight away applied data frame to the scaler. Dataset is split into test and train set I have applied standardization separately for both sets. Code is as follows.



```
[238] scaler = StandardScaler()
      scaler.fit(X_train)
      X_train_scaled = scaler.transform(X_train)
      X_test_scaled = scaler.transform(X_test)
```

Histogram after standardization;

```
dtype=object)
```

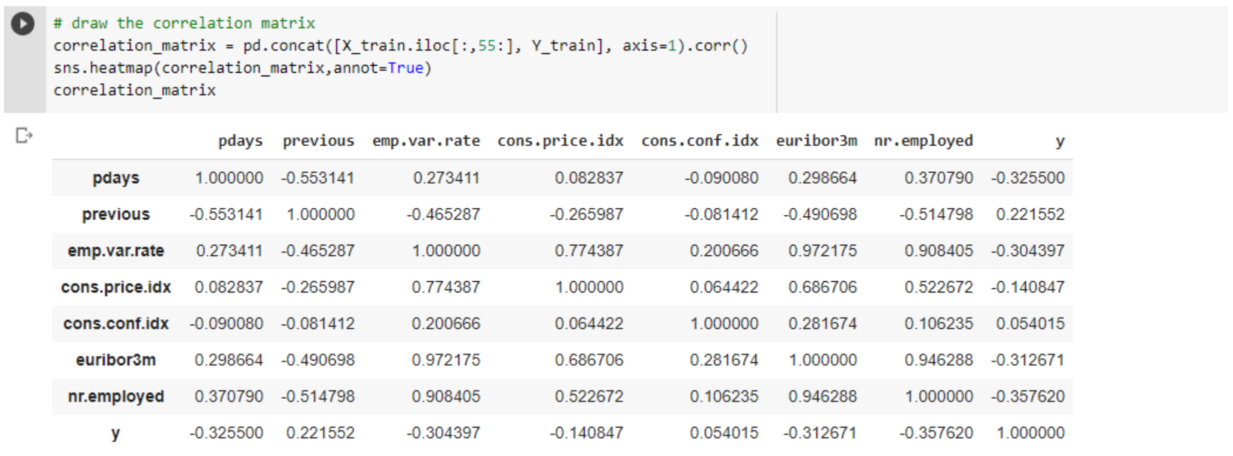


6. Perform Feature Engineering

i. Identifying significant and independent features

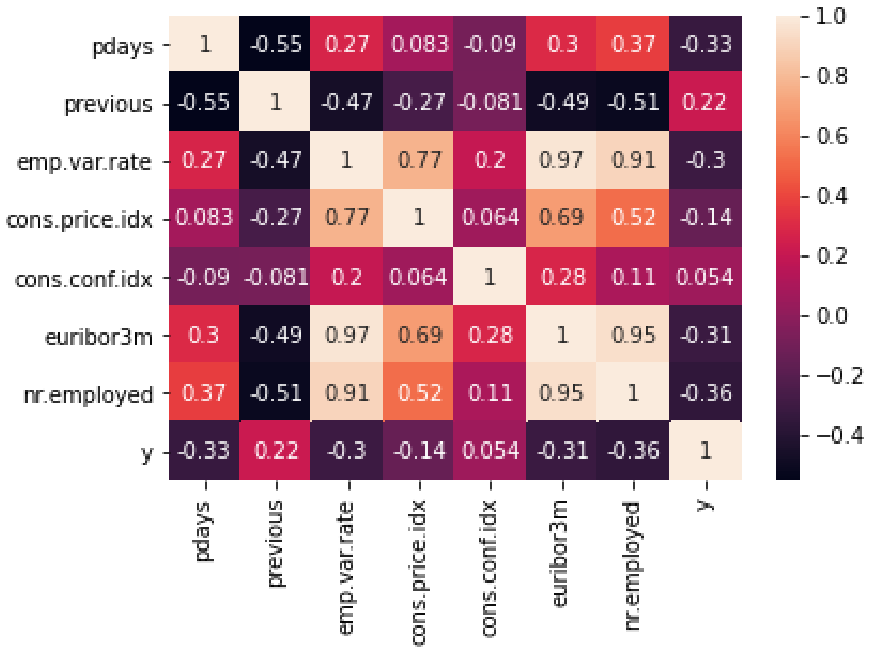
A feature is an individual attribute or characteristic of a process under study in machine learning. The quality of features in a dataset has a significant impact on the quality of machine learning algorithms' output. Choosing relevant features is a critical step in efficiently training and calibrating algorithms.

In order to capture the significance of the features, a correlation matrix is constructed using the following code and the following output was obtained.



```
# draw the correlation matrix
correlation_matrix = pd.concat([X_train.iloc[:,55:], Y_train], axis=1).corr()
sns.heatmap(correlation_matrix,annot=True)
correlation_matrix
```

	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
pdays	1.000000	-0.553141	0.273411	0.082837	-0.090080	0.298664	0.370790	-0.325500
previous	-0.553141	1.000000	-0.465287	-0.265987	-0.081412	-0.490698	-0.514798	0.221552
emp.var.rate	0.273411	-0.465287	1.000000	0.774387	0.200666	0.972175	0.908405	-0.304397
cons.price.idx	0.082837	-0.265987	0.774387	1.000000	0.064422	0.686706	0.522672	-0.140847
cons.conf.idx	-0.090080	-0.081412	0.200666	0.064422	1.000000	0.281674	0.106235	0.054015
euribor3m	0.298664	-0.490698	0.972175	0.686706	0.281674	1.000000	0.946288	-0.312671
nr.employed	0.370790	-0.514798	0.908405	0.522672	0.106235	0.946288	1.000000	-0.357620
y	-0.325500	0.221552	-0.304397	-0.140847	0.054015	-0.312671	-0.357620	1.000000



As it can be seen from the correlation matrix,

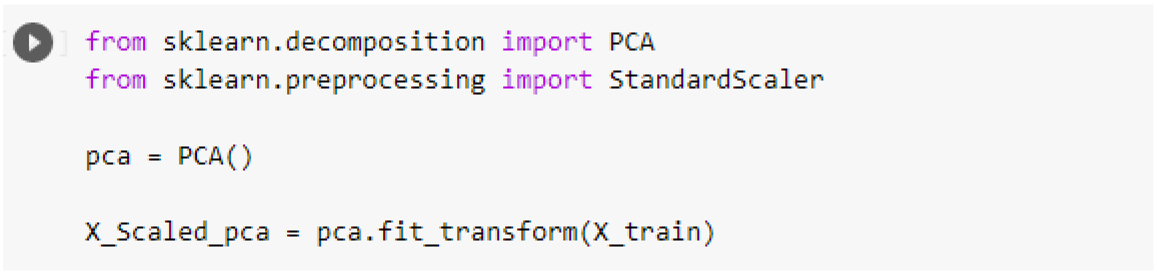
- Pdays and y are a negative correlation
- nr.employed and y are positively correlated.

In order to get a better idea to identify what features can be dropped, PCA is performed.

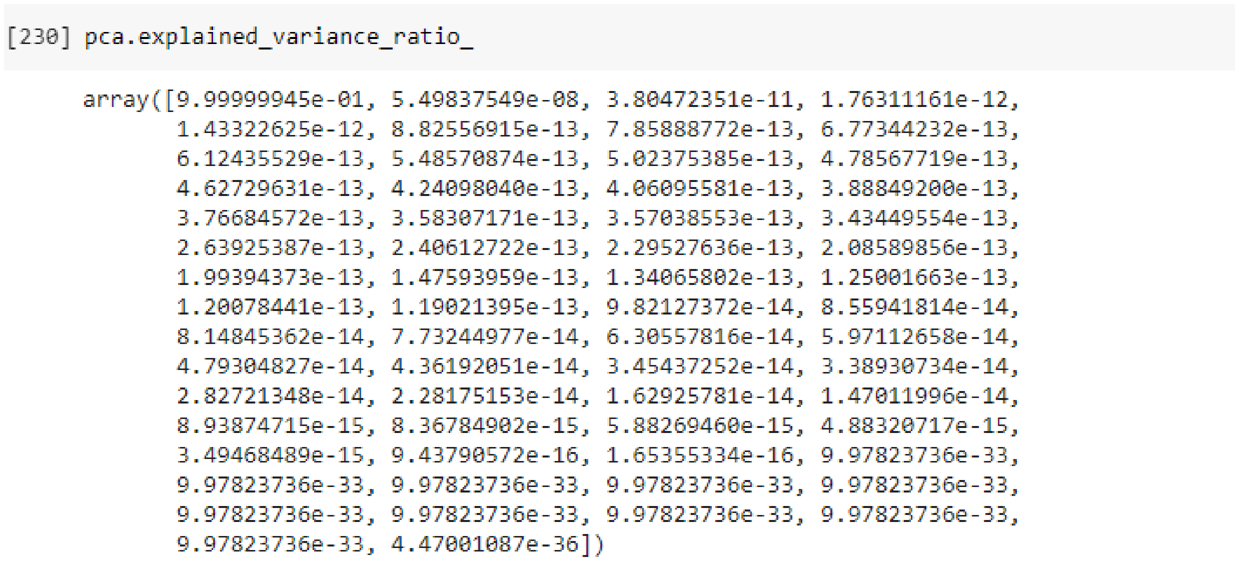
ii. PCA (Principal Component Analysis) for feature reduction

Next, it is required to identify what are the important variables in the dataset and extract them. To identify it, PCA is used. Therefore, explained variance ratio is calculated in order to identify the number of components to apply to PCA so that to capture maximum of 95% variance to capture while dropping 5% of variance. The proportion of each principal component axis' contribution to the variance of the entire dataset is represented by the explained variance ratio.

To select no of components for PCA following code is used,

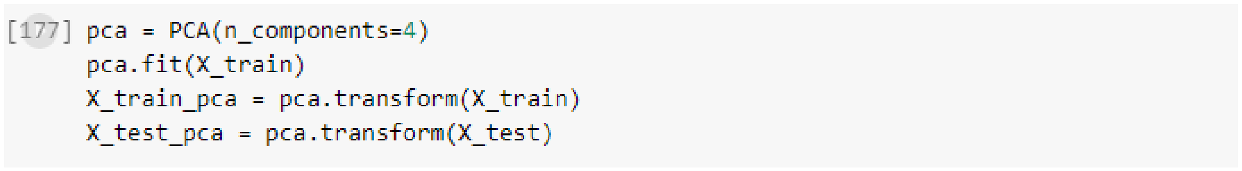


```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

pca = PCA()

X_Scaled_pca = pca.fit_transform(X_train)
```



```
[230] pca.explained_variance_ratio_

      array([9.99999945e-01, 5.49837549e-08, 3.80472351e-11, 1.76311161e-12,
             1.43322625e-12, 8.82556915e-13, 7.85888772e-13, 6.77344232e-13,
             6.12435529e-13, 5.48570874e-13, 5.02375385e-13, 4.78567719e-13,
             4.62729631e-13, 4.24098040e-13, 4.06095581e-13, 3.88849200e-13,
             3.76684572e-13, 3.58307171e-13, 3.57038553e-13, 3.43449554e-13,
             2.63925387e-13, 2.40612722e-13, 2.29527636e-13, 2.08589856e-13,
             1.99394373e-13, 1.47593959e-13, 1.34065802e-13, 1.25001663e-13,
             1.20078441e-13, 1.19021395e-13, 9.82127372e-14, 8.55941814e-14,
             8.14845362e-14, 7.73244977e-14, 6.30557816e-14, 5.97112658e-14,
             4.79304827e-14, 4.36192051e-14, 3.45437252e-14, 3.38930734e-14,
             2.82721348e-14, 2.28175153e-14, 1.62925781e-14, 1.47011996e-14,
             8.93874715e-15, 8.36784902e-15, 5.88269460e-15, 4.88320717e-15,
             3.49468489e-15, 9.43790572e-16, 1.65355334e-16, 9.97823736e-33,
             9.97823736e-33, 9.97823736e-33, 9.97823736e-33, 9.97823736e-33,
             9.97823736e-33, 9.97823736e-33, 9.97823736e-33, 9.97823736e-33,
             9.97823736e-33, 4.47001087e-36])
```

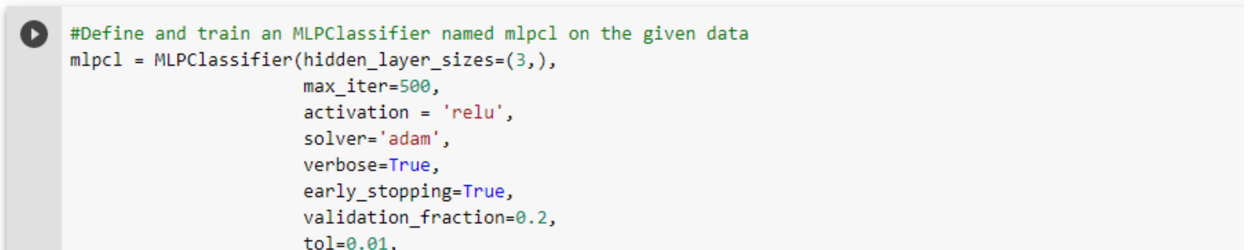
As it can be seen in the output. 1st variable shows 0.999 variance to the dataset. 99% of the variance is c captured by that variable. I have selected 4 components to capture 99.9% of the variance.



```
[177] pca = PCA(n_components=4)
      pca.fit(X_train)
      X_train_pca = pca.transform(X_train)
      X_test_pca = pca.transform(X_test)
```

7. Applying Multi-Layer Perception

I have used MLPClassifier class to implement a multi-layer perception algorithm to train the model using backpropagation. As the below image I have define and train an MLPClassifier named mlpcl on the dataset as below.

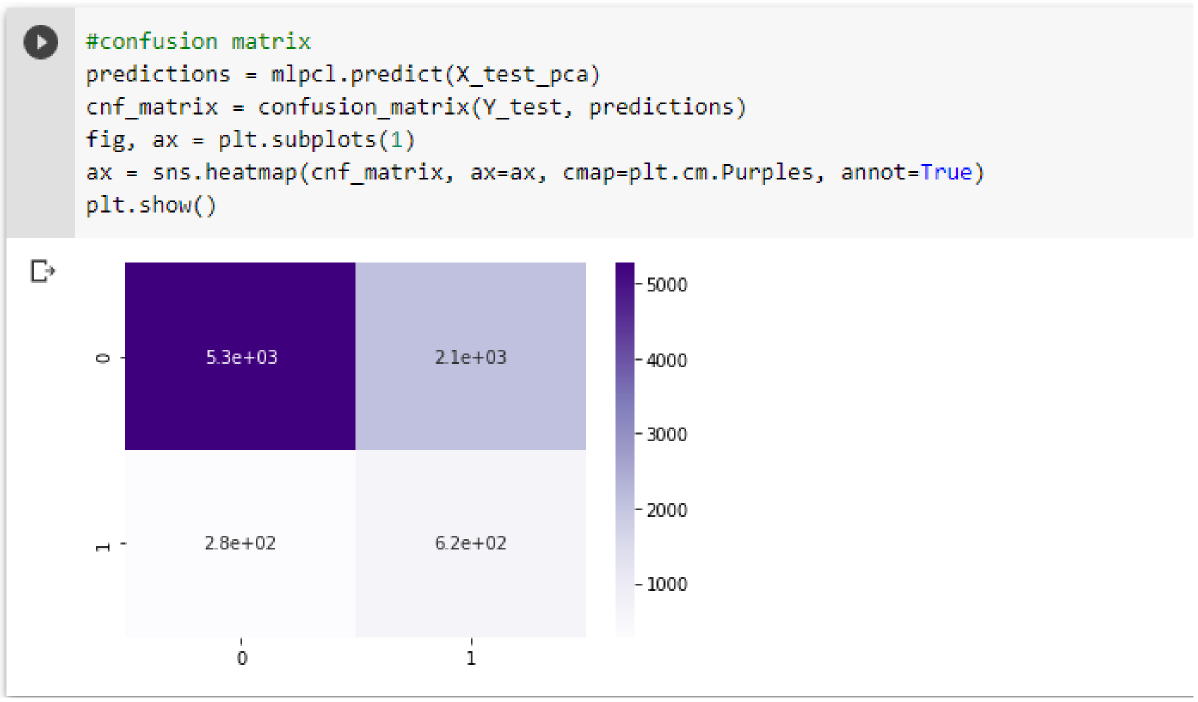


```
#Define and train an MLPClassifier named mlpcl on the given data
mlpcl = MLPClassifier(hidden_layer_sizes=(3,),
                      max_iter=500,
                      activation = 'relu',
                      solver='adam',
                      verbose=True,
                      early_stopping=True,
                      validation_fraction=0.2,
                      tol=0.01,
```

For this classifier I have given following parameters in order to create the classifier.

- Hidden layer size- 3
- Maximum iteration-500,
- Activation function - rectified linear unit function, returns f(x) = max(0, x)
- Solver (for weight optimization) - ‘adam’ refers to a stochastic gradient-based optimizer
- Verbose as true – to print the progress message to stdout.
- Early stopping as true – in order to terminate training when validation score is not improving.
- Validation fraction set to 0.2- refer to proportion of training data to set aside as validation set for early stopping.
- Tol to 0.01- refers to the tolerance for the optimization.
- Learning rate – 0.00001. this learning rate determines random number generation for weights and bias initialization.

Once the classifier is trained, I have generated a confusion matrix as follows to get a better idea about the performance.



```
#confusion matrix
predictions = mlpcl.predict(X_test_pca)
cnf_matrix = confusion_matrix(Y_test, predictions)
fig, ax = plt.subplots(1)
ax = sns.heatmap(cnf_matrix, ax=ax, cmap=plt.cm.Purples, annot=True)
plt.show()
```

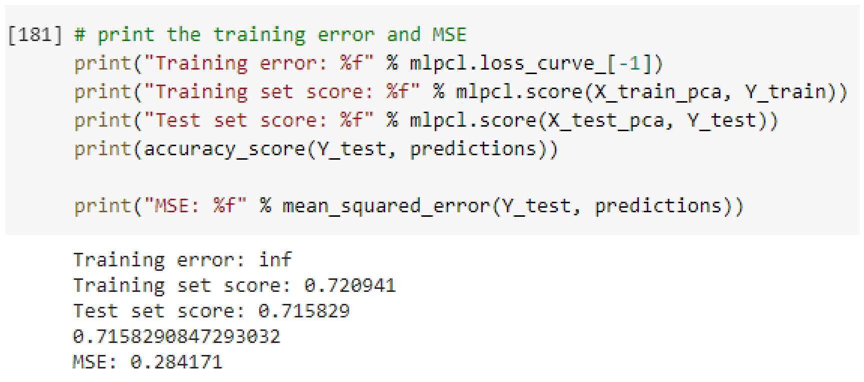
True Positive Values: 620

True Negative Values: 5300

False Negative Values: 2100

False Positive Values: 280

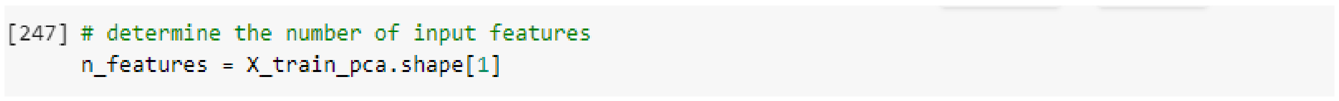
The Training error, Training set score, Test set score and the mean squared error value as follows.



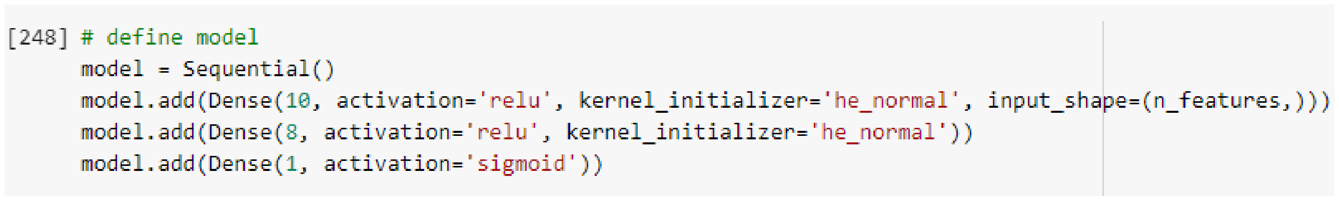
```
[181] # print the training error and MSE
      print("Training error: %f" % mlpcl.loss_curve_[-1])
      print("Training set score: %f" % mlpcl.score(X_train_pca, Y_train))
      print("Test set score: %f" % mlpcl.score(X_test_pca, Y_test))
      print(accuracy_score(Y_test, predictions))

      print("MSE: %f" % mean_squared_error(Y_test, predictions))
```

```
Training error: inf
Training set score: 0.720941
Test set score: 0.715829
0.7158290847293032
MSE: 0.284171
```

I have defined the model using sequential().



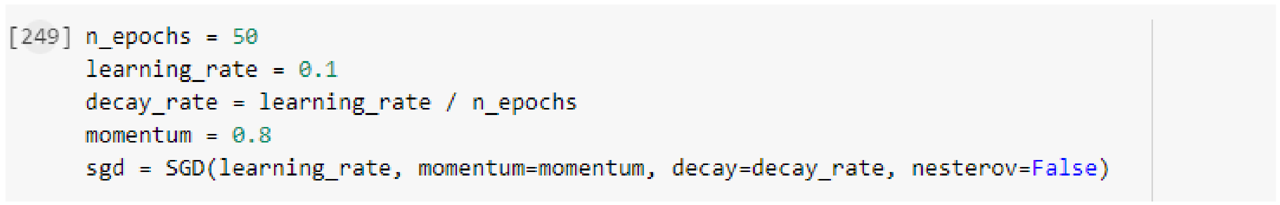
```
[247] # determine the number of input features
      n_features = X_train_pca.shape[1]
```



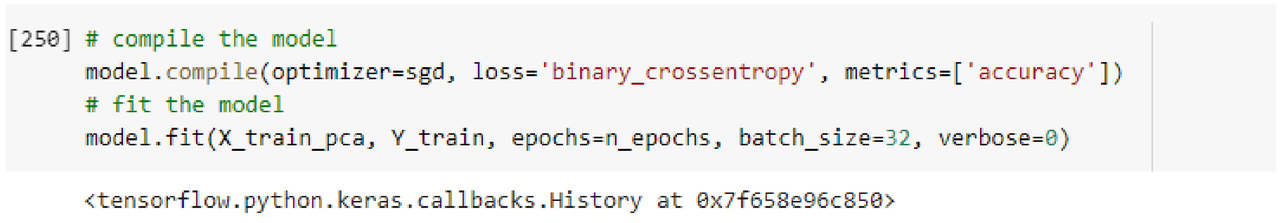
```
[248] # define model
      model = Sequential()
      model.add(Dense(10, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
      model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
      model.add(Dense(1, activation='sigmoid'))
```

For the first hidden layer there are 10 neurons and its activation function is relu. When considering the second hidden layer it has 8 neurons and activation function is same as the first hidden layer. And for the output layer only sigmoid function is used.

In order to compile the model I have used following code.

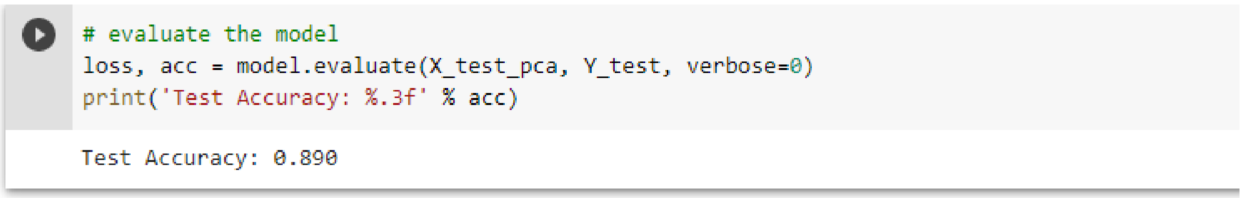


```
[249] n_epochs = 50
      learning_rate = 0.1
      decay_rate = learning_rate / n_epochs
      momentum = 0.8
      sgd = SGD(learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
```



```
[250] # compile the model
      model.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['accuracy'])
      # fit the model
      model.fit(X_train_pca, Y_train, epochs=n_epochs, batch_size=32, verbose=0)

      <tensorflow.python.keras.callbacks.History at 0x7f658e96c850>
```

In order to evaluate the test accuracy of the model,



```
# evaluate the model
loss, acc = model.evaluate(X_test_pca, Y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)

Test Accuracy: 0.890
```

As it can be seen from the above output the model shows test accuracy of 89%.