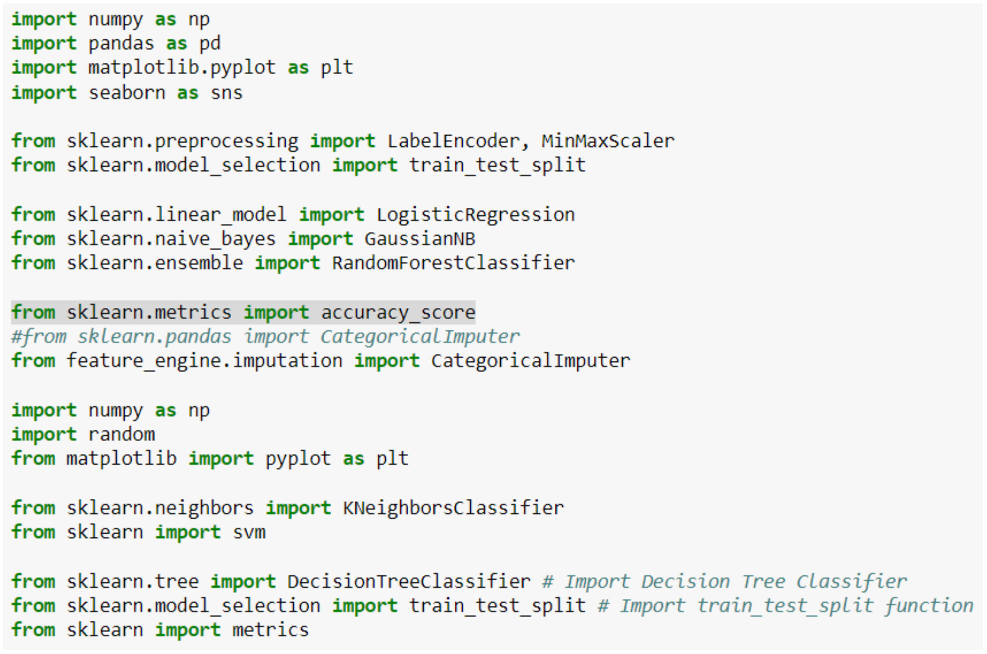
Introduction:

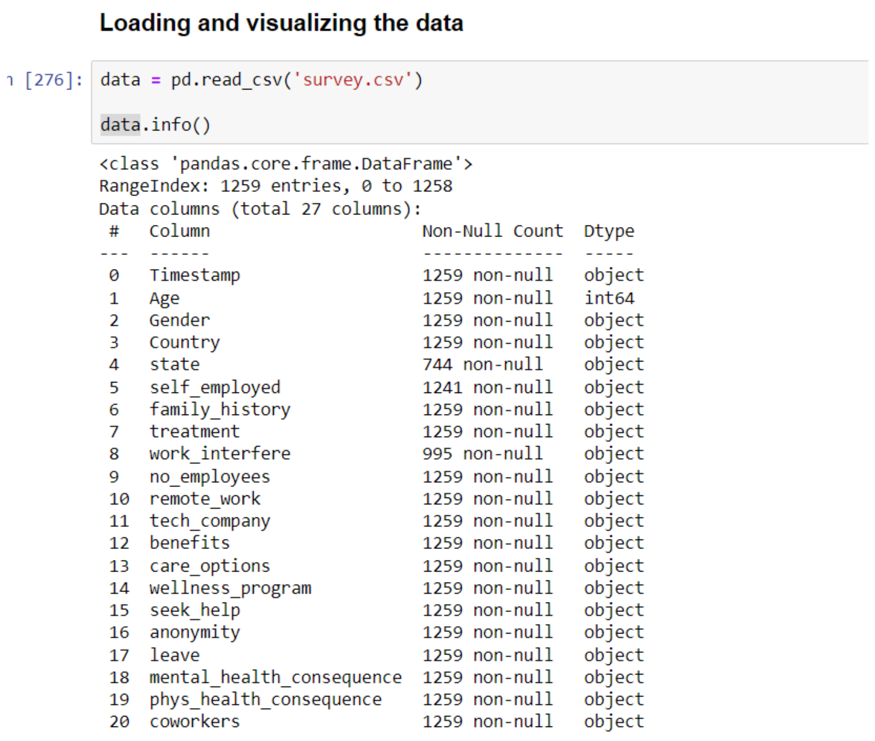
We have selected a dataset that is prepared on a survey of mental health issues on the people of different countries. Necessary inputs like family history, interference frequency in work, previously sought help or not, employment type, number of employees in the office etc. are taken as independent features. Based on these features we would try to predict whether a new person with a given set of features would be willing to go for treatment or not.

Data Analysis and Pre-processing:

Firstly, we are importing required libraries for pre-processing and looking at the data info. Then we would move on to analyzing individual independent columns.



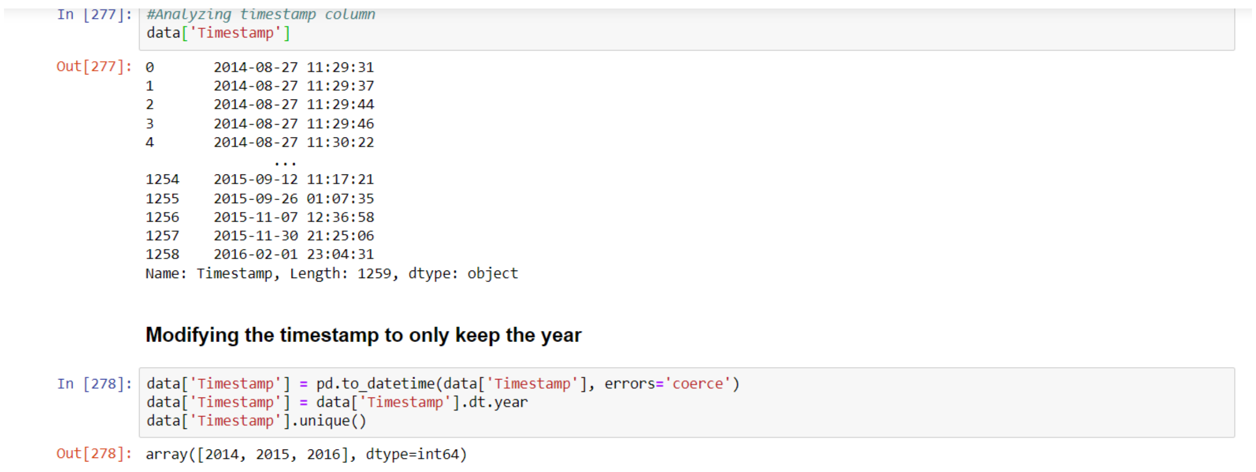
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score
#from sklearn.pandas import CategoricalImputer
from feature_engine.imputation import CategoricalImputer

import numpy as np
import random
from matplotlib import pyplot as plt

from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm

from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split function
from sklearn import metrics
```



Loading and visualizing the data

```
n [276]: data = pd.read_csv('survey.csv')

         data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1259 entries, 0 to 1258
Data columns (total 27 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Timestamp                  1259 non-null   object
 1   Age                        1259 non-null   int64
 2   Gender                     1259 non-null   object
 3   Country                    1259 non-null   object
 4   state                      744 non-null    object
 5   self_employed              1241 non-null   object
 6   family_history             1259 non-null   object
 7   treatment                  1259 non-null   object
 8   work_interfere             995 non-null    object
 9   no_employees               1259 non-null   object
 10  remote_work                1259 non-null   object
 11  tech_company               1259 non-null   object
 12  benefits                   1259 non-null   object
 13  care_options               1259 non-null   object
 14  wellness_program           1259 non-null   object
 15  seek_help                  1259 non-null   object
 16  anonymity                  1259 non-null   object
 17  leave                      1259 non-null   object
 18  mental_health_consequence  1259 non-null   object
 19  phys_health_consequence    1259 non-null   object
 20  coworkers                  1259 non-null   object
```

The first column to analyze is the timestamp. Here we can see that this column has dates. So, we took out the date field and kept only the year to use the relevant year for further analysis.



```
In [277]: #Analyzing timestamp column
          data['Timestamp']

Out[277]: 0       2014-08-27 11:29:31
          1       2014-08-27 11:29:37
          2       2014-08-27 11:29:44
          3       2014-08-27 11:29:46
          4       2014-08-27 11:30:22
                          ...
          1254    2015-09-12 11:17:21
          1255    2015-09-26 01:07:35
          1256    2015-11-07 12:36:58
          1257    2015-11-30 21:25:06
          1258    2016-02-01 23:04:31
          Name: Timestamp, Length: 1259, dtype: object
```

Modifying the timestamp to only keep the year

```
In [278]: data['Timestamp'] = pd.to_datetime(data['Timestamp'], errors='coerce')
          data['Timestamp'] = data['Timestamp'].dt.year
          data['Timestamp'].unique()

Out[278]: array([2014, 2015, 2016], dtype=int64)
```

Now we move on to age column. It can be seen that age column has some outliers. So, we have kept the age column values within an acceptable range of 12 to 100 years. We would address the nan values later when we would replace the nan values in all the columns.

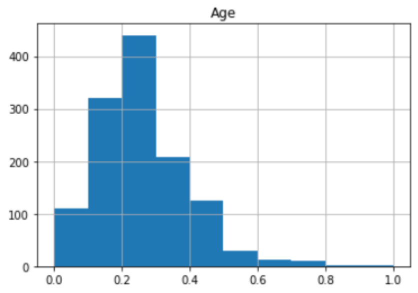
```
In [280]: data['Age'].unique()

Out[280]: array([         37,          44,          32,          31,          33,
                    35,          39,          42,          23,          29,
                    36,          27,          46,          41,          34,
                    30,          40,          38,          50,          24,
                    18,          28,          26,          22,          19,
                    25,          45,          21,         -29,          43,
                    56,          60,          54,         329,          55,
           99999999999,          48,          20,          57,          58,
                    47,          62,          51,          65,          49,
                 -1726,           5,          53,          61,           8,
                    11,          -1,          72], dtype=int64)
```

Keeping the age column within a range

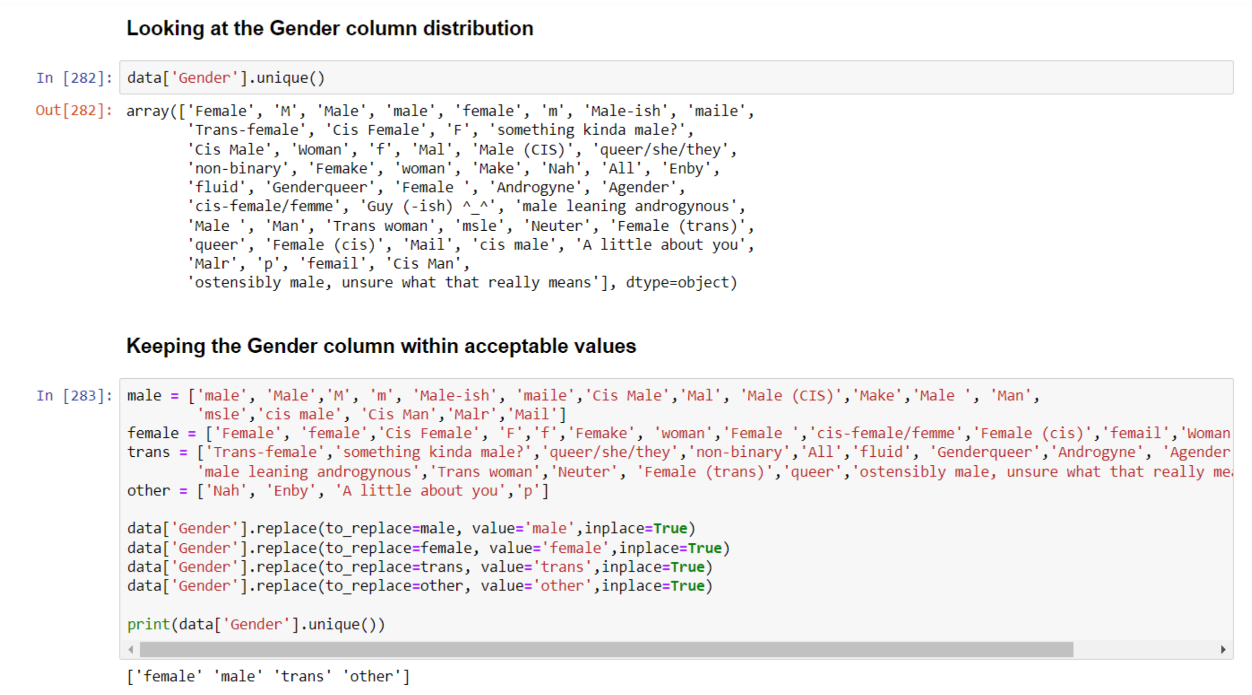
```
In [281]: irr_age = data[(data['Age'] < 12) | (data['Age'] > 100)]
          data['Age'] = data['Age'].drop(irr_age.index)
          data['Age']
          data['Age'].unique()

Out[281]: array([37., 44., 32., 31., 33., 35., 39., 42., 23., 29., 36., 27., 46.,
                 41., 34., 30., 40., 38., 50., 24., 18., 28., 26., 22., 19., 25.,
                 45., 21., nan, 43., 56., 60., 54., 55., 48., 20., 57., 58., 47.,
                 62., 51., 65., 49., 53., 61., 72.])
```

```
In [22]: # Let 'x' be the data with 1000 random points.
         x = np.random.randn(1000)
```

```
In [23]: data.hist(column='Age')

Out[23]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000001920FE22240>]],
               dtype=object)
```



Now, we move on to the gender column where data is not filled properly. We would replace the outliers with the appropriate data in this column.



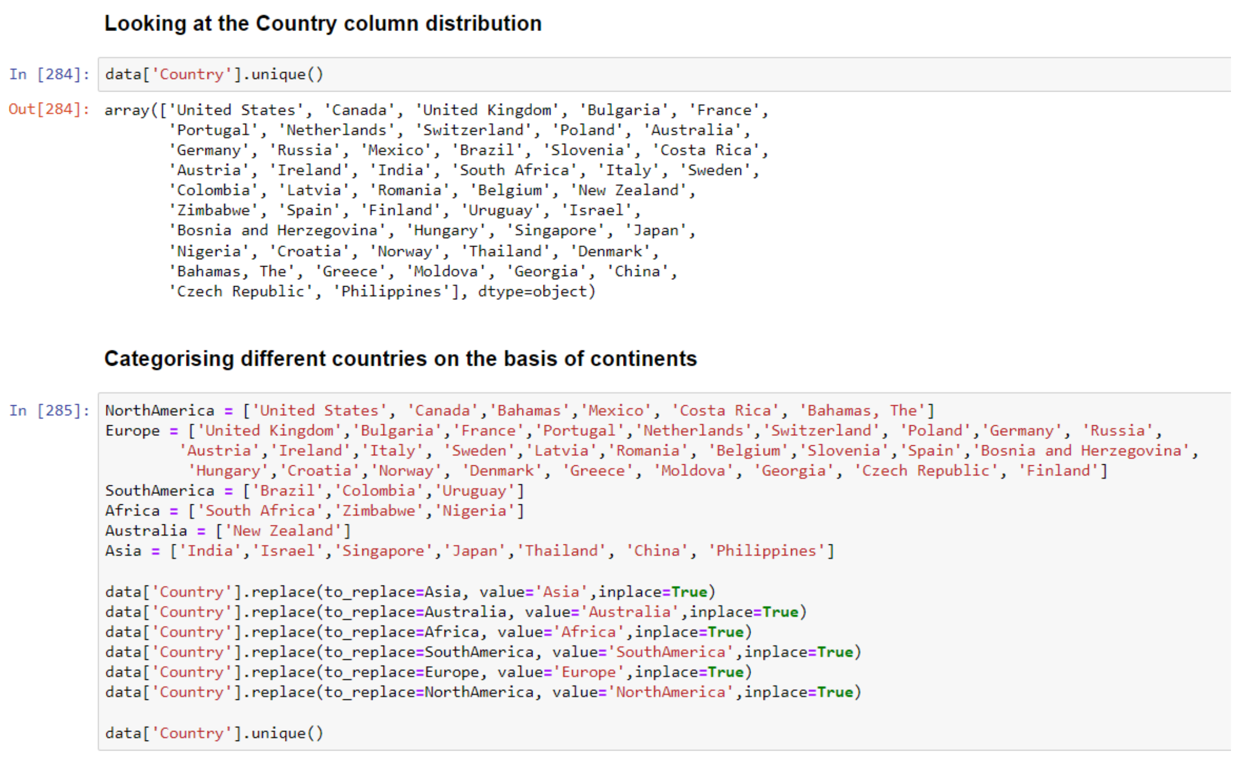
Looking at the Gender column distribution

```
In [282]: data['Gender'].unique()

Out[282]: array(['Female', 'M', 'Male', 'male', 'female', 'm', 'Male-ish', 'maile',
                 'Trans-female', 'Cis Female', 'F', 'something kinda male?',
                 'Cis Male', 'Woman', 'f', 'Mal', 'Male (CIS)', 'queer/she/they',
                 'non-binary', 'Femake', 'woman', 'Make', 'Nah', 'All', 'Enby',
                 'fluid', 'Genderqueer', 'Female ', 'Androgyne', 'Agender',
                 'cis-female/femme', 'Guy (-ish) ^_^', 'male leaning androgynous',
                 'Male ', 'Man', 'Trans woman', 'msle', 'Neuter', 'Female (trans)',
                 'queer', 'Female (cis)', 'Mail', 'cis male', 'A little about you',
                 'Malr', 'p', 'femail', 'Cis Man',
                 'ostensibly male, unsure what that really means'], dtype=object)
```

Keeping the Gender column within acceptable values

```
In [283]: male = ['male', 'Male','M', 'm', 'Male-ish', 'maile','Cis Male','Mal', 'Male (CIS)','Make','Male ', 'Man',
                  'msle','cis male', 'Cis Man','Malr','Mail']
          female = ['Female', 'female','Cis Female', 'F','f','Femake', 'woman','Female ','cis-female/femme','Female (cis)','femail','Woman
          trans = ['Trans-female','something kinda male?','queer/she/they','non-binary','All','fluid', 'Genderqueer','Androgyne', 'Agender
                  'male leaning androgynous','Trans woman','Neuter', 'Female (trans)','queer','ostensibly male, unsure what that really me
          other = ['Nah', 'Enby', 'A little about you','p']

          data['Gender'].replace(to_replace=male, value='male',inplace=True)
          data['Gender'].replace(to_replace=female, value='female',inplace=True)
          data['Gender'].replace(to_replace=trans, value='trans',inplace=True)
          data['Gender'].replace(to_replace=other, value='other',inplace=True)

          print(data['Gender'].unique())

          ['female' 'male' 'trans' 'other']
```

The column country has a large number of values. So this column would account for large variance in the training phase. That’s why we are categorizing the countries on the basis of continents.



Looking at the Country column distribution

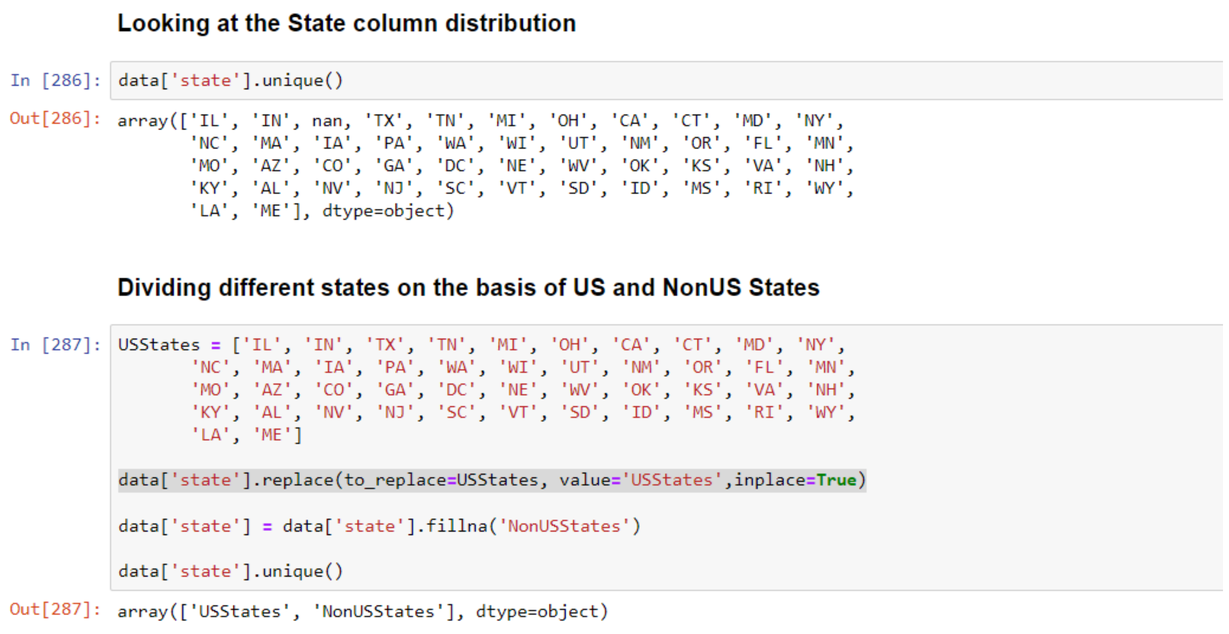
```
In [284]: data['Country'].unique()
```

```
Out[284]: array(['United States', 'Canada', 'United Kingdom', 'Bulgaria', 'France',
                 'Portugal', 'Netherlands', 'Switzerland', 'Poland', 'Australia',
                 'Germany', 'Russia', 'Mexico', 'Brazil', 'Slovenia', 'Costa Rica',
                 'Austria', 'Ireland', 'India', 'South Africa', 'Italy', 'Sweden',
                 'Colombia', 'Latvia', 'Romania', 'Belgium', 'New Zealand',
                 'Zimbabwe', 'Spain', 'Finland', 'Uruguay', 'Israel',
                 'Bosnia and Herzegovina', 'Hungary', 'Singapore', 'Japan',
                 'Nigeria', 'Croatia', 'Norway', 'Thailand', 'Denmark',
                 'Bahamas, The', 'Greece', 'Moldova', 'Georgia', 'China',
                 'Czech Republic', 'Philippines'], dtype=object)
```

Categorising different countries on the basis of continents

```
In [285]: NorthAmerica = ['United States', 'Canada','Bahamas','Mexico', 'Costa Rica', 'Bahamas, The']
          Europe = ['United Kingdom','Bulgaria','France','Portugal','Netherlands','Switzerland', 'Poland','Germany', 'Russia',
                  'Austria','Ireland','Italy', 'Sweden','Latvia','Romania', 'Belgium','Slovenia','Spain','Bosnia and Herzegovina',
                   'Hungary','Croatia','Norway', 'Denmark', 'Greece', 'Moldova', 'Georgia', 'Czech Republic', 'Finland']
          SouthAmerica = ['Brazil','Colombia','Uruguay']
          Africa = ['South Africa','Zimbabwe','Nigeria']
          Australia = ['New Zealand']
          Asia = ['India','Israel','Singapore','Japan','Thailand', 'China', 'Philippines']

          data['Country'].replace(to_replace=Asia, value='Asia',inplace=True)
          data['Country'].replace(to_replace=Australia, value='Australia',inplace=True)
          data['Country'].replace(to_replace=Africa, value='Africa',inplace=True)
          data['Country'].replace(to_replace=SouthAmerica, value='SouthAmerica',inplace=True)
          data['Country'].replace(to_replace=Europe, value='Europe',inplace=True)
          data['Country'].replace(to_replace=NorthAmerica, value='NorthAmerica',inplace=True)

          data['Country'].unique()
```

Now, the state column only has values if the subject is from USA. That’s why we are keeping only two unique values in this column as USState and NonUSState.



Looking at the State column distribution

```
In [286]: data['state'].unique()
```

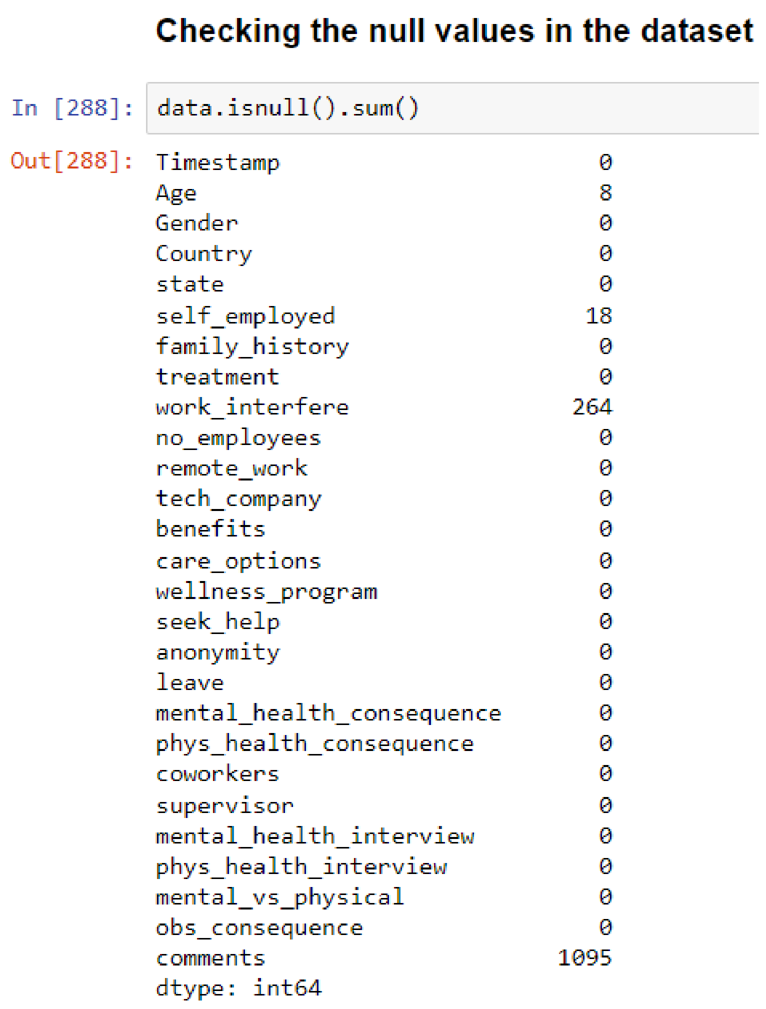
```
Out[286]: array(['IL', 'IN', nan, 'TX', 'TN', 'MI', 'OH', 'CA', 'CT', 'MD', 'NY',
                 'NC', 'MA', 'IA', 'PA', 'WA', 'WI', 'UT', 'NM', 'OR', 'FL', 'MN',
                 'MO', 'AZ', 'CO', 'GA', 'DC', 'NE', 'WV', 'OK', 'KS', 'VA', 'NH',
                 'KY', 'AL', 'NV', 'NJ', 'SC', 'VT', 'SD', 'ID', 'MS', 'RI', 'WY',
                 'LA', 'ME'], dtype=object)
```

Dividing different states on the basis of US and NonUS States

```
In [287]: USStates = ['IL', 'IN', 'TX', 'TN', 'MI', 'OH', 'CA', 'CT', 'MD', 'NY',
                 'NC', 'MA', 'IA', 'PA', 'WA', 'WI', 'UT', 'NM', 'OR', 'FL', 'MN',
                 'MO', 'AZ', 'CO', 'GA', 'DC', 'NE', 'WV', 'OK', 'KS', 'VA', 'NH',
                 'KY', 'AL', 'NV', 'NJ', 'SC', 'VT', 'SD', 'ID', 'MS', 'RI', 'WY',
                 'LA', 'ME']

          data['state'].replace(to_replace=USStates, value='USStates',inplace=True)

          data['state'] = data['state'].fillna('NonUSStates')

          data['state'].unique()
```

```
Out[287]: array(['USStates', 'NonUSStates'], dtype=object)
```

Now, we move on to checking the null values in the dataset. We could see the age, self_employed, work_interfere and comments columns have null values. We would replace the null values of the age column by its median since age is a numerical column. We would then replace the null values of self_employed, work_interfere by their mode since they are categorical values. Finally, we would drop the comments column as that field was optional in the survey and it does not provide any significance to the output determination.



Checking the null values in the dataset

```
In [288]: data.isnull().sum()

Out[288]: Timestamp                    0
          Age                          8
          Gender                       0
          Country                      0
          state                        0
          self_employed               18
          family_history               0
          treatment                    0
          work_interfere             264
          no_employees                 0
          remote_work                  0
          tech_company                 0
          benefits                     0
          care_options                 0
          wellness_program             0
          seek_help                    0
          anonymity                    0
          leave                        0
          mental_health_consequence    0
          phys_health_consequence      0
          coworkers                    0
          supervisor                   0
          mental_health_interview      0
          phys_health_interview        0
          mental_vs_physical           0
          obs_consequence              0
          comments                  1095
          dtype: int64
```

Removing the comments field from the dataset as it was optional in the survey

```
In [290]: data.drop('comments', axis=1, inplace=True)
          data.isnull().sum()
```

```
Out[290]: Timestamp                    0
          Age                          0
          Gender                       0
          Country                      0
          state                        0
          self_employed                0
          family_history               0
          treatment                    0
          work_interfere               0
          no_employees                 0
          remote_work                  0
          tech_company                 0
          benefits                     0
          care_options                 0
          wellness_program             0
          seek_help                    0
          anonymity                    0
          leave                        0
          mental_health_consequence    0
          phys_health_consequence      0
          coworkers                    0
          supervisor                   0
          mental_health_interview      0
          phys_health_interview        0
          mental_vs_physical           0
          obs_consequence              0
          dtype: int64
```

The age and timestamp fields have large difference in their distribution. Because timestamp only ranges from 2014 to 2016 whereas age ranges from 12 to 100. So we have to scale these features by normalization so that the models can reduce their cost functions optimally.

Feature Scaling for the age column

```
In [291]: scaler = MinMaxScaler()
          data['Age'] = scaler.fit_transform(data[['Age']])
          data['Age']
```

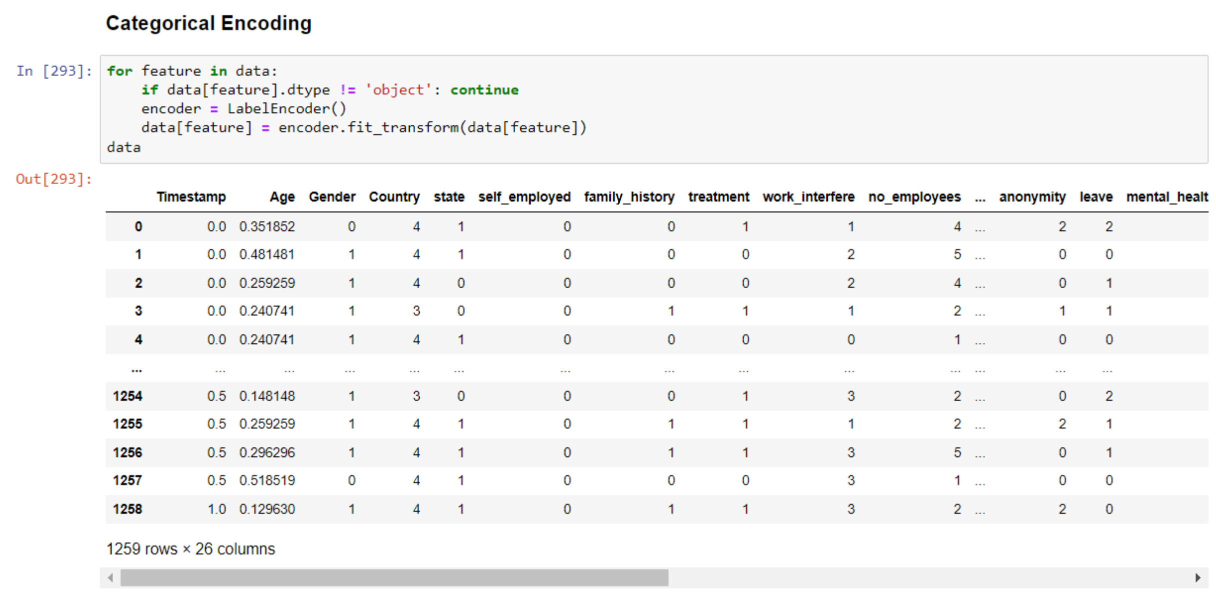
```
Out[291]: 0       0.351852
          1       0.481481
          2       0.259259
          3       0.240741
          4       0.240741
                    ...
          1254    0.148148
          1255    0.259259
          1256    0.296296
          1257    0.518519
          1258    0.129630
          Name: Age, Length: 1259, dtype: float64
```

Feature Scaling for the timestamp column

```
In [292]: scaler = MinMaxScaler()
          data['Timestamp'] = scaler.fit_transform(data[['Timestamp']])
          data['Timestamp']
```

```
Out[292]: 0       0.0
          1       0.0
          2       0.0
          3       0.0
          4       0.0
                 ...
          1254    0.5
          1255    0.5
          1256    0.5
          1257    0.5
          1258    1.0
          Name: Timestamp, Length: 1259, dtype: float64
```

Apart from age and timestamp, all other features are categorical values. So we are encoding to numerical values.



Categorical Encoding

In [293]:
```
for feature in data:
    if data[feature].dtype != 'object': continue
    encoder = LabelEncoder()
    data[feature] = encoder.fit_transform(data[feature])
data
```

Out[293]:

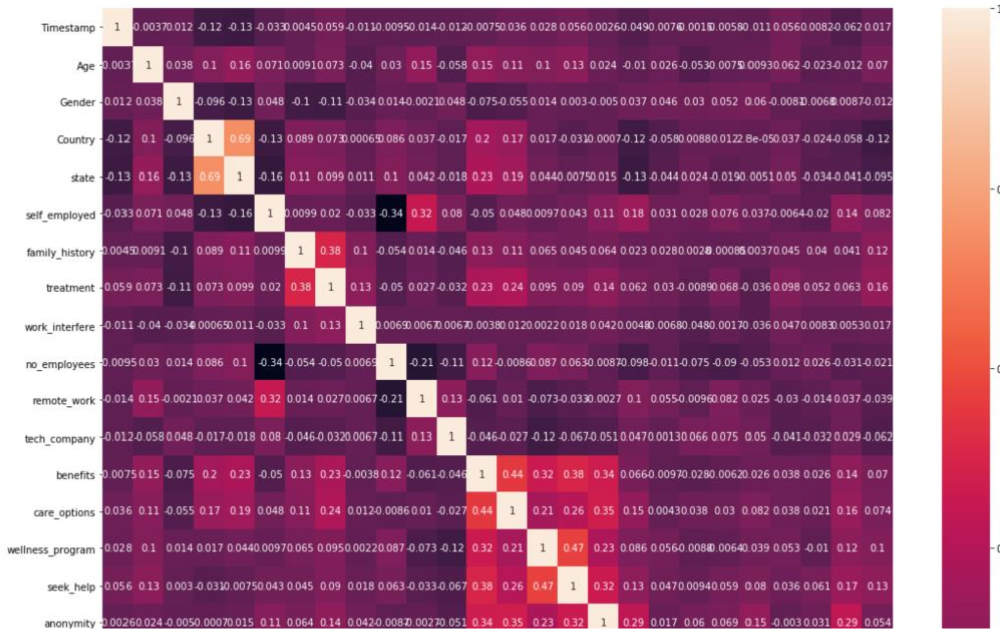
	Timestamp	Age	Gender	Country	state	self_employed	family_history	treatment	work_interfere	no_employees	...	anonymity	leave	mental_healt
0	0.0	0.351852	0	4	1	0	0	1	1	4	...	2	2	
1	0.0	0.481481	1	4	1	0	0	0	2	5	...	0	0	
2	0.0	0.259259	1	4	0	0	0	0	2	4	...	0	1	
3	0.0	0.240741	1	3	0	0	1	1	1	2	...	1	1	
4	0.0	0.240741	1	4	1	0	0	0	0	1	...	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1254	0.5	0.148148	1	3	0	0	0	1	3	2	...	0	2	
1255	0.5	0.259259	1	4	1	0	1	1	1	2	...	2	1	
1256	0.5	0.296296	1	4	1	0	1	1	3	5	...	0	1	
1257	0.5	0.518519	0	4	1	0	0	0	3	1	...	0	0	
1258	1.0	0.129630	1	4	1	0	1	1	3	2	...	2	0	

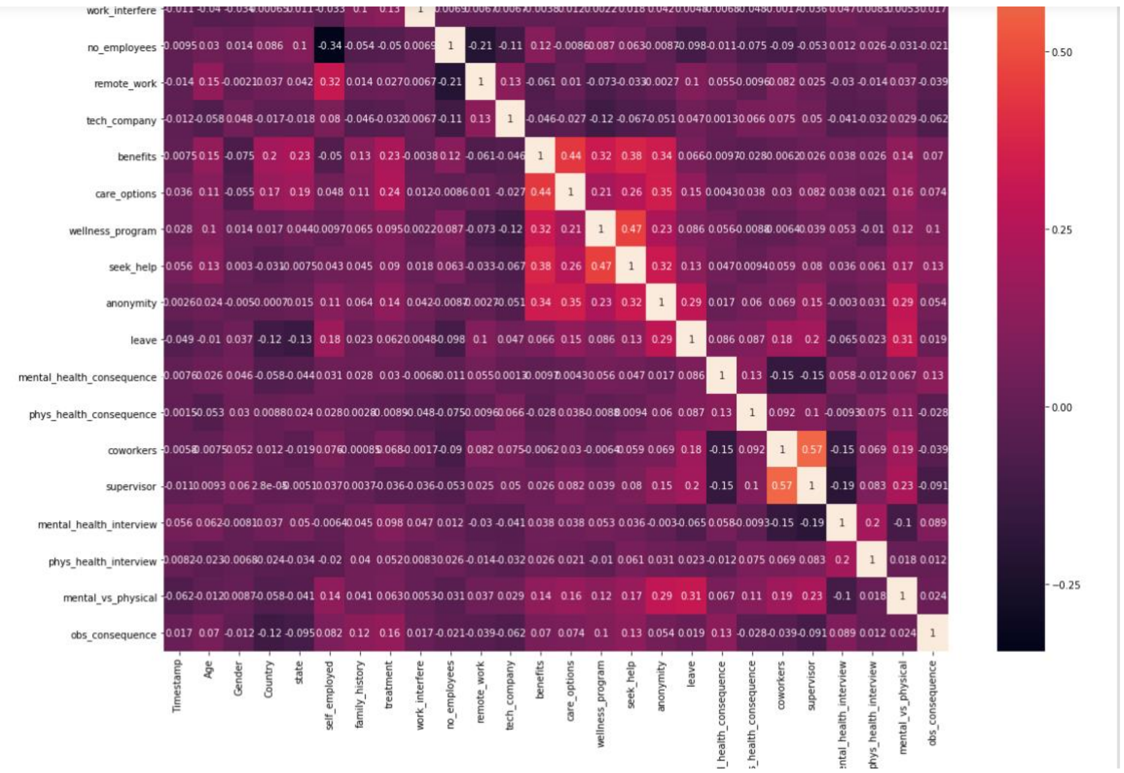
1259 rows × 26 columns

Next, we move onto plotting the co-relation matrix to find out the relations of variables among one another. From the heatmap of the matrix, we could easily find out the five top features which are co-related with the treatment variable. These are ‘family_history’, ‘care_options’, ‘benefits’, ‘seek_help’ and ‘obs_consequence’ with co-relation scores 0.38, 0.24, 0.23, 0.14, 0.13 respectively.

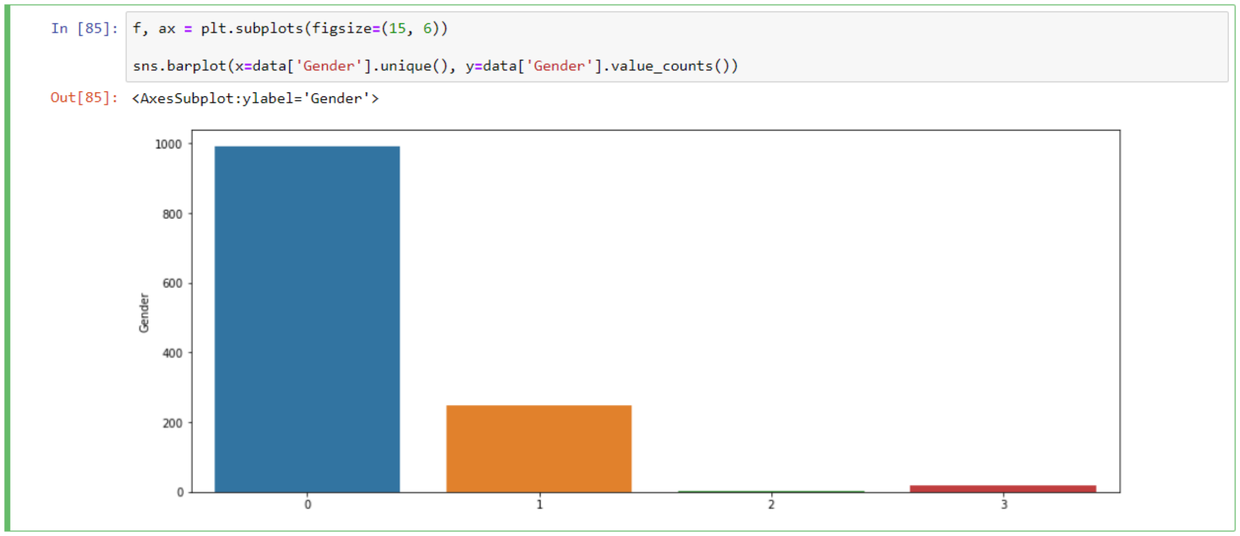
Co-relation matrix and heatmap

```
In [294]: corr = data.corr()
          f, ax = plt.subplots(figsize=(18, 18))
          sns.heatmap(corr, annot = True)

Out[294]: <matplotlib.axes._subplots.AxesSubplot at 0x1c1e2c16128>
```



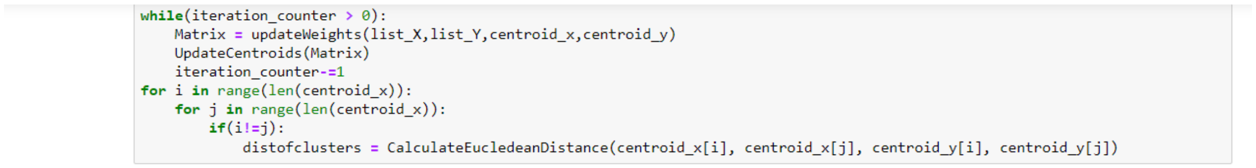


After categorical encoding, gender distribution is shown as a reference.

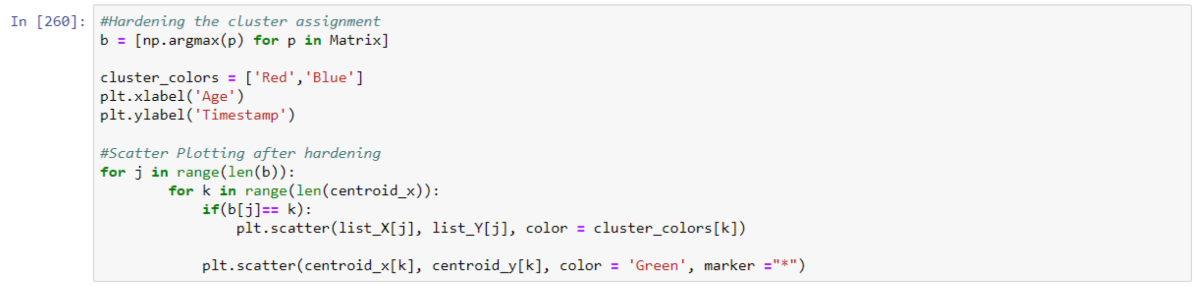

```
In [85]: f, ax = plt.subplots(figsize=(15, 6))

         sns.barplot(x=data['Gender'].unique(), y=data['Gender'].value_counts())

Out[85]: <AxesSubplot:ylabel='Gender'>
```

Application of Algorithms:

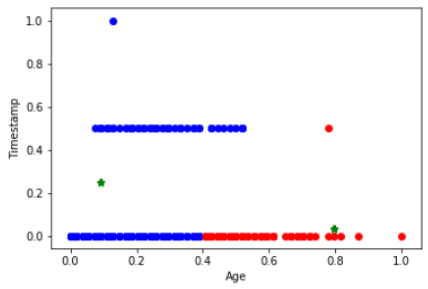
Fuzzy C: Among the independent features, only Age and Timestamp are continuous numerical values. So, we decided to implement Fuzzy C with these features to see how the values cluster together in these columns. We tried with different values of K, but K = 2 gave less the least overlapping clusters. Moreover, Dunn Index was the highest at K = 2 compared to K = 3,4,5. We have used the same process of implementation of fuzzy clustering as we did for the assignment 1. Since both age and timestamp columns are normalized, points remained mostly within a range in the scatter plot.



```
while(iteration_counter > 0):
    Matrix = updateWeights(list_X,list_Y,centroid_x,centroid_y)
    UpdateCentroids(Matrix)
    iteration_counter-=1
for i in range(len(centroid_x)):
    for j in range(len(centroid_x)):
        if(i!=j):
            distofclusters = CalculateEucledeanDistance(centroid_x[i], centroid_x[j], centroid_y[i], centroid_y[j])
```



```
In [260]: #Hardening the cluster assignment
          b = [np.argmax(p) for p in Matrix]

          cluster_colors = ['Red','Blue']
          plt.xlabel('Age')
          plt.ylabel('Timestamp')

          #Scatter Plotting after hardening
          for j in range(len(b)):
                  for k in range(len(centroid_x)):
                      if(b[j]== k):
                          plt.scatter(list_X[j], list_Y[j], color = cluster_colors[k])

                      plt.scatter(centroid_x[k], centroid_y[k], color = 'Green', marker ="*")
```



KNN: Next, we moved on to K-Nearest Neighbors for classification. When number of neighbors was fixed to 2, KNN didn't do a good job with only 56% accuracy. That's why we run the KNN with K values from 1 to 30. Then the errors at each K value was plotted in a graph against K. It was found out that K = 25 provided the least error. That's why we changed the K to 25 and accuracy increased by 12%.

KNN Implementation

```
In [302]: X = data.drop(columns=['treatment'])
          y = data['treatment']

          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

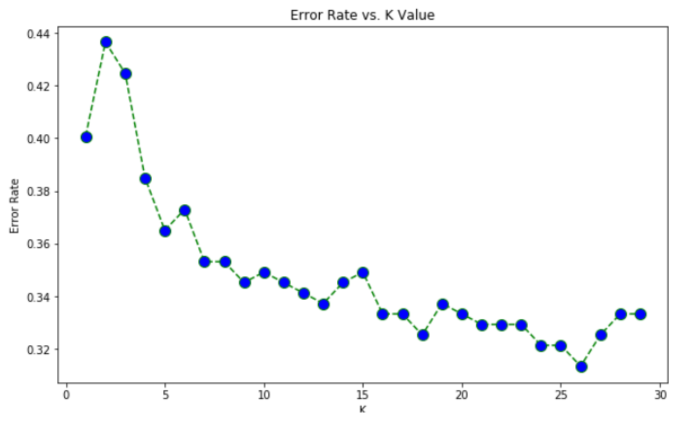
```
In [319]: knn = KNeighborsClassifier(n_neighbors=2)
          knn.fit(X_train,y_train)

          y_predicted = knn.predict(X_test)
          accuracy = '{:.0%}'.format(accuracy_score(y_test, y_predicted))

          print(f'K nearest neighbors model accuracy: {accuracy}')

          precision = '{:.0%}'.format(precision_score(y_test, y_predicted))

          print(f'K nearest neighbors model precision: {precision}')

          recall = '{:.0%}'.format(recall_score(y_test, y_predicted))

          print(f'K nearest neighbors model recall: {recall}')
```

```
K nearest neighbors model accuracy: 56%
K nearest neighbors model precision: 63%
K nearest neighbors model recall: 39%
```

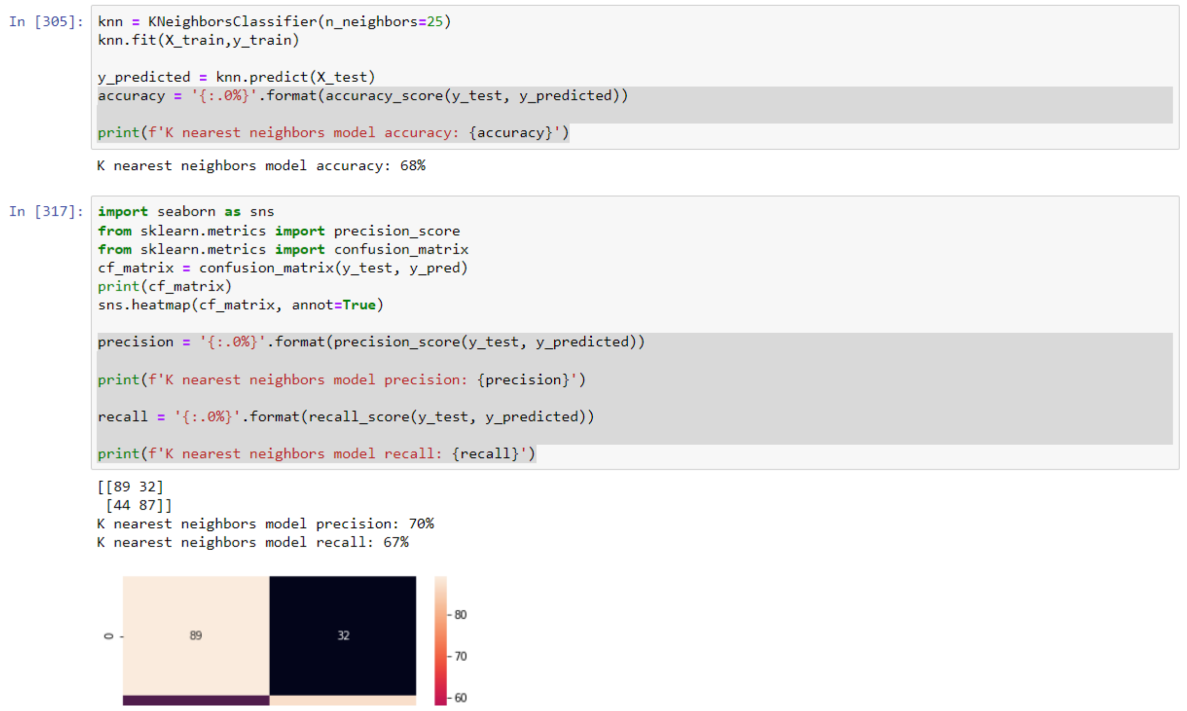
```
In [304]: error_rate = []
          for i in range(1,30):
              knn = KNeighborsClassifier(n_neighbors=i)
              knn.fit(X_train,y_train)
              pred_i = knn.predict(X_test)
              error_rate.append(np.mean(pred_i != y_test))

          plt.figure(figsize=(10,6))
          plt.plot(range(1,30),error_rate,color='green', linestyle='dashed',
                   marker='o',markerfacecolor='blue', markersize=10)
          plt.title('Error Rate vs. K Value')
```

```
In [304]: error_rate = []
          for i in range(1,30):
              knn = KNeighborsClassifier(n_neighbors=i)
              knn.fit(X_train,y_train)
              pred_i = knn.predict(X_test)
              error_rate.append(np.mean(pred_i != y_test))

          plt.figure(figsize=(10,6))
          plt.plot(range(1,30),error_rate,color='green', linestyle='dashed',
                   marker='o',markerfacecolor='blue', markersize=10)
          plt.title('Error Rate vs. K Value')
          plt.xlabel('K')
          plt.ylabel('Error Rate')
          print("Minimum error:-",min(error_rate),"at K =",error_rate.index(min(error_rate)))
```

```
Minimum error:- 0.3134920634920635 at K = 25
```



Next, we would use the confusion matrix to figure out how the KNN has performed on the test samples. From the confusion matrix, it is clear that the KNN model is able to predict more positive samples than negative, which is why it has a good precision rate when K = 25. Precision score improved by 7% and recall by a huge margin of 31%.



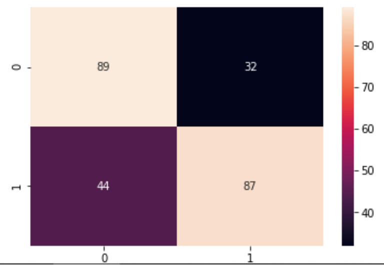
```
In [305]: knn = KNeighborsClassifier(n_neighbors=25)
          knn.fit(X_train,y_train)

          y_predicted = knn.predict(X_test)
          accuracy = '{:.0%}'.format(accuracy_score(y_test, y_predicted))

          print(f'K nearest neighbors model accuracy: {accuracy}')

          K nearest neighbors model accuracy: 68%

In [317]: import seaborn as sns
          from sklearn.metrics import precision_score
          from sklearn.metrics import confusion_matrix
          cf_matrix = confusion_matrix(y_test, y_pred)
          print(cf_matrix)
          sns.heatmap(cf_matrix, annot=True)

          precision = '{:.0%}'.format(precision_score(y_test, y_predicted))

          print(f'K nearest neighbors model precision: {precision}')

          recall = '{:.0%}'.format(recall_score(y_test, y_predicted))

          print(f'K nearest neighbors model recall: {recall}')

          [[89 32]
           [44 87]]
          K nearest neighbors model precision: 70%
          K nearest neighbors model recall: 67%
```

Naive Bayes: We decided to implement Naive Bayes as taught in the class to compare with the KNN performance. From the confusion matrix, it is clear to us that the Gaussian Naive Bayes was able to predict more false classes than positive. In terms of accuracy, Naive Bayes performs better than KNN with K = 25, while precision and recall scores are way better when compared with KNN with K = 2.

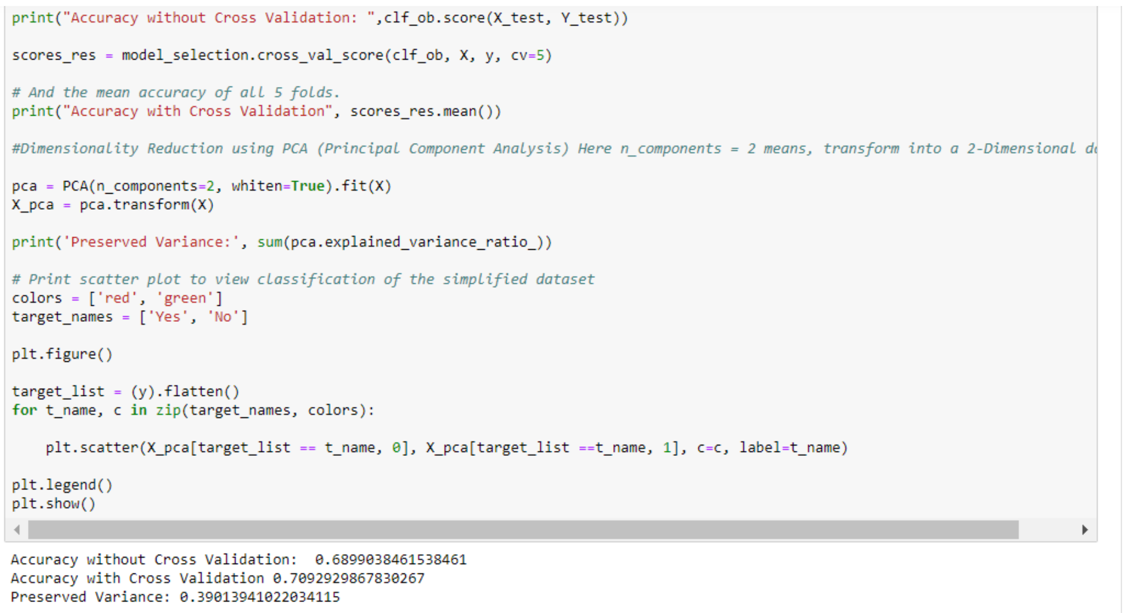
Naive Bayes Implementation

```
In [318]: from sklearn.naive_bayes import GaussianNB
          from sklearn import metrics
          from sklearn.metrics import recall_score

          gnb = GaussianNB()
          gnb.fit(X_train, y_train)
          y_pred = gnb.predict(X_test)


          print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

          cf_matrix = confusion_matrix(y_test, y_pred)
          print("\nConfusion Matrix: ",cf_matrix)

          sns.heatmap(cf_matrix, annot=True)

          precision = '{:.0%}'.format(precision_score(y_test, y_predicted))

          print(f'NV model precision: {precision}')

          recall = '{:.0%}'.format(recall_score(y_test, y_predicted))

          print(f'NV model recall: {recall}')
```

```
Accuracy: 0.6984126984126984

Confusion Matrix:  [[89 32]
 [44 87]]
NV model precision: 70%
NV model recall: 67%
```

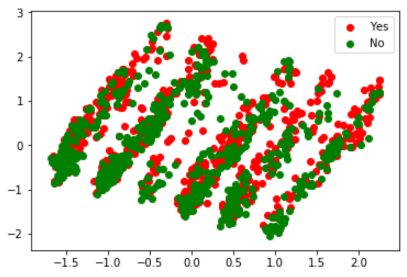


SVM: We have used all the independent features and the dependent treatment column for the SVM with an accuracy of 68%. Then we used K-Fold cross validation as discussed in the class with K = 5 to split the data into random 5 sets of train and test sets. In this way, we tried to develop a robust train test set for the SVM to work on. It can be seen that accuracy improved with K-Fold cross validation. As SVM determines a hyperplane with all the features used, we had to reduce the dimension to visualize it. That's where PCA came into play. So, K-Fold cross validation was implemented on SVM with a 'rbf' kernel for better margin determination. Then, the results were plotted in 2D scatter plot by PCA dimension reduction.

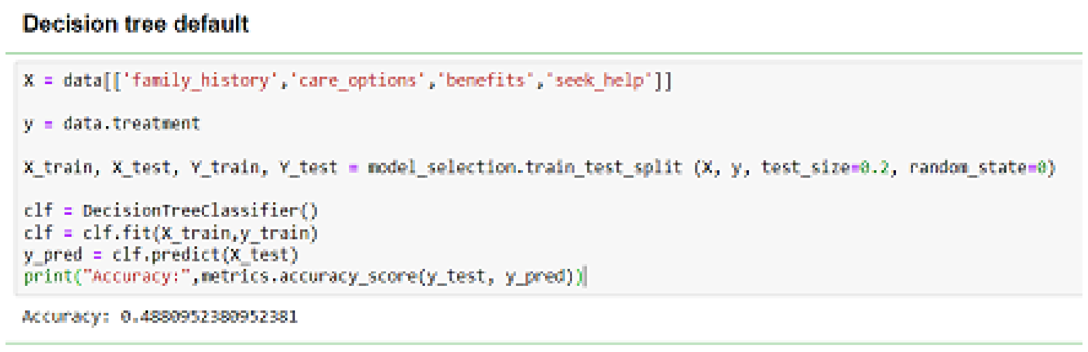


```
print("Accuracy without Cross Validation: ",clf_ob.score(X_test, Y_test))

scores_res = model_selection.cross_val_score(clf_ob, X, y, cv=5)

# And the mean accuracy of all 5 folds.
print("Accuracy with Cross Validation", scores_res.mean())

#Dimensionality Reduction using PCA (Principal Component Analysis) Here n_components = 2 means, transform into a 2-Dimensional d

pca = PCA(n_components=2, whiten=True).fit(X)
X_pca = pca.transform(X)

print('Preserved Variance:', sum(pca.explained_variance_ratio_))

# Print scatter plot to view classification of the simplified dataset
colors = ['red', 'green']
target_names = ['Yes', 'No']

plt.figure()

target_list = (y).flatten()
for t_name, c in zip(target_names, colors):

    plt.scatter(X_pca[target_list == t_name, 0], X_pca[target_list ==t_name, 1], c=c, label=t_name)

plt.legend()
plt.show()
```

```
Accuracy without Cross Validation:  0.6899038461538461
Accuracy with Cross Validation 0.7092929867830267
Preserved Variance: 0.39013941022034115
```



Decision Tree: We wanted to go for a less computationally expensive algorithm with a different approach than distance vectors and hyperplanes. Decision trees split the features by nodes which in turn form a tree to determine a new prediction. We thought this algorithm would be suited for out dataset because most of the features are categorical 'yes/no' values which can be easily implemented through a decision tree. At first we implemented decision tree on Scikit learns default parameters of gini impurity and none selected as maximum depth.

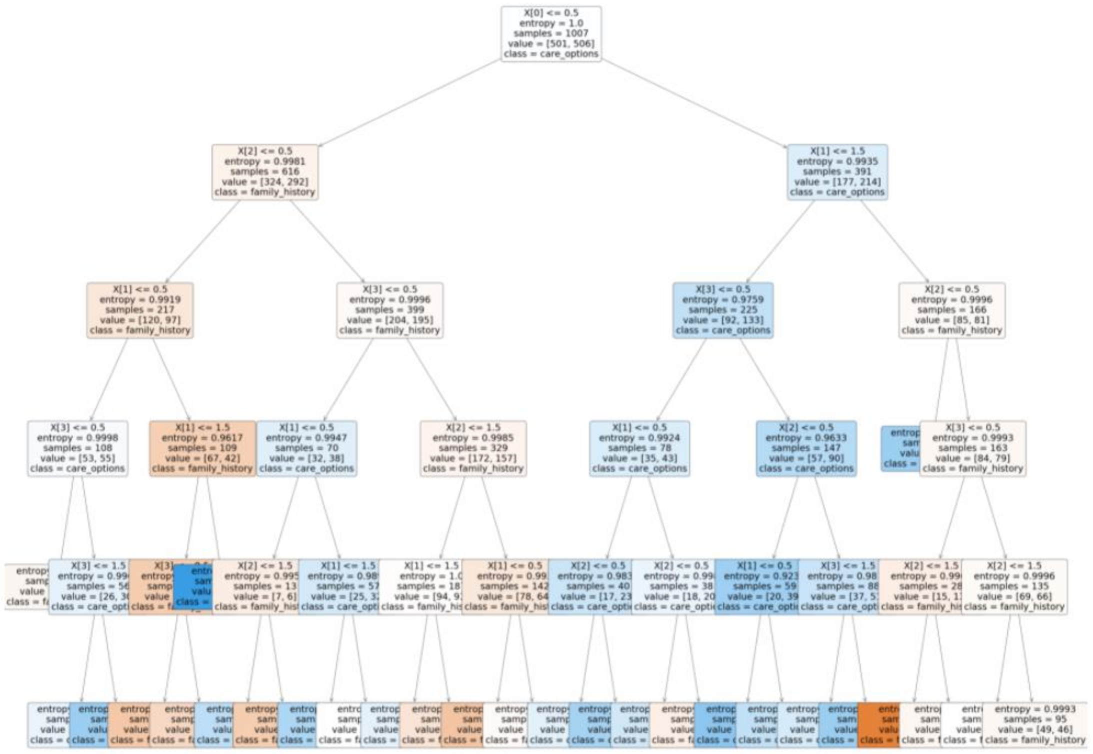


Decision tree default

```
X = data[['family_history','care_options','benefits','seek_help']]

y = data.treatment

X_train, X_test, Y_train, Y_test = model_selection.train_test_split (X, y, test_size=0.2, random_state=0)

clf = DecisionTreeClassifier()
clf = clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.4880952380952381
```

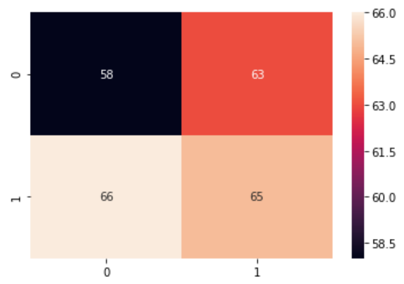
Then we changed into information gain approach of decision trees that it using entropy so as to minimize the disorder of the independent features with the target. As selecting a large value for max_depth can lead to overfitting, we selected max_depth within a reasonable bound of 5. From the heatmap of the correlation matrix, the columns ‘family_history’, ‘care_options’, ‘benefits’, ‘seek_help’ are the four columns which co-relate most with treatment. So we selected these four columns as a subset of 25 independent features and run the Decision tree with tuned parameters.



Next, we move on to the confusion matrix analysis of the decision tree. Here the values of true positive, false positive and false negative are very close to each other. The recall score of the model is low compared to the previous models because of high false negative value. So, it can be termed as Decision Tree has struggled more compared to KNN and Naive Bayes in terms of correctly identifying both ‘Yes’ and ‘No’ values of treatment column.

```
print(f'NV model recall: {recall}')
```

```
Confusion Matrix:  [[58 63]
 [66 65]]
NV model precision: 63%
NV model recall: 39%
```

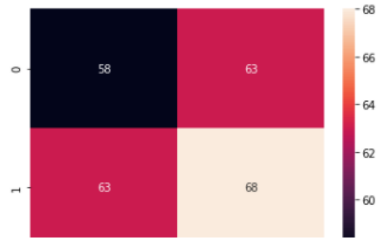


Random Forest: Random Forest would select the majority votes from the decision trees to find out the prediction of the target variable. We run the random forest with the same parameters as decision tree and the accuracy score improved than the optimal decision tree. Analyzing the confusion matrix, we would see that the Random forest was able to improve upon misclassification in terms of false negative and also improve the classification accuracy in terms of true positive.

Random Forest implementation

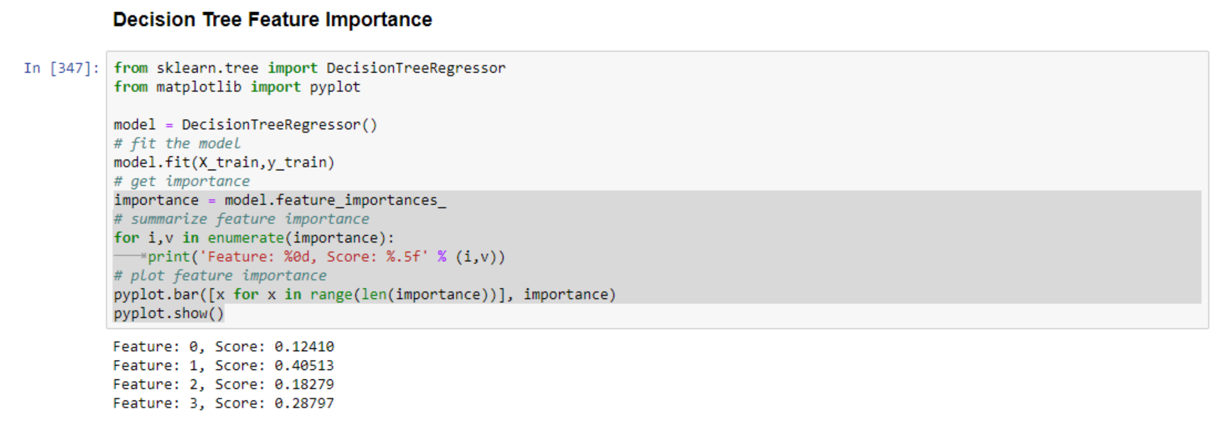
```
In [365]: from sklearn.ensemble import RandomForestClassifier

          X = data[['family_history','care_options','benefits','seek_help']]

          y = data.treatment

          X_train, X_test, Y_train, Y_test = model_selection.train_test_split (X, y, test_size=0.2, random_state=0)

          model = RandomForestClassifier(criterion="entropy", max_depth=5)

          model.fit(X_train,y_train)
          y_pred = model.predict(X_test)

          print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

          cf_matrix = confusion_matrix(y_test, y_pred)
          print("\nConfusion Matrix: ",cf_matrix)

          sns.heatmap(cf_matrix, annot=True)

          precision = '{:.0%}'.format(precision_score(y_test, y_predicted))

          print(f'RF model precision: {precision}')

          recall = '{:.0%}'.format(recall_score(y_test, y_predicted))

          print(f'RF model recall: {recall}')
```

```
Accuracy: 0.5

Confusion Matrix:  [[58 63]
 [63 68]]
RF model precision: 63%
RF model recall: 39%
```



Analysis of Decision Tree vs Random Forest on feature importance:

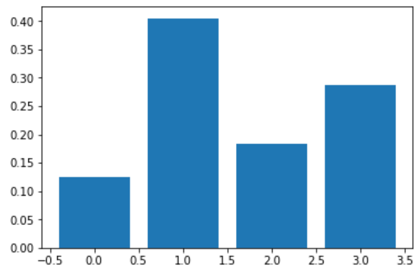
We have analyzed that decision trees randomly assign more or less importance to independent features which results in low accuracy. But if we look at the importance provided by random forest classifier on the 4 features chosen from the heatmap, we would see less variance than that of decision tree.

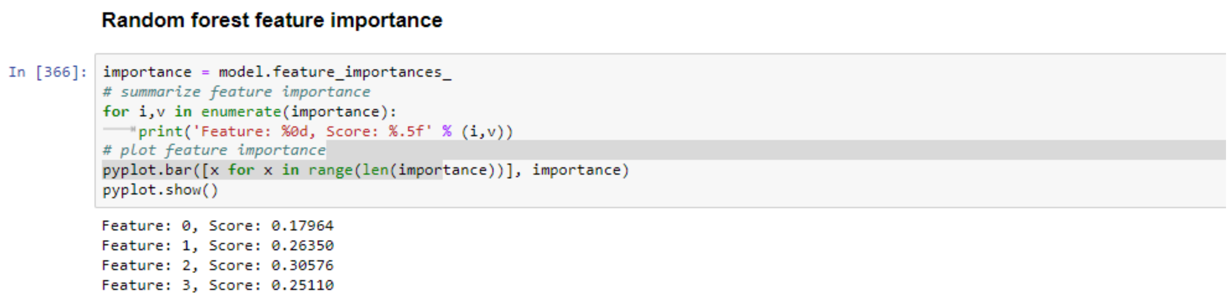


Decision Tree Feature Importance

```
In [347]: from sklearn.tree import DecisionTreeRegressor
          from matplotlib import pyplot

          model = DecisionTreeRegressor()
          # fit the model
          model.fit(X_train,y_train)
          # get importance
          importance = model.feature_importances_
          # summarize feature importance
          for i,v in enumerate(importance):
              print('Feature: %0d, Score: %.5f' % (i,v))
          # plot feature importance
          pyplot.bar([x for x in range(len(importance))], importance)
          pyplot.show()
```

```
Feature: 0, Score: 0.12410
Feature: 1, Score: 0.40513
Feature: 2, Score: 0.18279
Feature: 3, Score: 0.28797
```

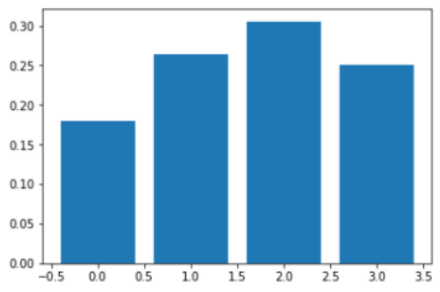




Random forest feature importance

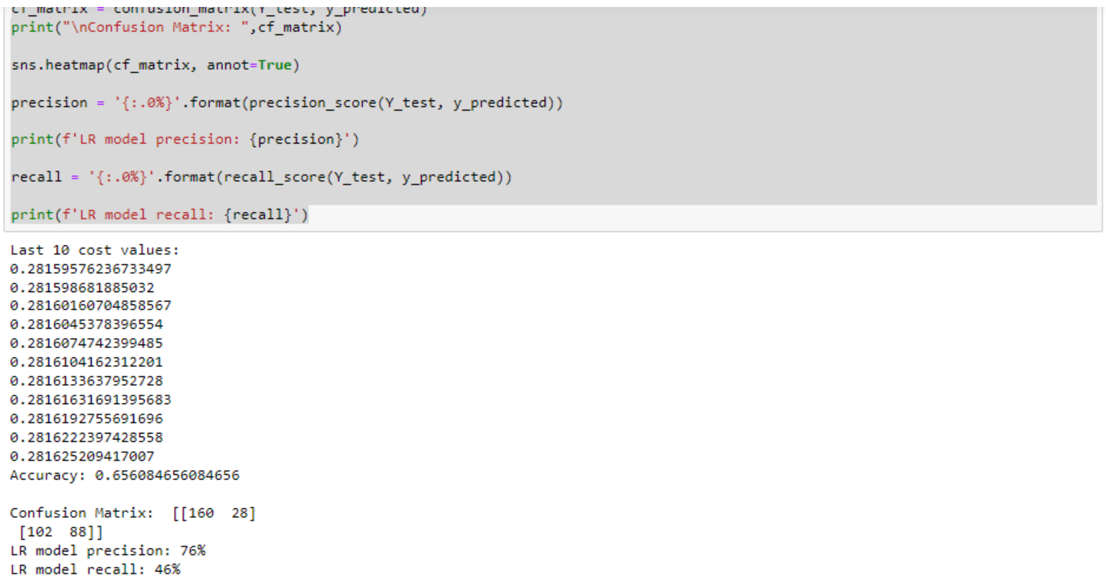
```
In [366]: importance = model.feature_importances_
          # summarize feature importance
          for i,v in enumerate(importance):
              print('Feature: %0d, Score: %.5f' % (i,v))
          # plot feature importance
          pyplot.bar([x for x in range(len(importance))], importance)
          pyplot.show()
```

```
Feature: 0, Score: 0.17964
Feature: 1, Score: 0.26350
Feature: 2, Score: 0.30576
Feature: 3, Score: 0.25110
```



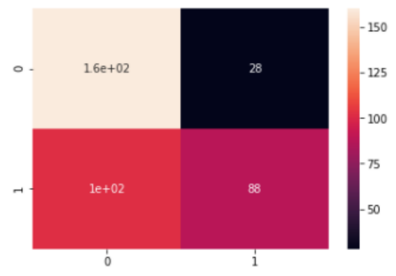
Logistic Regression:

We have implemented the Logistic regression as we did for assignment 2 with regularization, changed cost function, feature scaling and selection of dominant features. For this reason, we have found quite an improvement over the accuracy and the precision scores as seen in the figure below. Because of large false negative prediction, LR model’s recall was quite low compared to its precision.



```
cf_matrix = confusion_matrix(Y_test, y_predicted)
print("\nConfusion Matrix: ",cf_matrix)

sns.heatmap(cf_matrix, annot=True)

precision = '{:.0%}'.format(precision_score(Y_test, y_predicted))

print(f'LR model precision: {precision}')

recall = '{:.0%}'.format(recall_score(Y_test, y_predicted))

print(f'LR model recall: {recall}')
```

```
Last 10 cost values:
0.28159576236733497
0.281598681885032
0.28160160704858567
0.2816045378396554
0.2816074742399485
0.2816104162312201
0.2816133637952728
0.28161631691395683
0.2816192755691696
0.2816222397428558
0.281625209417007
Accuracy: 0.656084656084656

Confusion Matrix:  [[160  28]
 [102  88]]
LR model precision: 76%
LR model recall: 46%
```

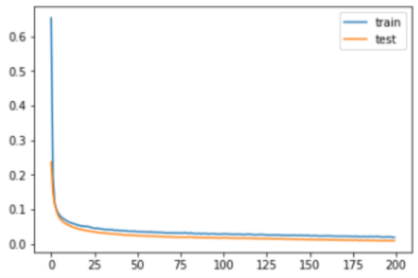


Neural Network (Autoencoder):

Lastly, we have trained an autoencoder network using all 25 independent features. The train-test loss graph at the training step was able to verify the working process of the network.

```
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
# define an encoder model (without the decoder)
encoder = Model(inputs=visible, outputs=bottleneck)
plot_model(encoder, 'encoder_no_compress.png', show_shapes=True)
# save the encoder to file
encoder.save('encoder.h5')
```

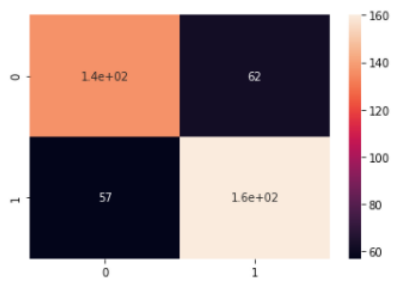
```
Epoch 199/200
 - 0s - loss: 0.0199 - val_loss: 0.0099
Epoch 200/200
 - 0s - loss: 0.0197 - val_loss: 0.0098
```



Finally, we used the trained auto encoder model to be fitted into Logistic Regression model which provided us with a much better accuracy and recall. Since the true positive and the true negative was higher compared to others, we can say that autoencoder was able to predict 'Yes' and 'No' classes more accurately than other models.

```
model.fit(X_train_encode, y_train)
# make predictions on the test set
yhat = model.predict(X_test_encode)
# calculate classification accuracy
acc = accuracy_score(y_test, yhat)
print("AutoEncoder Accuracy: ", round ((acc*100),2))

cf_matrix = confusion_matrix(y_test, yhat)
print("\nConfusion Matrix: ",cf_matrix)

sns.heatmap(cf_matrix, annot=True)

precision = '{:.0%}'.format(precision_score(y_test, yhat))

print(f'LR model precision: {precision}')

recall = '{:.0%}'.format(recall_score(y_test, yhat))

print(f'LR model recall: {recall}')
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not* compiled. Compile it manually.
AutoEncoder Accuracy:  71.39

Confusion Matrix:  [[137  62]
 [ 57 160]]
LR model precision: 72%
LR model recall: 74%
```


1.4e+02
62
57
1.6e+02
0
1
0
1
160
140
120
100
80
60