project_part_4_Clustering

March 10, 2022

1 NOTEBOOK 4: CLUSTERING

1.0.1 Team 3

- Anjali Sebastian
- Yesha Sharma
- Rupansh Phutela

1.0.2 What this Notebook does?

After Data selection, cleaning, pre-processing, EDA and Regression Analysis, we will now look at how we can cluster our data. Although our data has already target varibale y = Diabetes (Yes or No) we will try to cluster the data to see if any new patterns emerge. - Normalization of entire dataset due to varying ranges of different attributes - Simple Visualization of the Dataset - Cluster using Agglomerative Clustering - Cluster using K-Means Clustering - Cluster using Mini Batch K-Means Clustering - Cluster using MeanShift - Comparing clustering algorithmn with two clusters vis a vis real target labels. Here we assume C1 as label=0 (No Diabetes) and C2 as label = 1 (yes Diabetes) for doing our comparision . This can be flipped since the cluster created can have any meaning as such. (however in most cases we can reasonably assume one of the clusters to correspond to one of the target labels. - Analysis of all the clustering algorithms and conclusion

1.1 1. Import Packages and Setup

[1]:
```
# you need  Python 3.5
import sys
assert sys.version_info >= (3, 5)
```

[2]:
```
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
```

[3]:
```
import os
import pandas as pd
import numpy as np
import seaborn as sns
import time
import warnings
warnings.filterwarnings("ignore")
##############################################
```

[4]:
```
# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)


# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "clustering_kmeans"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

# method to save figures
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.2 2. Utility Functions

[5]:
```
import matplotlib.patches as mpatches
from matplotlib.colors import ListedColormap, BoundaryNorm

def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_labelled_scatter(X, y, class_labels):
    num_labels = len(class_labels)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    marker_array = ['o', '^', '*']
    color_array = ['#FFFF00','#00AAFF','#000000','#FF00AA','#2ca02c',␣
 ↪'#d62728','#9467bd','#8c564b','#e377c2']
    cmap_bold = ListedColormap(color_array)
    bnorm = BoundaryNorm(np.arange(0, num_labels + 1, 1), ncolors=num_labels)
    plt.figure()
```

```
    plt.scatter(X[:, 0], X[:, 1], s=65, c=y, cmap=cmap_bold, norm = bnorm,␣
↪alpha = 0.40, edgecolor='black', lw = 1)

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    h = []
    for c in range(0, num_labels):
        h.append(mpatches.Patch(color=color_array[c], label=class_labels[c]))
    plt.legend(handles=h)
    plt.show()

def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram

    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]
    ).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

1.3 3. Read Data and Display

[6]:
```
diabetes = pd.read_csv('./diabetes.csv')
```

[7]:
```
diabetes.head()
```

[7]:

	Unnamed: 0	Diabetes	BMI	State	HighBP	HighChol	CholCheck	\
0	0	0.0	28.17	AL	1.0	1.0	1.0	
1	1	0.0	18.54	AL	0.0	0.0	1.0	
2	2	1.0	31.62	AL	1.0	0.0	1.0	
3	6	1.0	32.98	AL	0.0	0.0	1.0	
4	9	1.0	16.65	AL	0.0	1.0	1.0	

```
   FruitConsume  VegetableConsume  Smoker  …  NoDoctorDueToCost  \
0           1.0               1.0     1.0  …                0.0
1           1.0               1.0     0.0  …                0.0
2           1.0               1.0     0.0  …                0.0
3           1.0               1.0     1.0  …                0.0
4           0.0               0.0     1.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0            3.0            15.0           0.0
1               1.0            2.0            10.0           0.0
2               1.0            3.0             0.0          30.0
3               1.0            4.0            30.0           0.0
4               0.0            1.0            20.0           0.0

   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 24 columns]
```

[8]:
```
#set datatypes of columns to boolean or categorical as appropriate
make_bool_int = ['Diabetes','HighBP','HighChol','CholCheck',\
                 ␣
 ↪'FruitConsume','VegetableConsume','Smoker','HeavyDrinker','Stroke','HeartDisease',\
                 ␣
 ↪'Healthcare','NoDoctorDueToCost','PhysicalActivity','DifficultyWalking','Gender']
make_categorical_int = ['GeneralHealth','Age','Education','Income']
```

[9]:
```
#drop the extra index column in datafram
diabetes=diabetes.drop(['Unnamed: 0'], axis=1)

#drop the state column in dataframe since it will not be used in the dataframe
diabetes=diabetes.drop(['State'], axis=1)
```

[10]:
```
diabetes.head()
```

[10]:
```
   Diabetes    BMI  HighBP  HighChol  CholCheck  FruitConsume  \
0       0.0  28.17     1.0       1.0        1.0           1.0
1       0.0  18.54     0.0       0.0        1.0           1.0
2       1.0  31.62     1.0       0.0        1.0           1.0
3       1.0  32.98     0.0       0.0        1.0           1.0
4       1.0  16.65     0.0       1.0        1.0           0.0
```

```
   VegetableConsume  Smoker  HeavyDrinker  Stroke  …  NoDoctorDueToCost  \
0               1.0     1.0           0.0     0.0  …                0.0
1               1.0     0.0           0.0     0.0  …                0.0
2               1.0     0.0           0.0     0.0  …                0.0
3               1.0     1.0           0.0     0.0  …                0.0
4               0.0     1.0           0.0     0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0            3.0            15.0           0.0
1               1.0            2.0            10.0           0.0
2               1.0            3.0             0.0          30.0
3               1.0            4.0            30.0           0.0
4               0.0            1.0            20.0           0.0

   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 22 columns]
```

[11]:
```
# deep copy before next stage
df = diabetes.copy(deep = True)
```

[12]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 243317 entries, 0 to 243316
Data columns (total 22 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   Diabetes            243317 non-null  float64
 1   BMI                 243317 non-null  float64
 2   HighBP              243317 non-null  float64
 3   HighChol            243317 non-null  float64
 4   CholCheck           243317 non-null  float64
 5   FruitConsume        243317 non-null  float64
 6   VegetableConsume    243317 non-null  float64
 7   Smoker              243317 non-null  float64
 8   HeavyDrinker        243317 non-null  float64
 9   Stroke              243317 non-null  float64
 10  HeartDisease        243317 non-null  float64
 11  Healthcare          243317 non-null  float64
 12  NoDoctorDueToCost   243317 non-null  float64
 13  PhysicalActivity    243317 non-null  float64
 14  GeneralHealth       243317 non-null  float64
```

```
 15  PhysicalHealth      243317 non-null  float64
 16  MentalHealth        243317 non-null  float64
 17  DifficultyWalking   243317 non-null  float64
 18  Gender              243317 non-null  float64
 19  Age                 243317 non-null  float64
 20  Education           243317 non-null  float64
 21  Income              243317 non-null  float64
dtypes: float64(22)
memory usage: 40.8 MB
```

[13]:
```
df.shape
```

[13]:
```
(243317, 22)
```

1.4 4. Normalization and Simple Vizualization

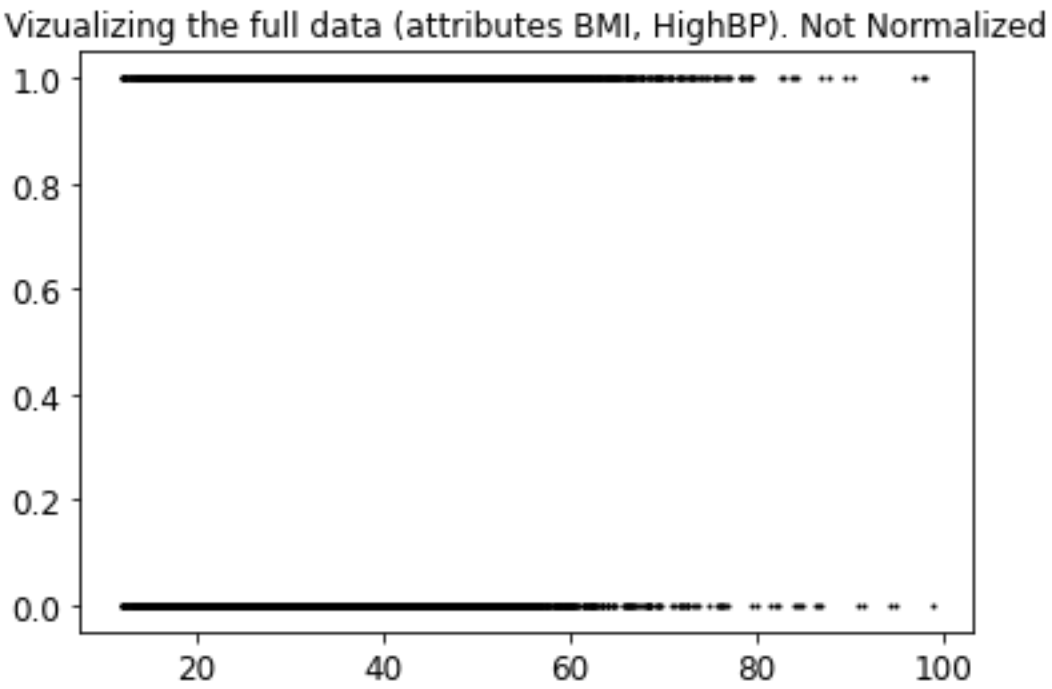
[14]:
```
X_columns = ['BMI', 'HighBP', 'HighChol', 'CholCheck', 'FruitConsume',
       'VegetableConsume', 'Smoker', 'HeavyDrinker', 'Stroke', 'HeartDisease',
       'Healthcare', 'NoDoctorDueToCost', 'PhysicalActivity', 'GeneralHealth',
       'PhysicalHealth', 'MentalHealth', 'DifficultyWalking', 'Gender', 'Age',
       'Education', 'Income']
```

Note: The entire data set is 0.24 million enteries. The agglomerative clustering and Mean Shift clustering algorithms we causing the kernal to crash because the sized of data set is too large. So we are going to take a random sample of 10,000 entries to perform clustering and see how all the clustering algorithmns perform.

[15]:
```
# Selecting a random sample for the data set

#sampling a random number of values since plotting all 0.2 million datapoints␣
 ↪will crash the kernal
number_of_samples = 10000
df_sample = df.sample(number_of_samples, random_state=42)
```

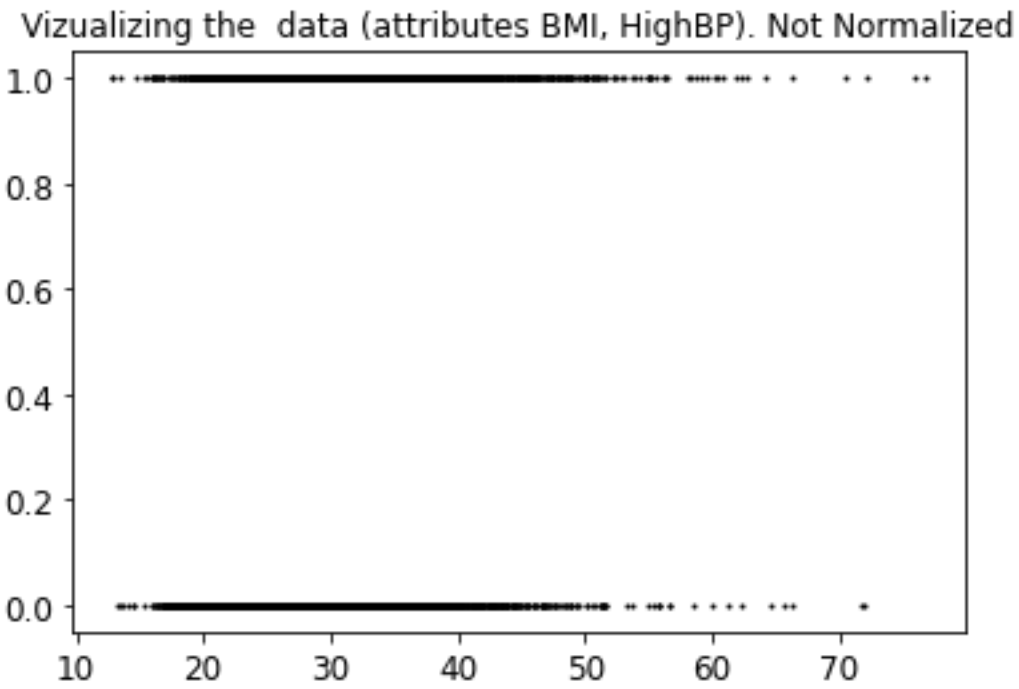
[16]:
```
# separating the target column y = Diabetes before clustering
# for complete dataset
X_df = df[X_columns].values
y_df = df[['Diabetes']]
plot_data(X_df)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Not Normalized")
```

[16]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Not
Normalized')
```



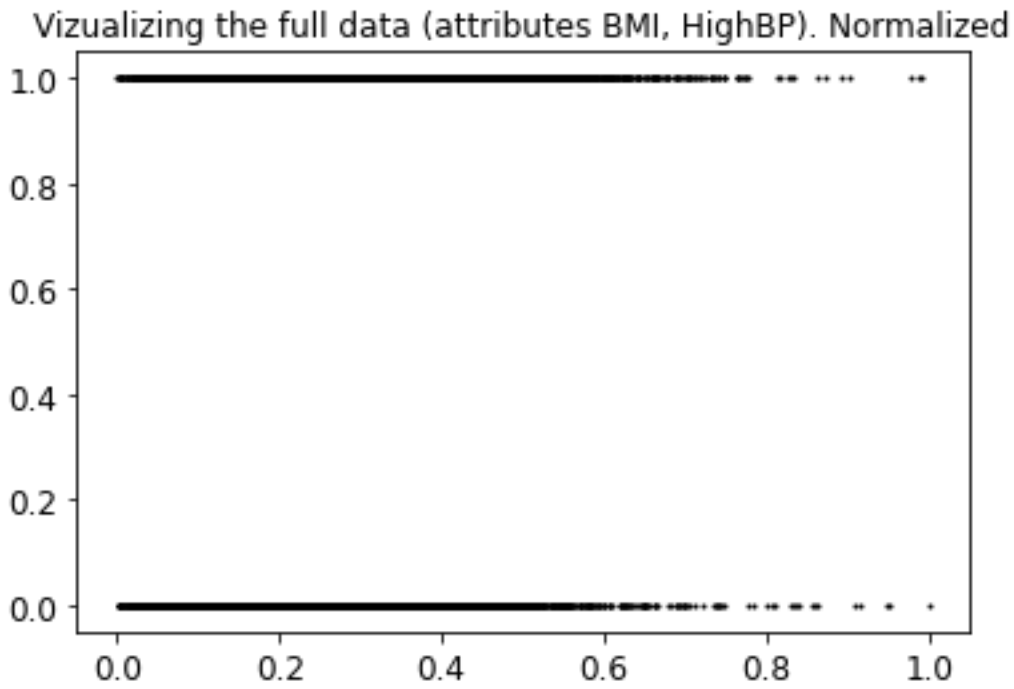
[17]:
```
# separating the target column y = Diabetes before clustering

# for sampled dataset
X_sample_df = df_sample[X_columns].values
y_sample_df = df_sample[['Diabetes']]
plot_data(X_sample_df)
plt.title("Vizualizing the  data (attributes BMI, HighBP). Not Normalized")
```

[17]:
```
Text(0.5, 1.0, 'Vizualizing the  data (attributes BMI, HighBP). Not Normalized')
```



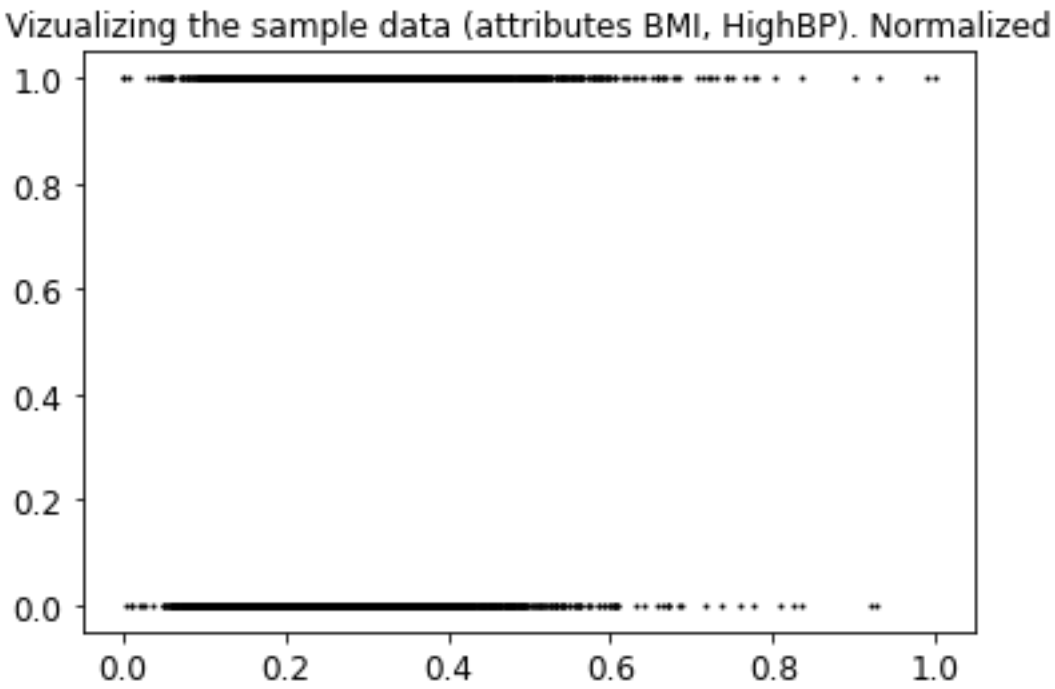
[18]:
```
# Using minmax scaler for normalization
from sklearn.preprocessing import MinMaxScaler

# normalization full dataset
X_normalized = MinMaxScaler().fit(X_df).transform(X_df)
df_normalized = pd.DataFrame(X_normalized, columns=X_columns )
plot_data(X_normalized)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Normalized")
```

[18]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Normalized')
```



[19]:
```
# normalization sample dataset
X_sample_normalized = MinMaxScaler().fit(X_sample_df).transform(X_sample_df)
df_sample_normalized = pd.DataFrame(X_sample_normalized, columns=X_columns )
plot_data(X_sample_normalized)
plt.title("Vizualizing the sample data (attributes BMI, HighBP). Normalized")
```

[19]:
```
Text(0.5, 1.0, 'Vizualizing the sample data (attributes BMI, HighBP).
Normalized')
```



[20]:
```
# Normalized features in pandas format
df_normalized.head()
```

[20]:

	BMI	HighBP	HighChol	CholCheck	FruitConsume	VegetableConsume	\
0	0.186505	1.0	1.0	1.0	1.0	1.0	
1	0.075433	0.0	0.0	1.0	1.0	1.0	
2	0.226298	1.0	0.0	1.0	1.0	1.0	
3	0.241984	0.0	0.0	1.0	1.0	1.0	
4	0.053633	0.0	1.0	1.0	0.0	0.0	

	Smoker	HeavyDrinker	Stroke	HeartDisease	…	NoDoctorDueToCost	\
0	1.0	0.0	0.0	0.0	…	0.0	
1	0.0	0.0	0.0	0.0	…	0.0	
2	0.0	0.0	0.0	0.0	…	0.0	
3	1.0	0.0	0.0	0.0	…	0.0	
4	1.0	0.0	0.0	0.0	…	0.0	

	PhysicalActivity	GeneralHealth	PhysicalHealth	MentalHealth	\
0	0.0	0.50	0.500000	0.0	
1	1.0	0.25	0.333333	0.0	
2	1.0	0.50	0.000000	1.0	
3	1.0	0.75	1.000000	0.0	
4	0.0	0.00	0.666667	0.0	

```
   DifficultyWalking  Gender       Age  Education    Income
0                1.0     0.0  1.000000        0.4  0.285714
1                0.0     0.0  0.833333        0.8  0.571429
2                1.0     0.0  0.750000        1.0  0.857143
3                1.0     1.0  0.833333        1.0  0.857143
4                1.0     0.0  0.833333        0.2  0.285714

[5 rows x 21 columns]
```

[21]:
```
# Normalized features in numpy format
X_normalized
```

[21]:
```
array([[0.18650519, 1.        , 1.        , …, 1.        , 0.4       ,
        0.28571429],
       [0.07543253, 0.        , 0.        , …, 0.83333333, 0.8       ,
        0.57142857],
       [0.22629758, 1.        , 0.        , …, 0.75      , 1.        ,
        0.85714286],
       …,
       [0.1905421 , 0.        , 0.        , …, 0.5       , 0.4       ,
        0.        ],
       [0.227797  , 0.        , 1.        , …, 0.        , 0.6       ,
        0.28571429],
       [0.09192618, 0.        , 0.        , …, 0.33333333, 1.        ,
        1.        ]])
```

Note: The data pairs are as follows: - Full Data 1. X_df (pandas) with y_df(pandas) : not normalized full data set 2. X_normalized (numpy) with y_df(pandas) : normalized full X in numpy (easy for clustering) 3. df_normalized (pandas) with y_df(pandas) : normalized X in pandas format (easy for tracking feature names) - Sample Data of 10,000 randomly selected rows 1. X_sample_df (pandas) with y_sample_df(pandas) : not normalized sample data set 2. X_df_normalized (numpy) with y_sample_df(pandas) : normalized sample X in numpy (easy for clustering) 3. df_sample_normalized (pandas) with y_sample_df(pandas) : normalized X sample in pandas format (easy for tracking feature names)

- For all our clustering we will use only the normalized versions of the dataset.

1.5 5. AGGLOMERATIVE CLUSTERING

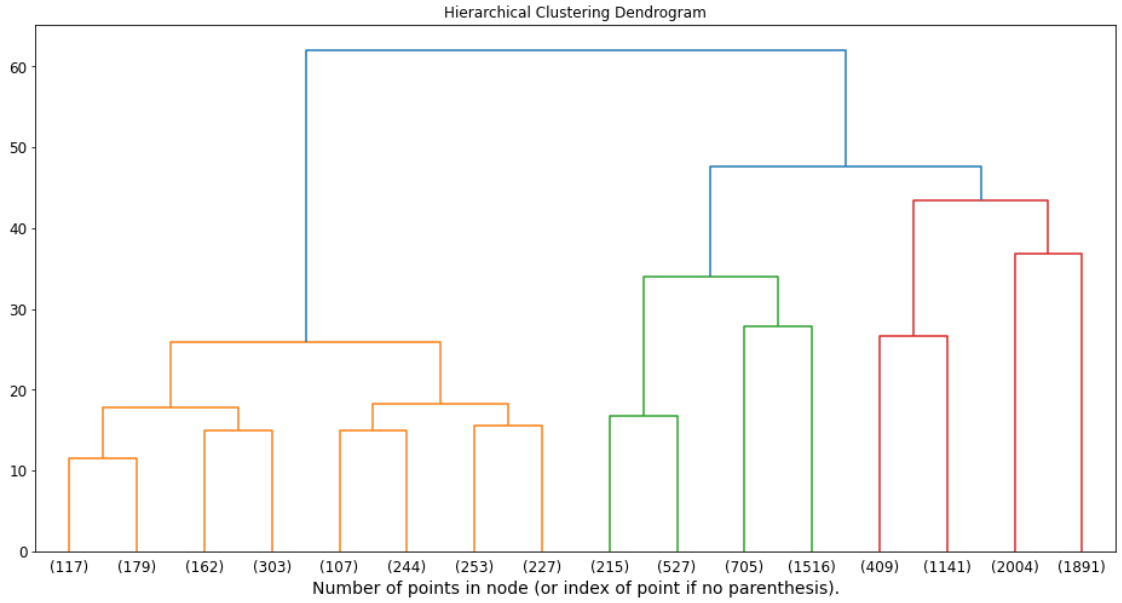
- For agglomerative Clustering we will only use the samples dataset (10,000) values as the dendrogram will not be visualized properly if all 0.24 million data points are used.
- We are using the normalized sample of 10,000 records

Apply Agglomerative Clustering

[22]:
```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram
aggo = AgglomerativeClustering(distance_threshold=0,n_clusters=None)
aggo = aggo.fit(df_sample_normalized)
```

Dendrogram Vizualization

[23]:
```
fig , ax = plt.subplots(1,1,figsize=(16, 8))
plt.title("Hierarchical Clustering Dendrogram")
# plot the top three levels of the dendrogram
plot_dendrogram(aggo, truncate_mode="level", p=3)
plt.xlabel("Number of points in node (or index of point if no parenthesis).")
plt.show()
```



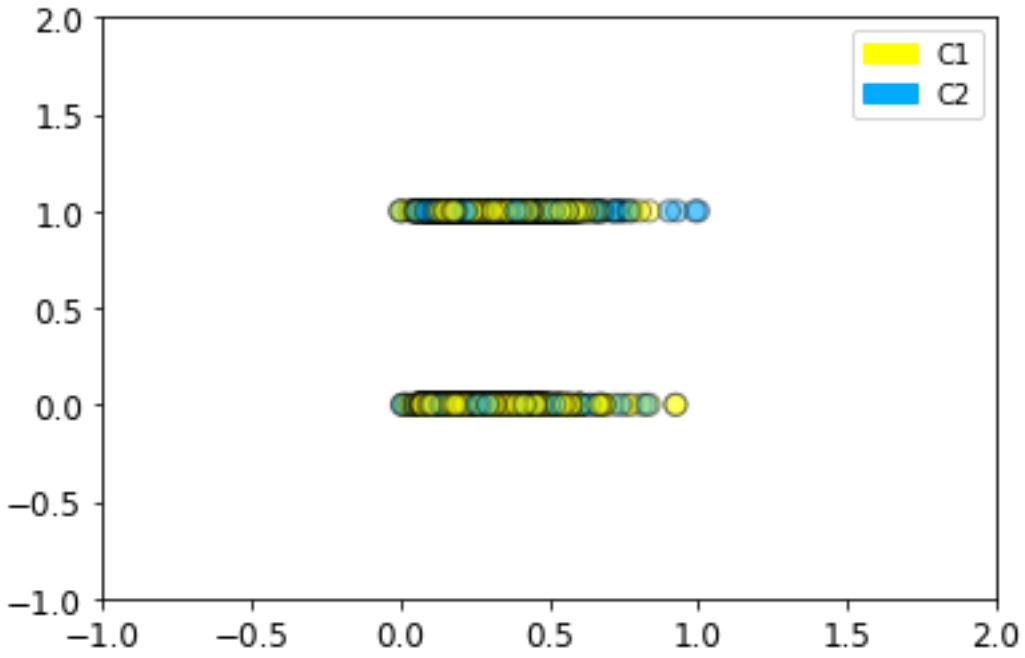
Notes: Using the dendogram we can see that we can go with either 3 clusters or 2 clusters.

Compare Clustering Generated Labels with Known Labels

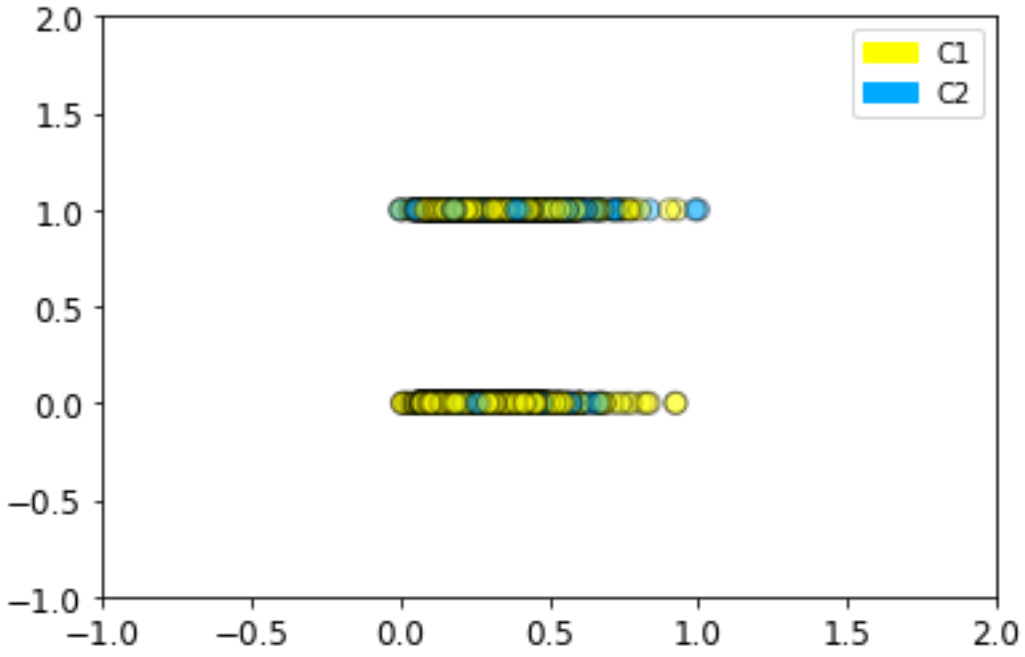
[24]:
```
# Use agglomerative Clustering to make 2 clusters
aggo_2c = AgglomerativeClustering(n_clusters=2)
aggo_2c = aggo_2c.fit(df_sample_normalized)
aggo_2c.labels_
```

[24]:
```
array([0, 0, 0, …, 0, 0, 1])
```

[25]:
```
# Labels Generated By Agglomerative Clustering
plot_labelled_scatter(X_sample_normalized, aggo_2c.labels_, ['C1', 'C2'])
```



[26]:
```
# Known Labels from target y = Diabetes
plot_labelled_scatter(X_sample_normalized, y_sample_df.to_numpy(), ['C1', 'C2'])
```



[27]:
```
# find similarity between the labels
df_temp = pd.DataFrame(aggo_2c.labels_, columns=["Cluster_Labels"] )
df_similarity = pd.concat([y_sample_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 79.67999999999999
```

[28]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
percentage_positive
print("Total Percentage of positive over actuals", percentage_positive)
```

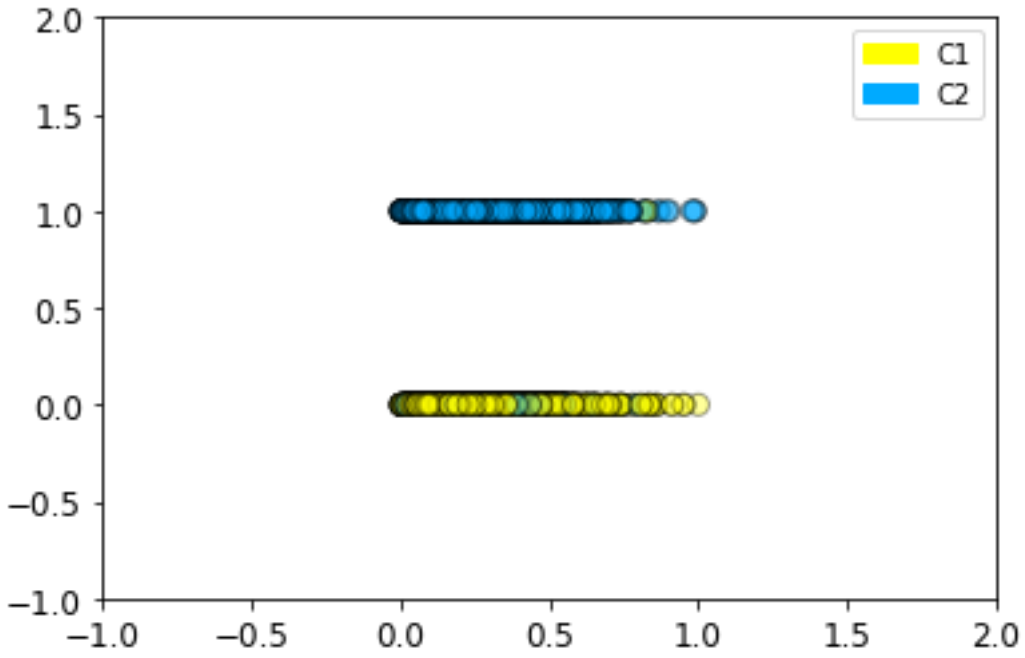
```
Total Percentage of positive over actuals 0.3527443105756359
```

1.6 6. KMEANS CLUSTERING

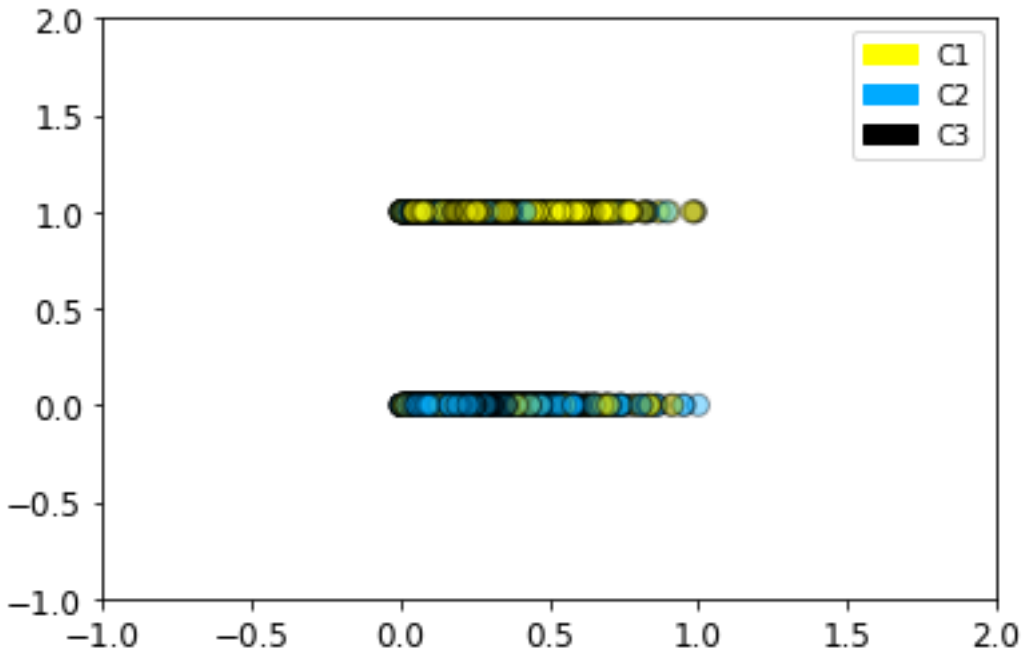
- KMeans is a very computationatility efficent algo. It is able to cluster the entire dataset in reasonable time

Simple KMeans Models -With 2,3,and 4 clusters

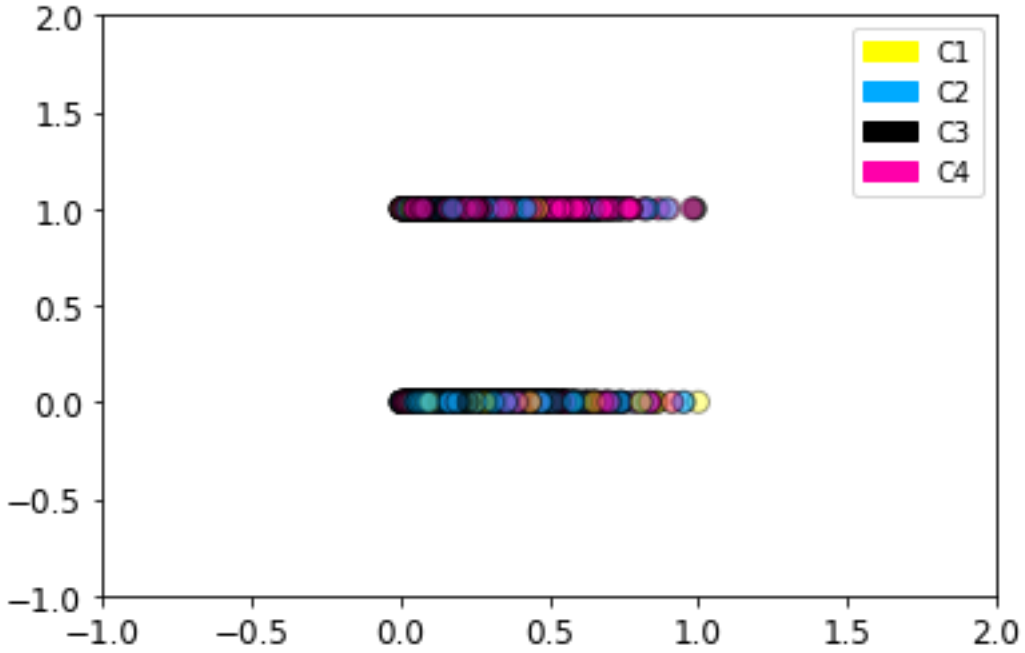
[29]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, random_state=0)
kmeans.fit(X_normalized)
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])
```



[30]:
```
kmeans = KMeans(n_clusters=3, random_state=0)
kmeans.fit(X_normalized)
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3'])
```

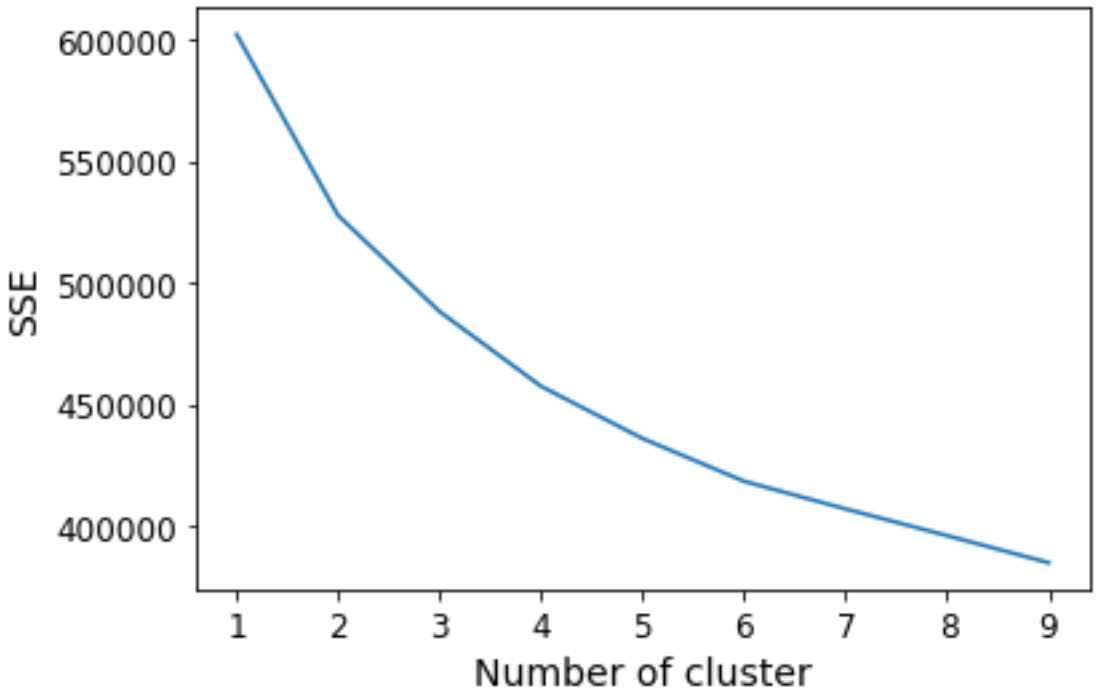


[31]:
```
kmeans = KMeans(n_clusters=4, random_state=0)
kmeans.fit(X_normalized)
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3','C4'])
```



Inertia Measures

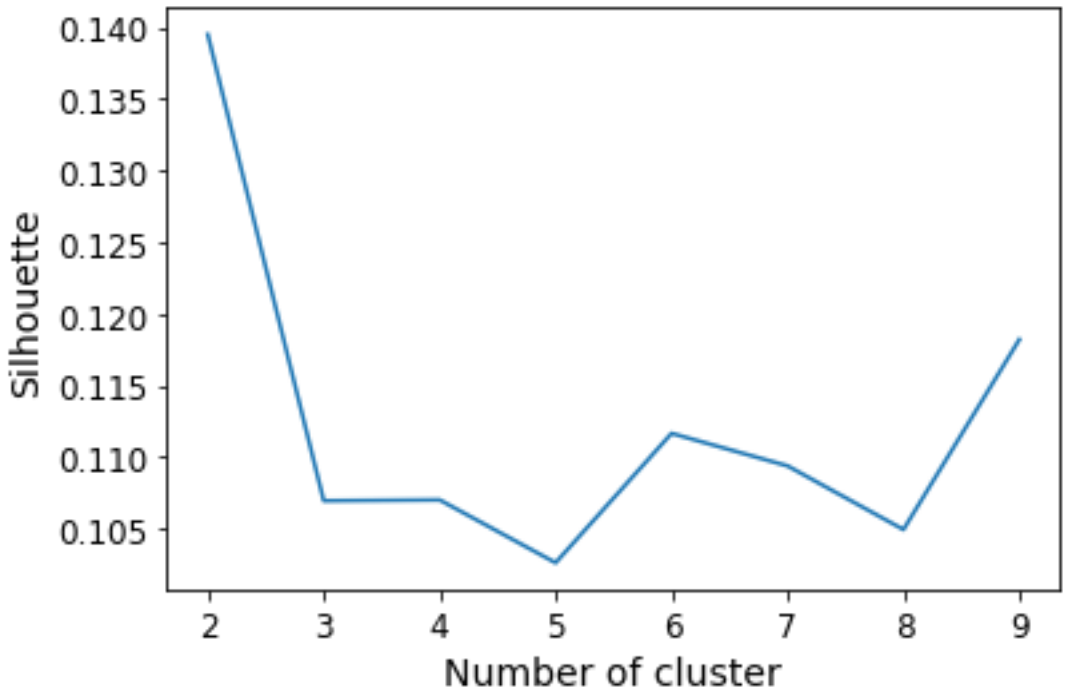
[32]:
```
sse = {}
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(X_normalized)
    df["clusters"] = kmeans.labels_

    # Inertia: Sum of distances of samples to their closest cluster center
    sse[k] = kmeans.inertia_

plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.ylabel("SSE")
plt.show()
```



Silhouette Score

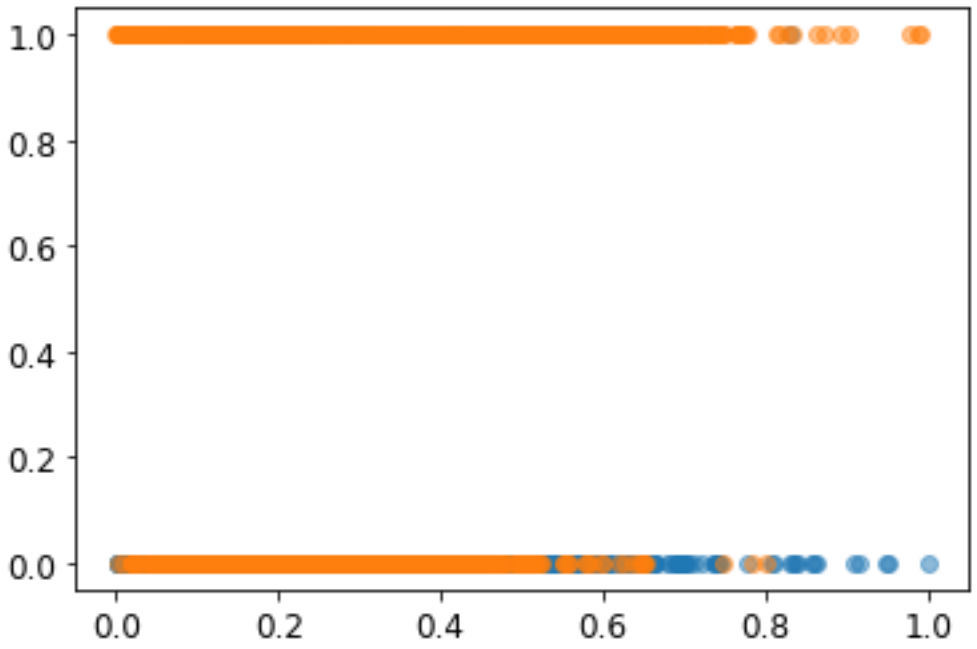
- Generating a silloutte score is computationally expensive so we are doing it for only the sample dataset

[33]:
```
# using only 10000 samples
from sklearn import metrics
score = {}
for k in range(2, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_sample_normalized)
    labels = kmeans.labels_

    score[k] = metrics.silhouette_score(df_sample_normalized, labels,␣
 ↪metric='euclidean')

plt.figure()
plt.plot(list(score.keys()), list(score.values()))
plt.xlabel("Number of cluster")
plt.ylabel("Silhouette")
plt.show()
```

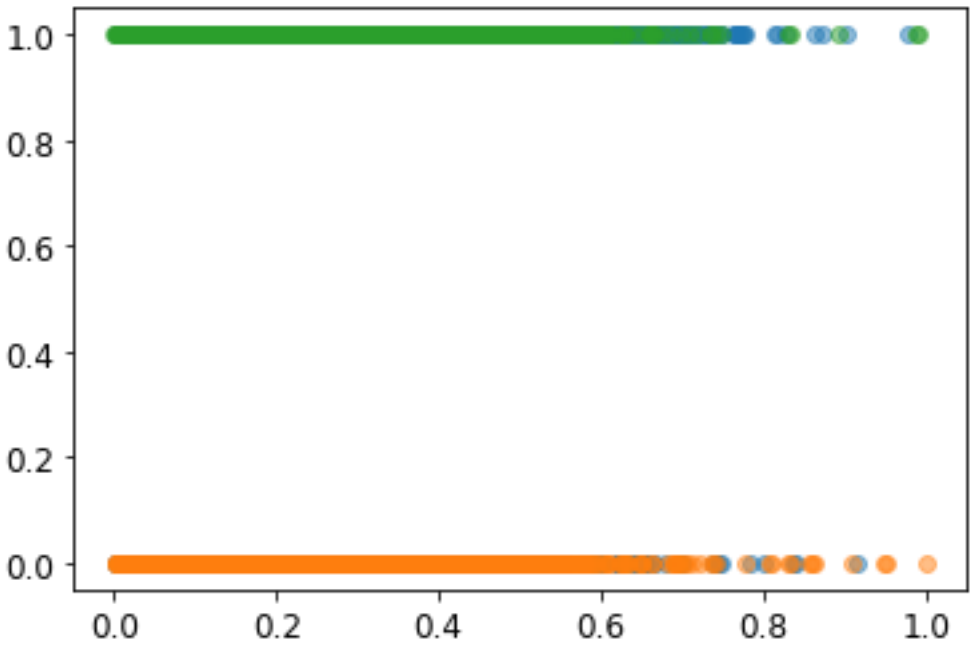


Vizualizing Kmeans for different cluster numbers

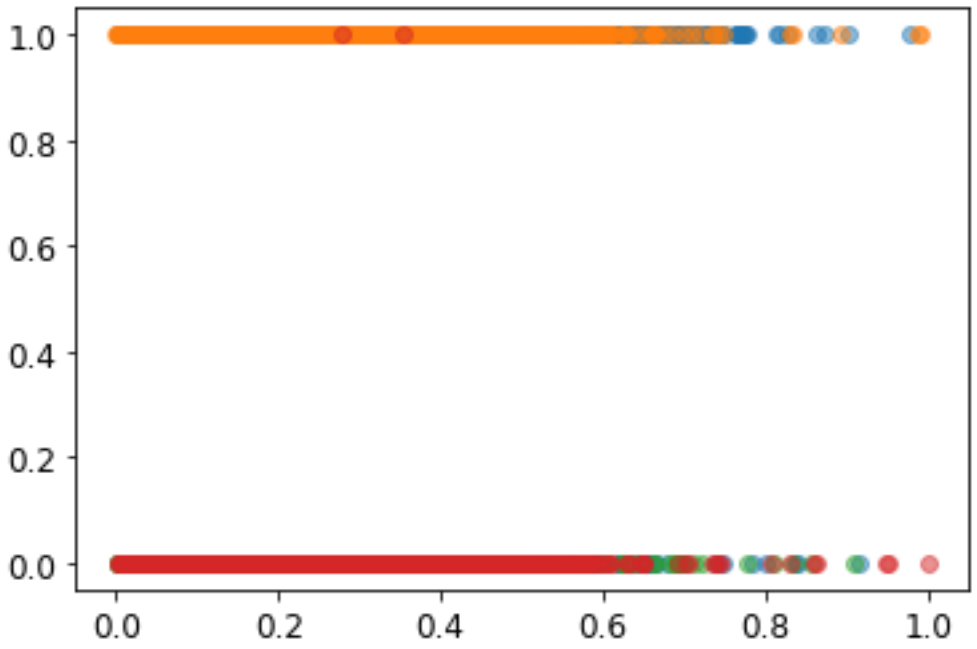
[34]:
```
model_2c = KMeans(n_clusters=2)
model_2c.fit(X_normalized)
yhat = model_2c.predict(X_normalized)
clusters = np.unique(yhat)
for cluster in clusters:
    row_ix = np.where (yhat == cluster)
    plt.scatter(X_normalized[row_ix,0],X_normalized[row_ix,1],alpha = 0.5)
```



[35]:
```
model_3c = KMeans(n_clusters=3)
model_3c.fit(X_normalized)
yhat = model_3c.predict(X_normalized)
clusters = np.unique(yhat)
for cluster in clusters:
    row_ix = np.where (yhat == cluster)
    plt.scatter(X_normalized[row_ix,0],X_normalized[row_ix,1],alpha = 0.5)
```



[36]:
```
model_4c = KMeans(n_clusters=4)
model_4c.fit(X_normalized)
yhat = model_4c.predict(X_normalized)
clusters = np.unique(yhat)
for cluster in clusters:
    row_ix = np.where (yhat == cluster)
    plt.scatter(X_normalized[row_ix,0],X_normalized[row_ix,1],alpha = 0.5)
```



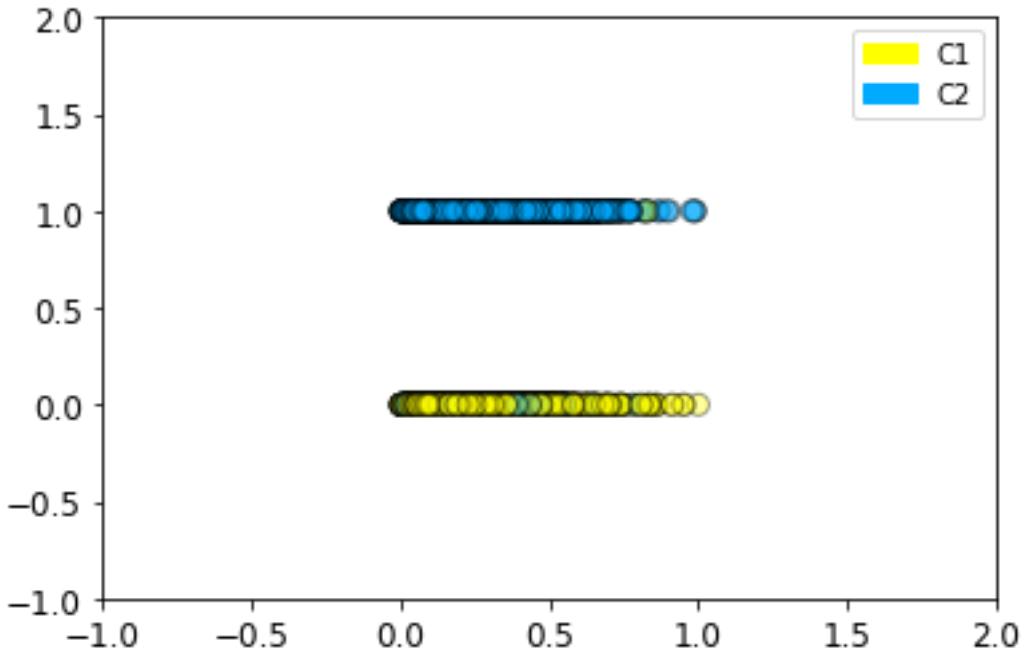
Note: Choice of number of clusters. Looking at agglomerative clustering we should choose either 2 or 3 cluster. Looking at the inertia score 2,3 or 4 . Using silouttte score 2 , 5 or 7 would be a good option. Since we know that the target label has 2 class "Has diabetes" and "Does not have diabetes" n_clusters = 2 would be a good choice

Compare Clustering Generated Labels with Known Labels

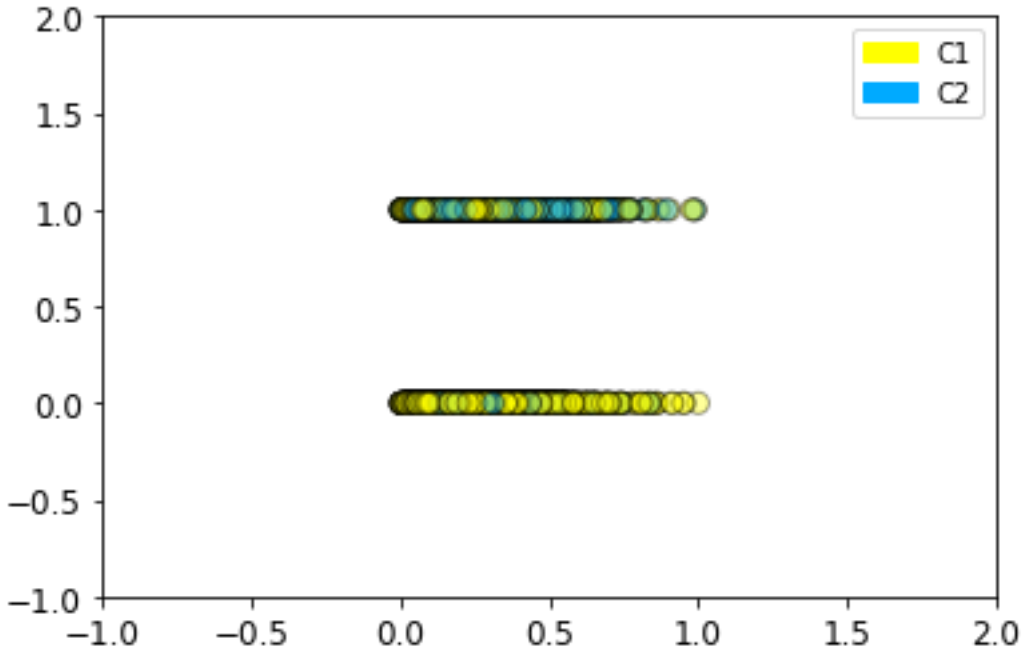
[37]:
```
# Kmeans with 2 clusters
kmeans = KMeans(n_clusters=2, random_state=0)
kmeans.fit(X_normalized)
```

[37]:
```
KMeans(n_clusters=2, random_state=0)
```

[38]:
```
# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])
```



[39]:
```
# using actual target labels
plot_labelled_scatter(X_normalized, y_df.to_numpy(), ['C1', 'C2'])
```



[40]:
```
# find similarity between the labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
df_similarity = pd.concat([y_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
df_similarity.head()
```

[40]:

	Diabetes	Cluster_Labels
0	0.0	1
1	0.0	0
2	1.0	1
3	1.0	0
4	1.0	1

[41]:
```
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 63.39589917679406
```

[42]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.7845548032522168
```

Note: 63.39% and percentage positive is across the entire dataset

[43]:
```
# Kmeans with 2 clusters or sample
kmeans_s = KMeans(n_clusters=2, random_state=0)
kmeans_s.fit(X_sample_normalized)

# find similarity between the labels
df_temp = pd.DataFrame(kmeans_s.labels_, columns=["Cluster_Labels"] )
df_similarity = pd.concat([y_sample_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)

same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 65.91
```

[44]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.7831325301204819
```

Note: 65.91% is for the sample of 10,000 datapoints

1.6.1 7. KMeans Mini Batch

- We are doing MiniBatch for 2 clusters with different batch sizes (8,32,128)
- First we will find ideal cluster numbers for different batch sizes using intertia and silloutte score

[45]:
```
from sklearn.cluster import MiniBatchKMeans
```

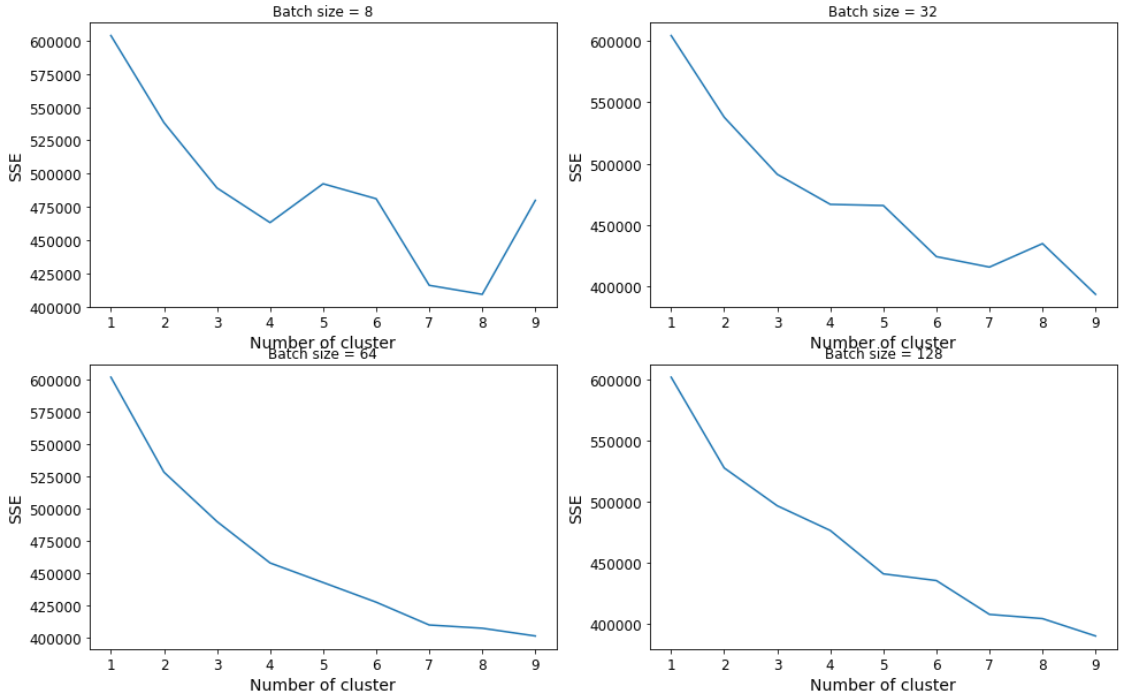
Inertia Measure

- Batch Size = 8, 32, 64, 128

[46]:
```
sse = {}
for k in range(1, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=8,␣
↪max_iter=100).fit(X_normalized)
    df["clusters"] = kmeans.labels_

    # Inertia: Sum of distances of samples to their closest cluster center
    sse[k] = kmeans.inertia_

fig, ([ax1, ax2], [ax3,ax4]) = plt.subplots(2, 2)
fig.set_size_inches(16, 10)
ax1.plot(list(sse.keys()), list(sse.values()))
ax1.set_xlabel("Number of cluster")
ax1.set_ylabel("SSE")
ax1.set_title("Batch size = 8")

sse = {}
for k in range(1, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=32,␣
↪max_iter=100).fit(X_normalized)
    df["clusters"] = kmeans.labels_

    # Inertia: Sum of distances of samples to their closest cluster center
    sse[k] = kmeans.inertia_


ax2.plot(list(sse.keys()), list(sse.values()))
```

```
ax2.set_xlabel("Number of cluster")
ax2.set_ylabel("SSE")
ax2.set_title("Batch size = 32")


sse = {}
for k in range(1, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=64,␣
 ↪max_iter=100).fit(X_normalized)
    df["clusters"] = kmeans.labels_

    # Inertia: Sum of distances of samples to their closest cluster center
    sse[k] = kmeans.inertia_


ax3.plot(list(sse.keys()), list(sse.values()))
ax3.set_xlabel("Number of cluster")
ax3.set_ylabel("SSE")
ax3.set_title("Batch size = 64")


sse = {}
for k in range(1, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=128,␣
 ↪max_iter=100).fit(X_normalized)
    df["clusters"] = kmeans.labels_

    # Inertia: Sum of distances of samples to their closest cluster center
    sse[k] = kmeans.inertia_


ax4.plot(list(sse.keys()), list(sse.values()))
ax4.set_xlabel("Number of cluster")
ax4.set_ylabel("SSE")
ax4.set_title("Batch size = 128")
```

[46]: `Text(0.5, 1.0, 'Batch size = 128')`

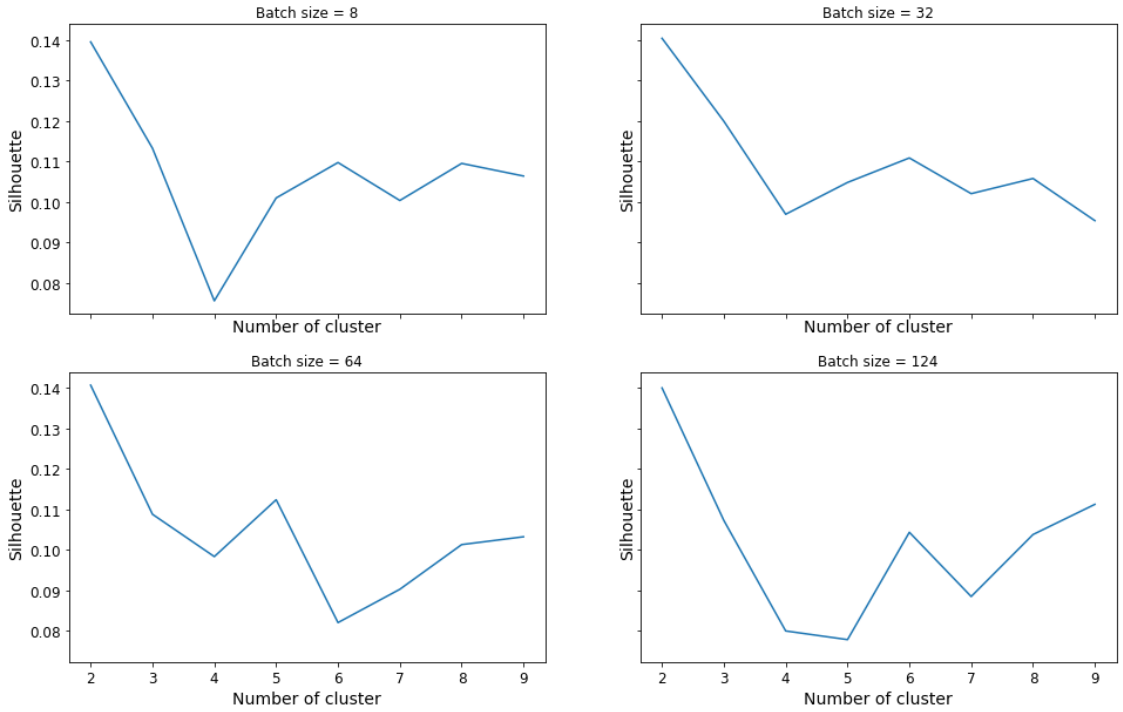


Note Looking at the inertia measures for different batch sizes we can see that there are elbows at 2, 4 and 8 clusters which is prominent. As we increase the batch size the intertia curve looks very similar to the one we plotted for the Kmeans algorithm.

[47]:
```
# using only 10,000 samples
number_of_samples = 10000
df_sample = df_normalized.sample(number_of_samples, random_state=42)
X_sample = df_sample.to_numpy()

fig, ([ax1, ax2], [ax3,ax4]) = plt.subplots(2,2,sharex=True, sharey=True)
fig.set_size_inches(16, 10)


score = {}
for k in range(2, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=8,␣
 ↪max_iter=100).fit(df_sample)
    labels = kmeans.labels_

    score[k] = metrics.silhouette_score(df_sample, labels, metric='euclidean')

ax1.plot(list(score.keys()), list(score.values()))
ax1.set_xlabel("Number of cluster")
ax1.set_ylabel("Silhouette")
ax1.set_title("Batch size = 8")
```

```
score = {}
for k in range(2, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=32,␣
 ↪max_iter=100).fit(df_sample)
    labels = kmeans.labels_

    score[k] = metrics.silhouette_score(df_sample, labels, metric='euclidean')

ax2.plot(list(score.keys()), list(score.values()))
ax2.set_xlabel("Number of cluster")
ax2.set_ylabel("Silhouette")
ax2.set_title("Batch size = 32")

score = {}
for k in range(2, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=64,␣
 ↪max_iter=100).fit(df_sample)
    labels = kmeans.labels_

    score[k] = metrics.silhouette_score(df_sample, labels, metric='euclidean')

ax3.plot(list(score.keys()), list(score.values()))
ax3.set_xlabel("Number of cluster")
ax3.set_ylabel("Silhouette")
ax3.set_title("Batch size = 64")

score = {}
for k in range(2, 10):
    kmeans = MiniBatchKMeans(n_clusters=k,random_state=0,batch_size=128,␣
 ↪max_iter=100).fit(df_sample)
    labels = kmeans.labels_

    score[k] = metrics.silhouette_score(df_sample, labels, metric='euclidean')

ax4.plot(list(score.keys()), list(score.values()))
ax4.set_xlabel("Number of cluster")
ax4.set_ylabel("Silhouette")
ax4.set_title("Batch size = 124")

plt.xlabel("Number of cluster")
plt.ylabel("Silhouette")
```

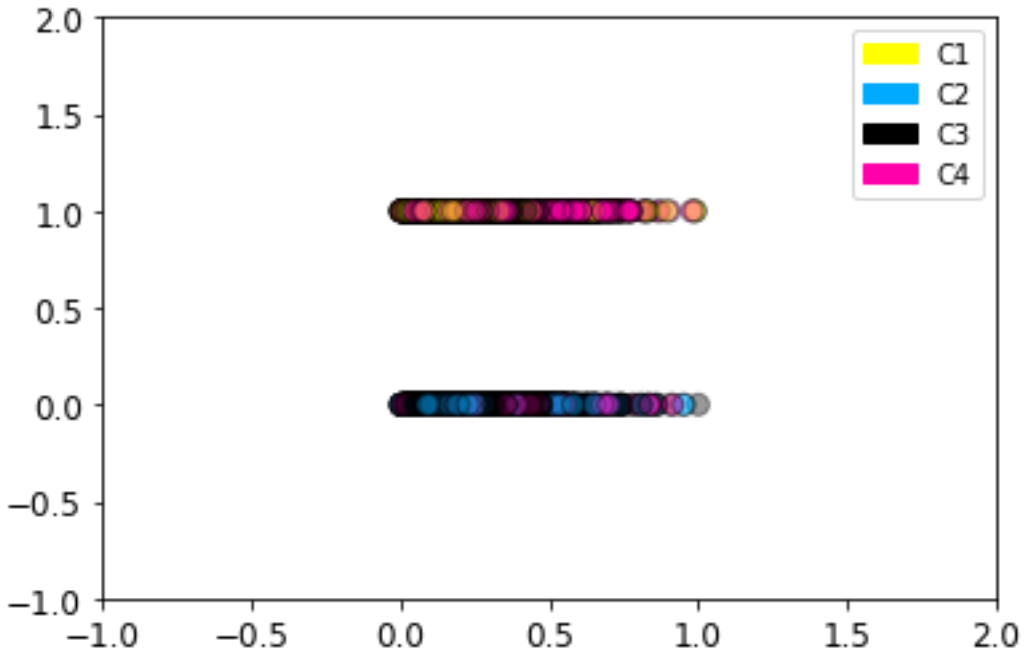
[47]: Text(0, 0.5, 'Silhouette')



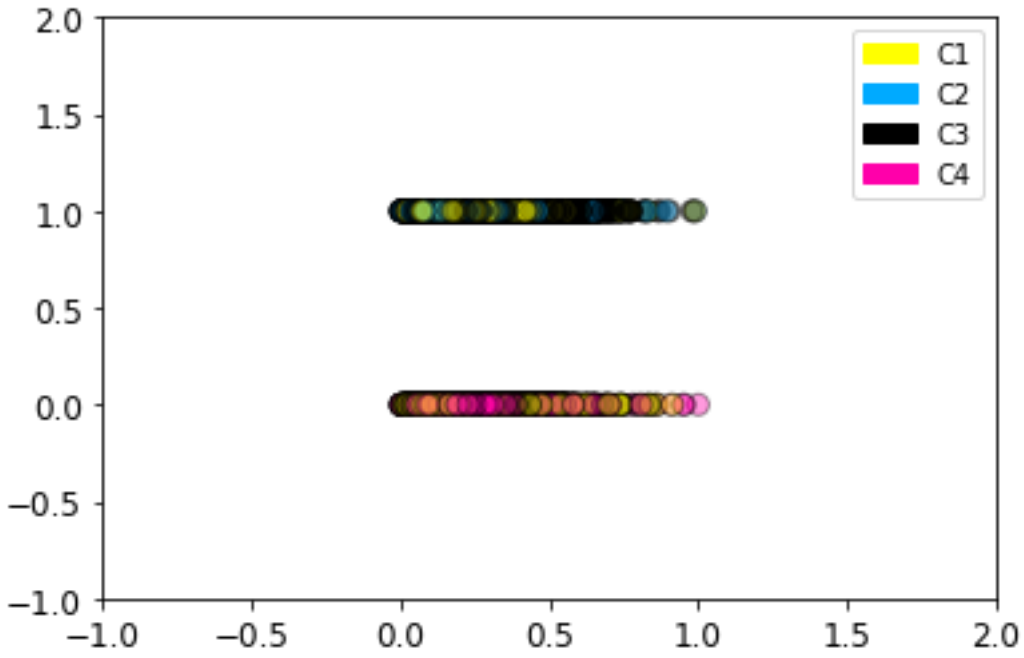
Note Looking at the silloutte measures for different batch sizes we can see that 2 , 4 and 7 .which is prominent. As we increase the batch size the intertia curve looks very similar to the one we plotted for the Kmeans algorithm. Based on both Inertia and Siloutte scores cluster size of 2 or 4 is good option.

Mini Batch KMeans Models for 4 Clusters

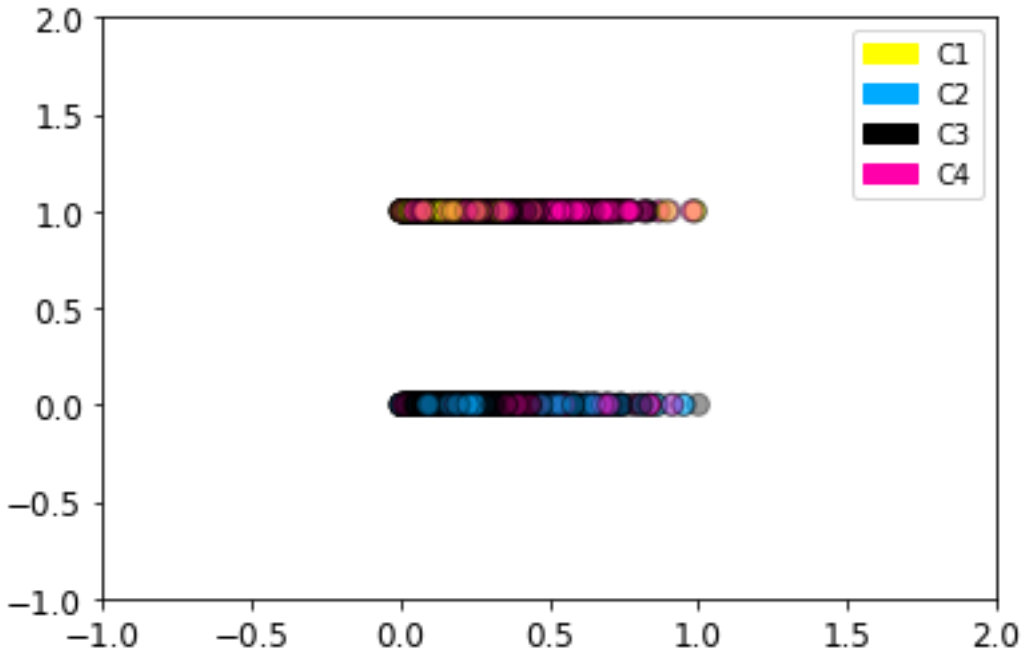
[48]:
```
# Kmeans Mini batch with 4 clusters
# Batch Size = 8
kmeans = MiniBatchKMeans(n_clusters=4,random_state=0,batch_size=8,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3','C4'])
```



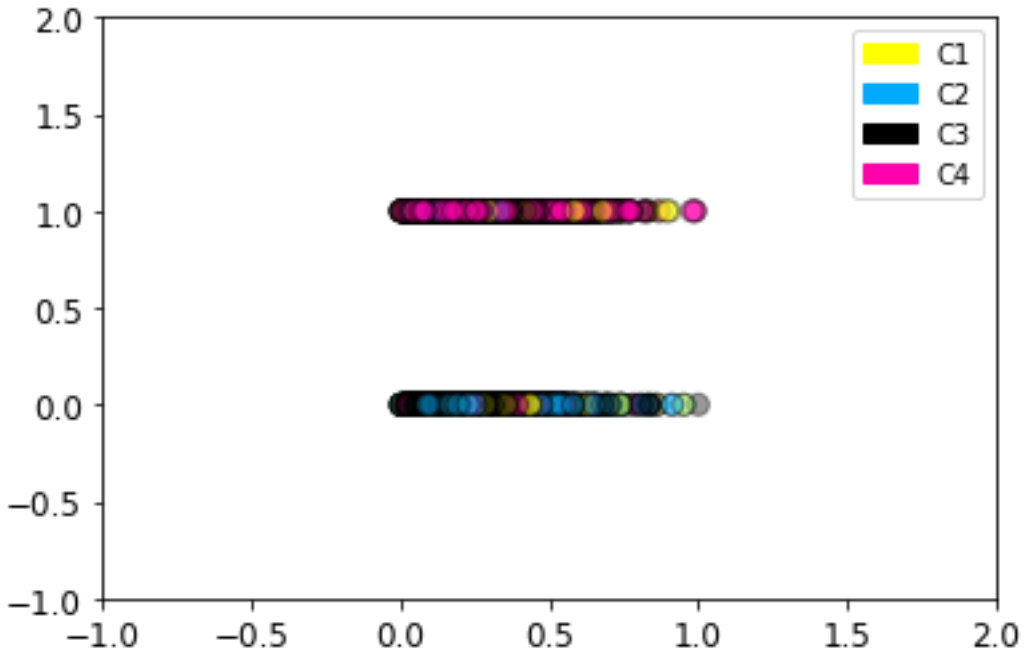
[49]:
```
# Kmeans Mini batch with 4 clusters
# Batch Size = 32
kmeans = MiniBatchKMeans(n_clusters=4,random_state=0,batch_size=32,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3','C4'])
```



[50]:
```
# Kmeans Mini batch with 4 clusters
# Batch Size = 64
kmeans = MiniBatchKMeans(n_clusters=4,random_state=0,batch_size=64,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3','C4'])
```



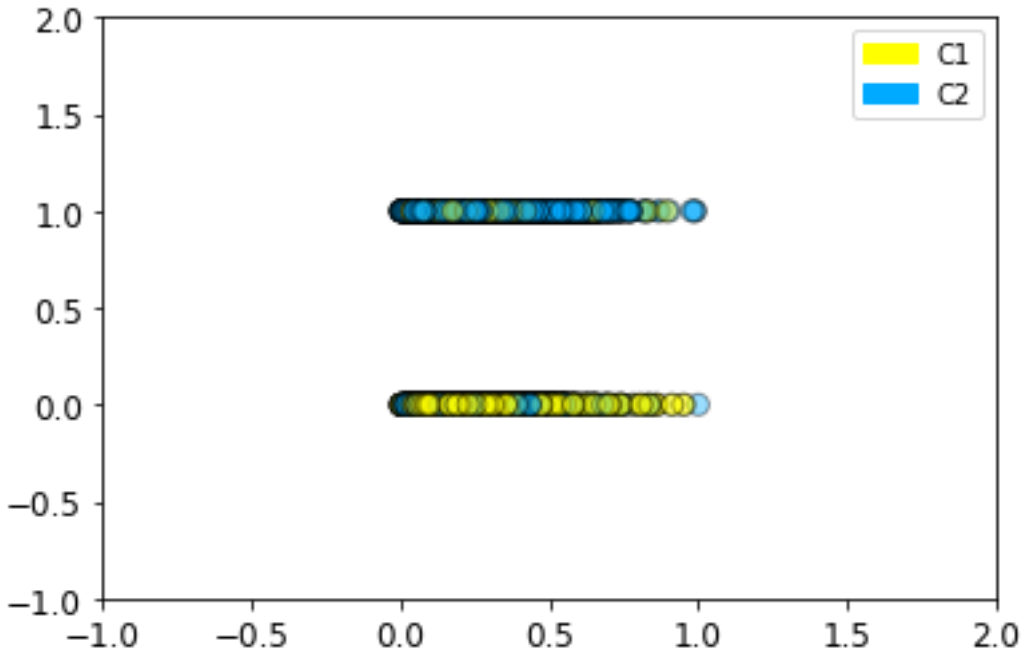
[51]:
```
# Kmeans Mini batch with 4 clusters
# Batch Size = 128
kmeans = MiniBatchKMeans(n_clusters=4,random_state=0,batch_size=128,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2','C3','C4'])
```

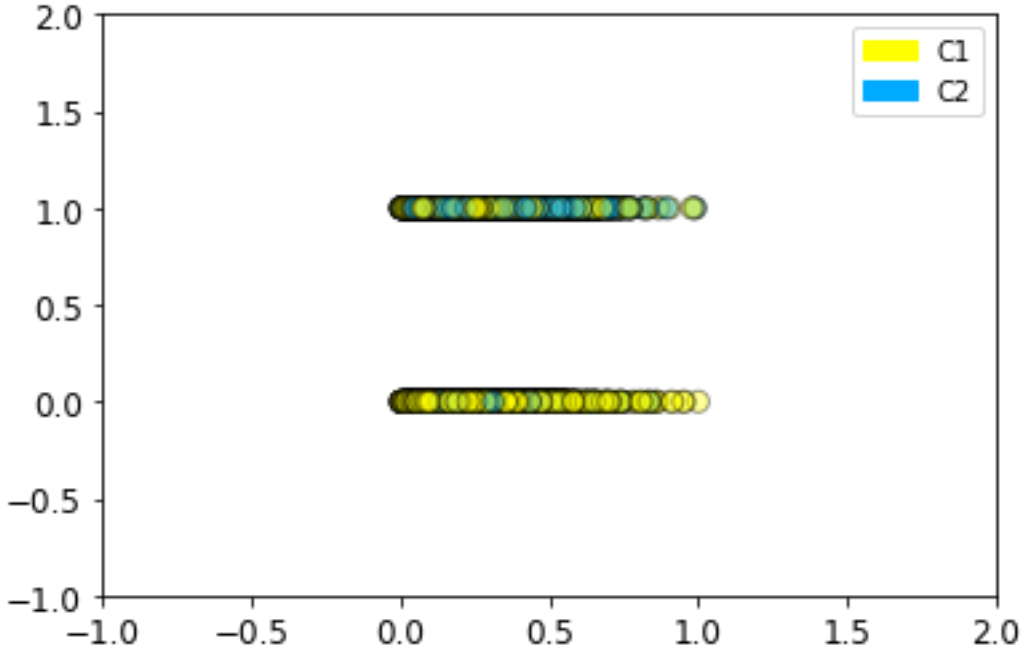


Mini Batch KMeans Models Clusters = 2

- all models are for two clusters
- diffrent batch sizes of 8,32,64 and 128
- comparing the generated labels with known labels

[52]:
```
# Kmeans Mini batch with 2 clusters
kmeans = MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=8,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])

# using actual target labels
plot_labelled_scatter(X_normalized, y_df.to_numpy(), ['C1', 'C2'])
```





[53]:
```
# find similarity between the labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
```

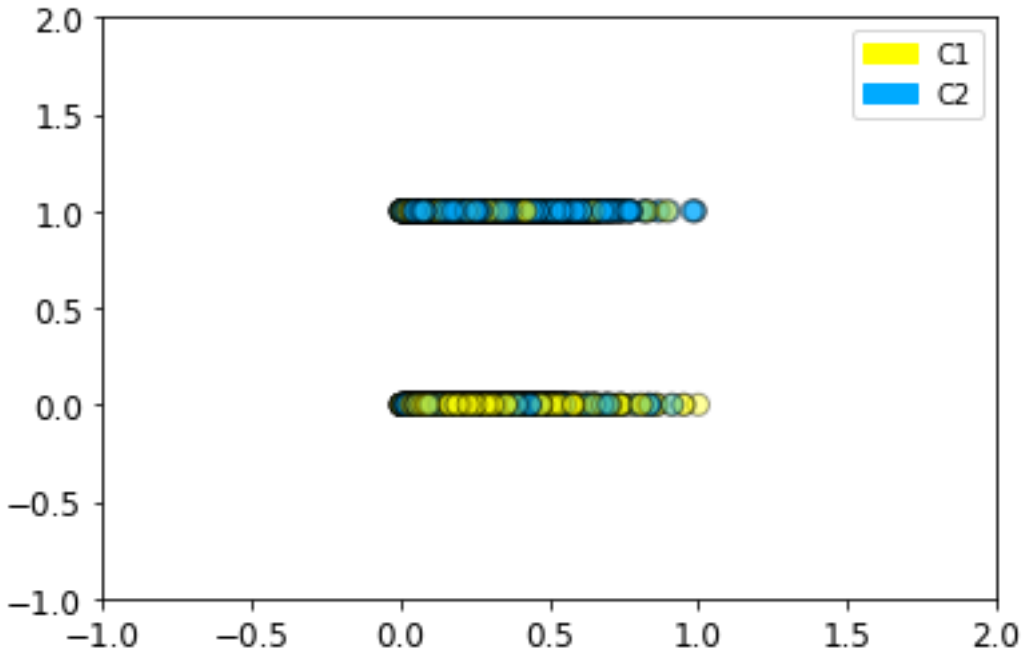
```
df_similarity = pd.concat([y_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

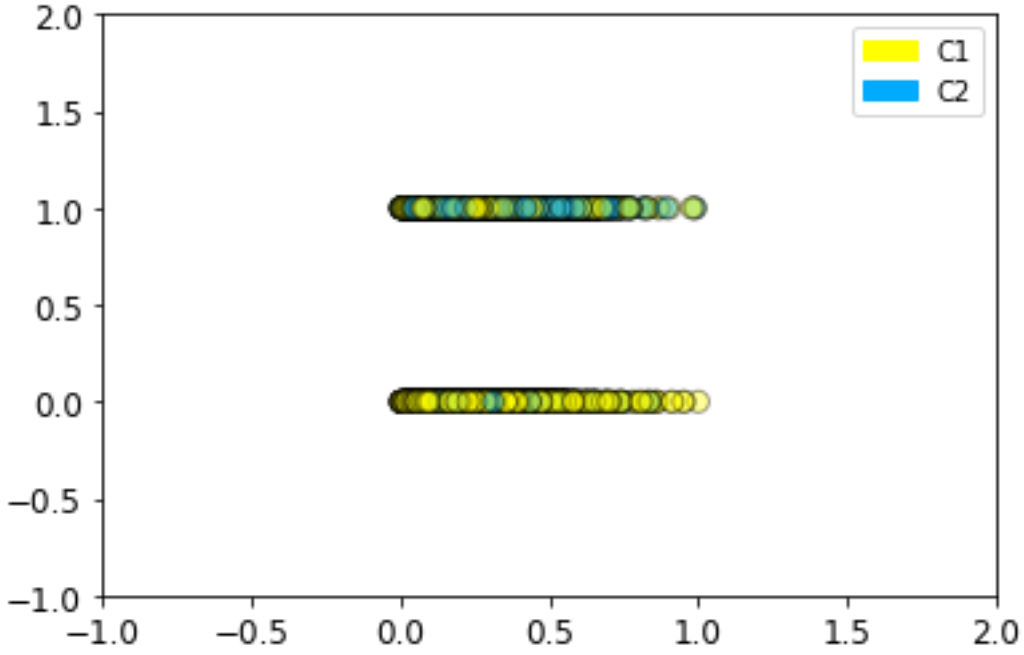
```
The percentage of datapoints that were in the cluster with similar label to
known label 69.4678135929672
```

[54]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.7107000198305901
```

[55]:
```
# Kmeans Mini batch with 2 clusters
kmeans = MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=32,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])

# using actual target labels
plot_labelled_scatter(X_normalized, y_df.to_numpy(), ['C1', 'C2'])
```





[56]:
```
# find similarity between the labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
```

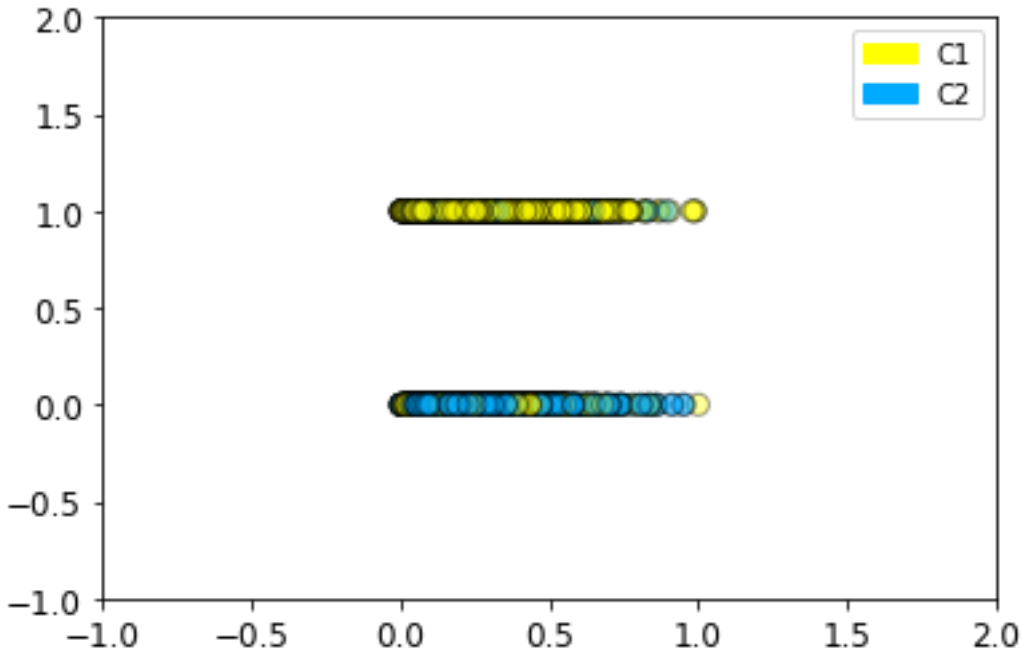
```
df_similarity = pd.concat([y_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

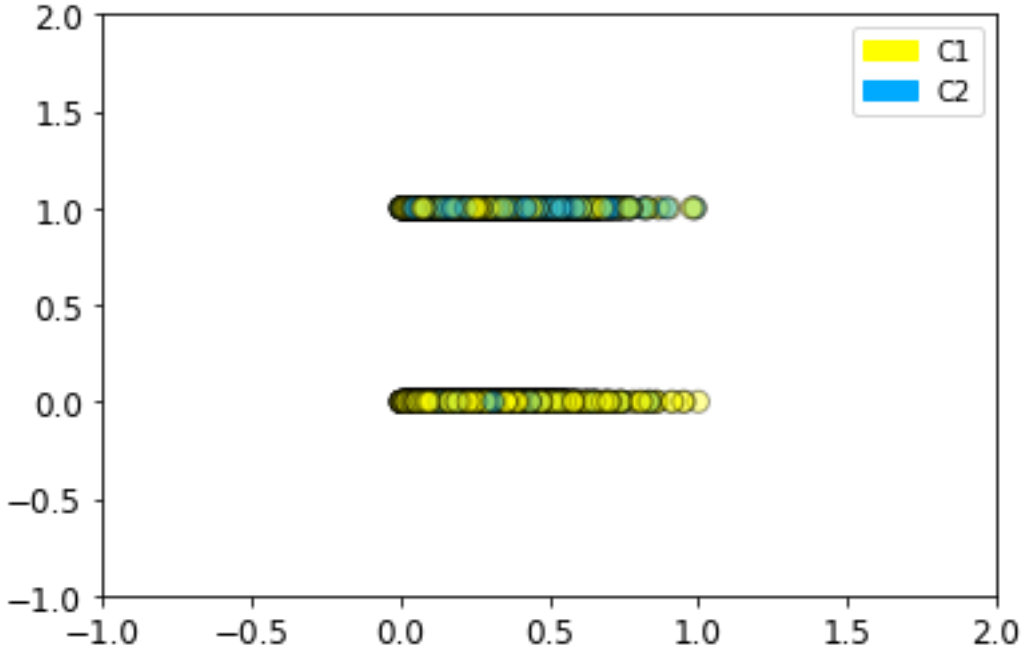
```
The percentage of datapoints that were in the cluster with similar label to
known label 73.36725341838014
```

[57]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.6292529533414545
```

[58]:
```
# Kmeans Mini batch with 2 clusters
kmeans = MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=64,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])

# using actual target labels
plot_labelled_scatter(X_normalized, y_df.to_numpy(), ['C1', 'C2'])
```





[59]:
```
# find similarity between the labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
```

```
df_similarity = pd.concat([y_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 34.09256237747465
```

Note: 34.40 here the cluster labels might be inverted vis a vis the actual labels. So We will recalculate the similarity by flipping cluster label 0 to 1 and 1 to 0.

[60]:
```
df_similarity['Cluster_Labels'].value_counts()
```

[60]:
```
1    141043
0    102274
Name: Cluster_Labels, dtype: int64
```

[61]:
```
df_similarity['Flipped_cluster_label'] = 0
df_similarity.loc[df_similarity['Cluster_Labels']==0,'Flipped_cluster_label'] =␣
 ↪1
df_similarity.head()
```

[61]:

	Diabetes	Cluster_Labels	Flipped_cluster_label
0	0.0	0	1
1	0.0	1	0
2	1.0	0	1
3	1.0	1	0
4	1.0	0	1

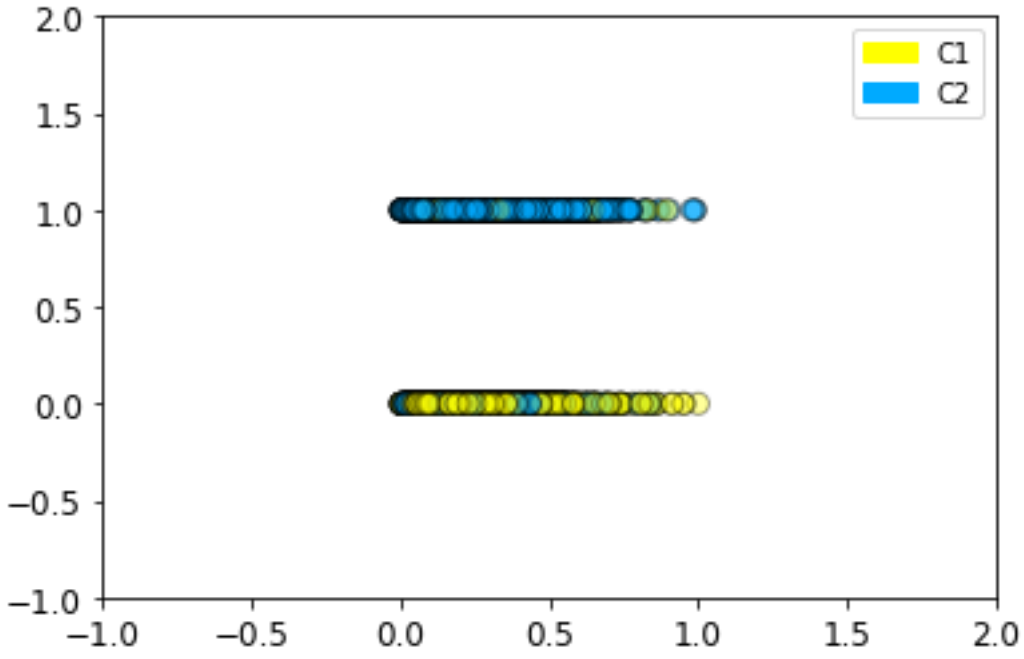
[62]:
```
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Flipped_cluster_label'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

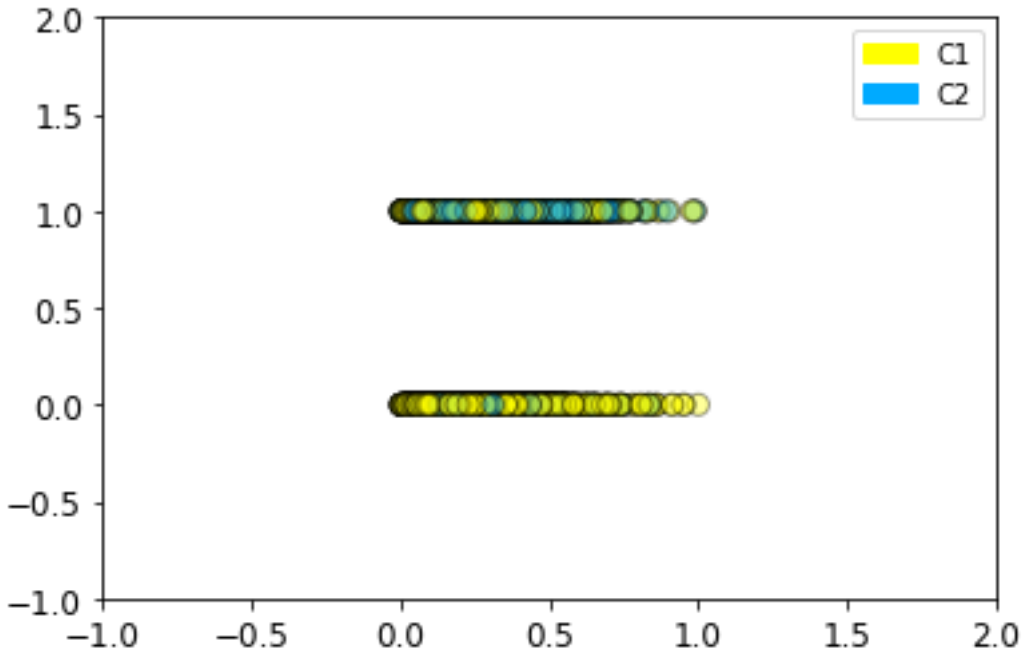
```
The percentage of datapoints that were in the cluster with similar label to
known label 65.90743762252535
```

[63]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Flipped_cluster_label'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.7736763081107113
```

[64]:
```
# Kmeans Mini batch with 2 clusters
kmeans = MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=128,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)

# cluster labels
plot_labelled_scatter(X_normalized, kmeans.labels_, ['C1', 'C2'])

# using actual target labels
plot_labelled_scatter(X_normalized, y_df.to_numpy(), ['C1', 'C2'])
```





[65]:
```
# find similarity between the labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
df_similarity = pd.concat([y_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 65.97812729895568
```

[66]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.7745545199580725
```

Note: Looking at the above similarities. Kmeans Mini Batch with Batch Size = 32 is doing a fairly good job at making the clusters. The clusters have some relation to the traget labels . At least 71 % corresponding labels and 65% pick up of label = 1 (we are more interested in positive labels).

1.6.2 8. Mean Shift Clustering

- we use estimate_bandwidth to get a intial bandwidth that we start with.

Mean Shift Clustering with Estimated Bandwidth

[67]:
```
from sklearn.cluster import MeanShift, estimate_bandwidth

# The following bandwidth can be automatically detected using estimate_bandwith
bandwidth = estimate_bandwidth(X_normalized, n_samples=5000)
bandwidth
```

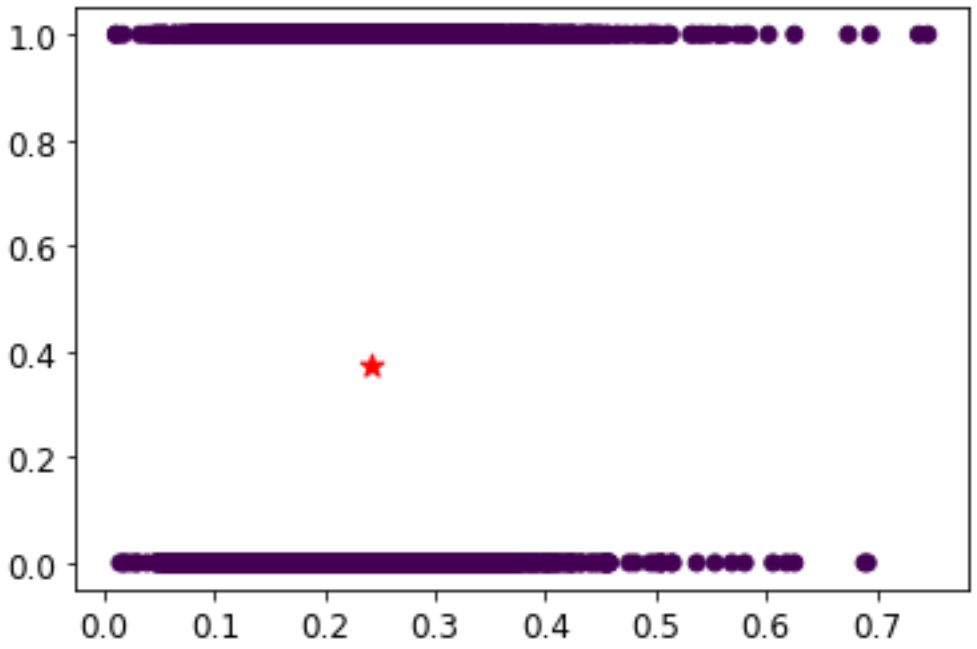
[67]:
```
1.9548072934064948
```

[68]:
```
#sampling a random number of values since plotting all 0.2 million datapoints␣
↪will crash the kernal

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True,n_jobs=-1)
ms.fit(df_sample_normalized)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)
```

```
number of estimated clusters : 1
```

[69]:
```
X_sample = df_sample.to_numpy()
plt.scatter(X_sample[:,0],X_sample[:,1],c=labels)
plt.scatter(cluster_centers[:,0],cluster_centers[:,1],marker="*",color="r",s=80)
```

[69]:
```
<matplotlib.collections.PathCollection at 0x7f82d0b6a340>
```



Note: Meanshift with the estimated bandwidth is resulting in only 1 cluster.

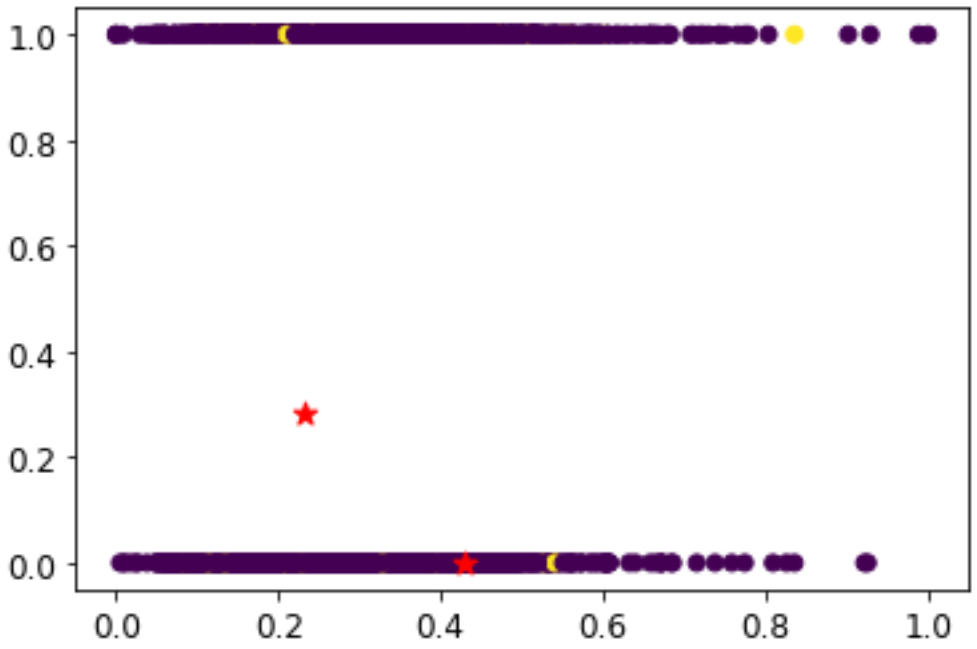
Mean Shift Model with Lower Bandwidth to Produce atleast 2 clusters

[70]:
```
 ms = MeanShift(bandwidth=1.6, bin_seeding=True,n_jobs=-1)
ms.fit(df_sample_normalized)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)
```

```
number of estimated clusters : 2
```

[71]:
```
X_sample = df_sample_normalized.to_numpy()
plt.scatter(X_sample[:,0],X_sample[:,1],c=labels)
plt.scatter(cluster_centers[:,0],cluster_centers[:,1],marker="*",color="r",s=80)
```

[71]:
```
<matplotlib.collections.PathCollection at 0x7f82e582ef40>
```



Note: We have reduced the bandwidth so that two cluster centres arise. However here the cluster centers do not make much sense

[72]:
```
# find similarity between the labels
# find similarity between the labels
df_temp = pd.DataFrame(ms.labels_, columns=["Cluster_Labels"] )
df_similarity = pd.concat([y_sample_df.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
same = df_similarity.loc[df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'],:]
total_correct = same.shape[0]/df_similarity.shape[0]*100
print("The percentage of datapoints that were in the cluster with similar label␣
 ↪to known label", total_correct)
```

```
The percentage of datapoints that were in the cluster with similar label to
known label 84.39999999999999
```

[73]:
```
correct_positive =  df_similarity.loc[(df_similarity['Diabetes'] ==␣
 ↪df_similarity['Cluster_Labels'])&(df_similarity['Diabetes'] == 1) ,:]
total_known_positive = df_similarity.loc[df_similarity['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals", percentage_positive)
```

```
Total Percentage of positive over actuals 0.028112449799196786
```

2 Conclusion

Key assumption in the comparisions between the cluster generated labels and the target labels is that C1 = 0 is no diabetes and C2 = 1 is yes diabetic. If the correspondence is less than 50% then it is possible that the cluster labels are actually reversed. We can never be sure. But in almost all our models the label matching between C1 = 0 and C2 = 1 is higher than 65% so we can try to find which model which has clusters that are relating more closely to our target labels.

The agglomerative clustering and Mean Shift clustering algorithms were causing the kernal to crash because the sized of data set is too large (0.24 million). Both meanshift and agglomerative clustering are much more computationally intensive as compared to KMeans and Mini Batch KMeans. Both Mini Batch KMeans and KMeans were able to cluster the entire dataset set is very resonable time.

In Agglomerative Clustering using the dendogram the best cluster size was 2 or 3.

MeanShift Clustering was producing cluster centers that were not modelling the dataset well.

In KMeans and MiniBatch Kmeans we found the optimal number of clusters using both inertia (elbow method) and silloutte score. Based on the cluster number analysis we felt that 2 and 4 clusters were the best cluster numbers. We did different Mini batch KMeans models with 2 , 4 clusters and batch size = 8 ,32,64,128.

It looks like 2 clusters with Kmeans or mini batch kmeans is the best. It is also better in terms of computational complexity and understandability. In the KMeans Mini Batch with batch size 32 the two clusters formed have a good but not great correspondence to the actual labels of the target (No,Yes) Diabetes - 71 % overall and 65% of positive labels were matching between the cluster created labels and actual labels

After completing the classification analysis we can further try and relate our classification results to our clustering results.

3 REFERENCES

https://www.analyticsvidhya.com/blog/2019/05/beginners-guide-hierarchical-clustering/

https://scikit-learn.org/stable/auto_examples/cluster/plot_agglomerative_dendrogram.html

https://mclguide.readthedocs.io/en/latest/sklearn/clusterdim.html

https://scikit-learn.org/stable/auto_examples/cluster/plot_mean_shift.html

https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

https://www.codecademy.com/learn/machine-learning/modules/dspath-clustering/cheatsheet

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

https://towardsdatascience.com/clustering-how-to-find-hyperparameters-using-inertia-b0343c6fe819

Material from Machine Learning Course, Seattle University

Material from Introduction to Data Science, Seattle University

4 –END–

4.1 Next Notebook -> project_part_4_Classification_NB (NaiveBayes)

project_part_4_Classification_NB

March 10, 2022

1 NOTEBOOK 5: CLASSIFICATION

1.0.1 Team 3

- Anjali Sebastian
- Yesha Sharma
- Rupansh Phutela

1.0.2 What this Notebook does?

After Data selection, cleaning, pre-processing, EDA and Regression Analysis & Clustering we will now look at how we can perform classfication on our data. Our data has target varibale y = Diabetes (Yes or No) we will try to classify the data to see the performance of different classifiers. - Normalization of entire dataset due to varying ranges of different attributes - Simple Visualization of the Dataset

CLASSIFICATION Algorithms

- Gaussian Naive Bayes
- Neural Networks(MLP)
- Support Vector Machine

Naive Bayes We first try the Gaussian Naive Bayes algorithm. Since we have 21 features, we first need to find important features with the help of importances parameter of the Decision Tree - Perform PCA on important features and reduce it to suitable dimensions - Handle imbalanced dataset with sampling - Perform Gaussian Naive Bayes classification - Change hyper parameters of Gaussian Naive Bayes and classify

1.1 1. Import Packages and Setup

[1]:
```
# you need  Python 3.5
import sys
assert sys.version_info >= (3, 5)
```

[2]:
```
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
```

[3]:
```
import os
import pandas as pd
import numpy as np
import seaborn as sns
import time
import warnings
warnings.filterwarnings("ignore")
################################################
```

[4]:
```
# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)


# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "clustering_kmeans"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

# method to save figures
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.2 2. Utility Functions

[5]:
```
import matplotlib.patches as mpatches
from matplotlib.colors import ListedColormap, BoundaryNorm

def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_labelled_scatter(X, y, class_labels):
    num_labels = len(class_labels)
```

```
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    marker_array = ['o', '^', '*']
    color_array = ['#FFFF00','#00AAFF','#000000','#FF00AA','#2ca02c',␣
↪'#d62728','#9467bd','#8c564b','#e377c2']
    cmap_bold = ListedColormap(color_array)
    bnorm = BoundaryNorm(np.arange(0, num_labels + 1, 1), ncolors=num_labels)
    plt.figure()

    plt.scatter(X[:, 0], X[:, 1], s=65, c=y, cmap=cmap_bold, norm = bnorm,␣
↪alpha = 0.40, edgecolor='black', lw = 1)

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    h = []
    for c in range(0, num_labels):
        h.append(mpatches.Patch(color=color_array[c], label=class_labels[c]))
    plt.legend(handles=h)
    plt.show()
```

[6]:
```
# a function to plot a bar graph of important features
def plot_feature_importances(clf, feature_names):
    c_features = len(feature_names)
    #plt.figure(figsize=(15,4))
    plt.figure(figsize=(8,8))
    plt.barh(range(c_features), clf.feature_importances_)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature name")
    plt.yticks(np.arange(c_features), feature_names)
```

2 CHANGE ME FOR DIABETES

[7]:
```
import numpy
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib.colors import ListedColormap, BoundaryNorm
from sklearn import neighbors
import matplotlib.patches as mpatches
import graphviz
from sklearn.tree import export_graphviz
import matplotlib.patches as mpatches
```

```
def plot_class_regions_for_classifier(clf, X, y, X_test=None, y_test=None,␣
↪title=None, target_names = None, plot_decision_regions = True):

    numClasses = numpy.amax(y) + 1
    color_list_light = ['#FFFFAA', '#EFEFEF', '#AAFFAA', '#AAAAFF']
    color_list_bold = ['#EEEE00', '#000000', '#00CC00', '#0000CC']
    print(color_list_light[0:numClasses])
    cmap_light = ListedColormap(color_list_light[0:numClasses])
    cmap_bold  = ListedColormap(color_list_bold[0:numClasses])

    h = 0.03
    k = 0.5
    x_plot_adjust = 0.1
    y_plot_adjust = 0.1
    plot_symbol_size = 50

    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()

    x2, y2 = numpy.meshgrid(numpy.arange(x_min-k, x_max+k, h), numpy.
↪arange(y_min-k, y_max+k, h))
    # numpy.c_ Translates slice objects to concatenation along the second axis
    # e.g. np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
    # ravel() Returns a contiguous flattened array.
    # x = np.array([[1, 2, 3], [4, 5, 6]])
    # np.ravel(x) = [1 2 3 4 5 6]
    P = clf.predict(numpy.c_[x2.ravel(), y2.ravel()])
    P = P.reshape(x2.shape)

    #contour region
    plt.figure()
    if plot_decision_regions:
        plt.contourf(x2, y2, P, cmap=cmap_light, alpha = 0.8)

    #train data plot
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=plot_symbol_size,␣
↪edgecolor = 'black')
    plt.xlim(x_min - x_plot_adjust, x_max + x_plot_adjust)
    plt.ylim(y_min - y_plot_adjust, y_max + y_plot_adjust)

    #test data
    if (X_test is not None):
        plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
↪s=plot_symbol_size, marker='^', edgecolor = 'black')
```

```
        train_score = clf.score(X, y)
        test_score  = clf.score(X_test, y_test)
        title = title + "\nTrain score = {:.2f}, Test score = {:.2f}".
↪format(train_score, test_score)


    if (target_names is not None):
        legend_handles = []
        for i in range(0, len(target_names)):
            patch = mpatches.Patch(color=color_list_bold[i],␣
↪label=target_names[i])
            legend_handles.append(patch)
        plt.legend(loc=0, handles=legend_handles)

    if (title is not None):
        plt.title(title)
    plt.show()
```

[8]:
```
# Show confusion matrix
def plot_confusion_matrix(confusion_mat, cln):
    plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.gray)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(cln)
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

2.1 3. Read Data and Display

[9]:
```
diabetes = pd.read_csv('./diabetes.csv')
```

[10]:
```
diabetes.head()
```

[10]:

	Unnamed: 0	Diabetes	BMI	State	HighBP	HighChol	CholCheck	\
0	0	0.0	28.17	AL	1.0	1.0	1.0	
1	1	0.0	18.54	AL	0.0	0.0	1.0	
2	2	1.0	31.62	AL	1.0	0.0	1.0	
3	6	1.0	32.98	AL	0.0	0.0	1.0	
4	9	1.0	16.65	AL	0.0	1.0	1.0	

	FruitConsume	VegetableConsume	Smoker	…	NoDoctorDueToCost	\
0	1.0	1.0	1.0	…	0.0	
1	1.0	1.0	0.0	…	0.0	
2	1.0	1.0	0.0	…	0.0	

```
3           1.0                1.0     1.0  …                0.0
4           0.0                0.0     1.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0            3.0            15.0           0.0
1               1.0            2.0            10.0           0.0
2               1.0            3.0             0.0          30.0
3               1.0            4.0            30.0           0.0
4               0.0            1.0            20.0           0.0

   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 24 columns]
```

[11]:
```
#set datatypes of columns to boolean or categorical as appropriate
make_bool_int = ['Diabetes','HighBP','HighChol','CholCheck',\
                 ␣
 ↪'FruitConsume','VegetableConsume','Smoker','HeavyDrinker','Stroke','HeartDisease',\
                 ␣
 ↪'Healthcare','NoDoctorDueToCost','PhysicalActivity','DifficultyWalking','Gender']
make_categorical_int = ['GeneralHealth','Age','Education','Income']
```

[12]:
```
#drop the extra index column in datafram
diabetes=diabetes.drop(['Unnamed: 0'], axis=1)

#drop the state column in dataframe since it will not be used in the dataframe
diabetes=diabetes.drop(['State'], axis=1)
```

[13]:
```
diabetes.head()
```

[13]:
```
   Diabetes    BMI  HighBP  HighChol  CholCheck  FruitConsume  \
0       0.0  28.17     1.0       1.0        1.0           1.0
1       0.0  18.54     0.0       0.0        1.0           1.0
2       1.0  31.62     1.0       0.0        1.0           1.0
3       1.0  32.98     0.0       0.0        1.0           1.0
4       1.0  16.65     0.0       1.0        1.0           0.0

   VegetableConsume  Smoker  HeavyDrinker  Stroke  …  NoDoctorDueToCost  \
0               1.0     1.0           0.0     0.0  …                0.0
1               1.0     0.0           0.0     0.0  …                0.0
2               1.0     0.0           0.0     0.0  …                0.0
3               1.0     1.0           0.0     0.0  …                0.0
```

```
4               0.0     1.0           0.0     0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0            3.0            15.0           0.0
1               1.0            2.0            10.0           0.0
2               1.0            3.0             0.0          30.0
3               1.0            4.0            30.0           0.0
4               0.0            1.0            20.0           0.0

   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 22 columns]
```

[14]:
```
# deep copy before next stage
df = diabetes.copy(deep = True)
```

[15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 243317 entries, 0 to 243316
Data columns (total 22 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   Diabetes            243317 non-null  float64
 1   BMI                 243317 non-null  float64
 2   HighBP              243317 non-null  float64
 3   HighChol            243317 non-null  float64
 4   CholCheck           243317 non-null  float64
 5   FruitConsume        243317 non-null  float64
 6   VegetableConsume    243317 non-null  float64
 7   Smoker              243317 non-null  float64
 8   HeavyDrinker        243317 non-null  float64
 9   Stroke              243317 non-null  float64
 10  HeartDisease        243317 non-null  float64
 11  Healthcare          243317 non-null  float64
 12  NoDoctorDueToCost   243317 non-null  float64
 13  PhysicalActivity    243317 non-null  float64
 14  GeneralHealth       243317 non-null  float64
 15  PhysicalHealth      243317 non-null  float64
 16  MentalHealth        243317 non-null  float64
 17  DifficultyWalking   243317 non-null  float64
 18  Gender              243317 non-null  float64
 19  Age                 243317 non-null  float64
```

```
 20  Education           243317 non-null  float64
 21  Income              243317 non-null  float64
dtypes: float64(22)
memory usage: 40.8 MB
```

[16]:
```
df.shape
```

[16]:
```
(243317, 22)
```

2.2 4. Normalization and Simple Vizualization

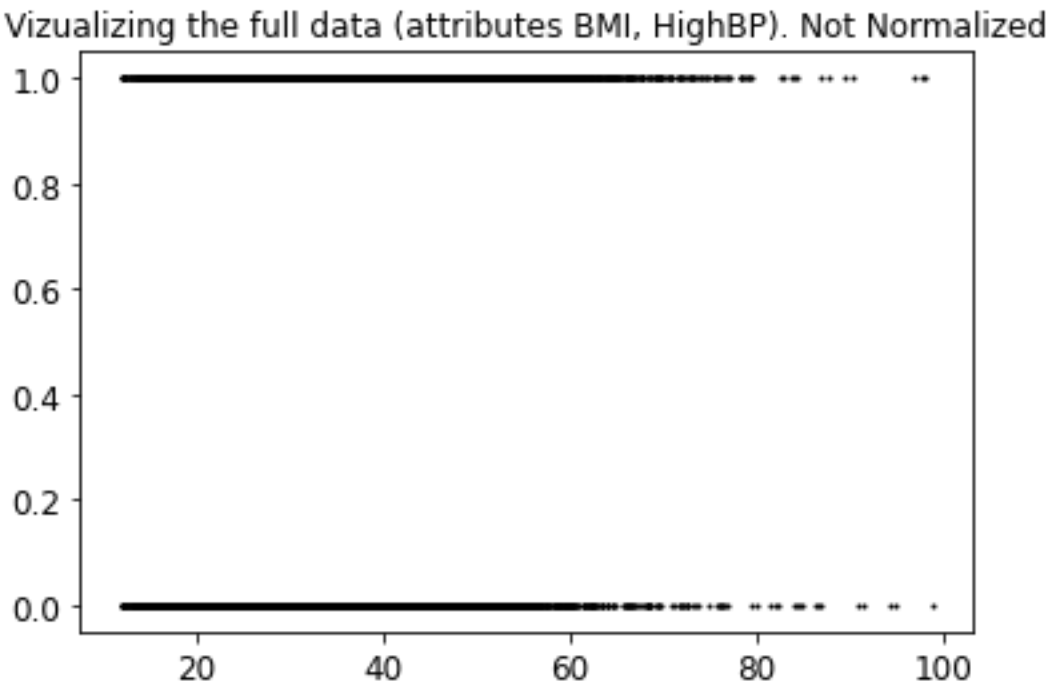
[17]:
```
X_columns = ['BMI', 'HighBP', 'HighChol', 'CholCheck', 'FruitConsume',
       'VegetableConsume', 'Smoker', 'HeavyDrinker', 'Stroke', 'HeartDisease',
       'Healthcare', 'NoDoctorDueToCost', 'PhysicalActivity', 'GeneralHealth',
       'PhysicalHealth', 'MentalHealth', 'DifficultyWalking', 'Gender', 'Age',
       'Education', 'Income']
```

Note: The entire data set is 0.24 million enteries. The agglomerative clustering and Mean Shift clustering algorithms we causing the kernal to crash because the sized of data set is too large. So we are going to take a random sample of 10,000 entries to perform clustering and see how all the clustering algorithmns perform.

[18]:
```
# Selecting a random sample for the data set

#sampling a random number of values since plotting all 0.2 million datapoints␣
 ↪will crash the kernal
number_of_samples = 10000
df_sample = df.sample(number_of_samples, random_state=42)
```

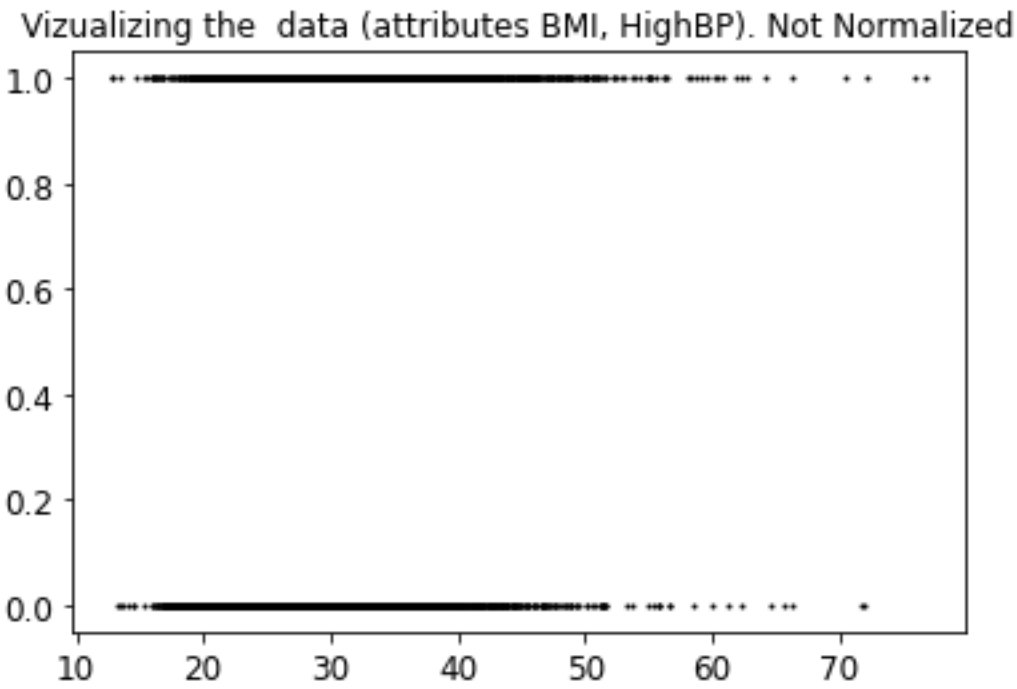
[19]:
```
# separating the target column y = Diabetes before clustering
# for complete dataset
X_df = df[X_columns].values
y_df = df[['Diabetes']]
plot_data(X_df)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Not Normalized")
```

[19]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Not
Normalized')
```



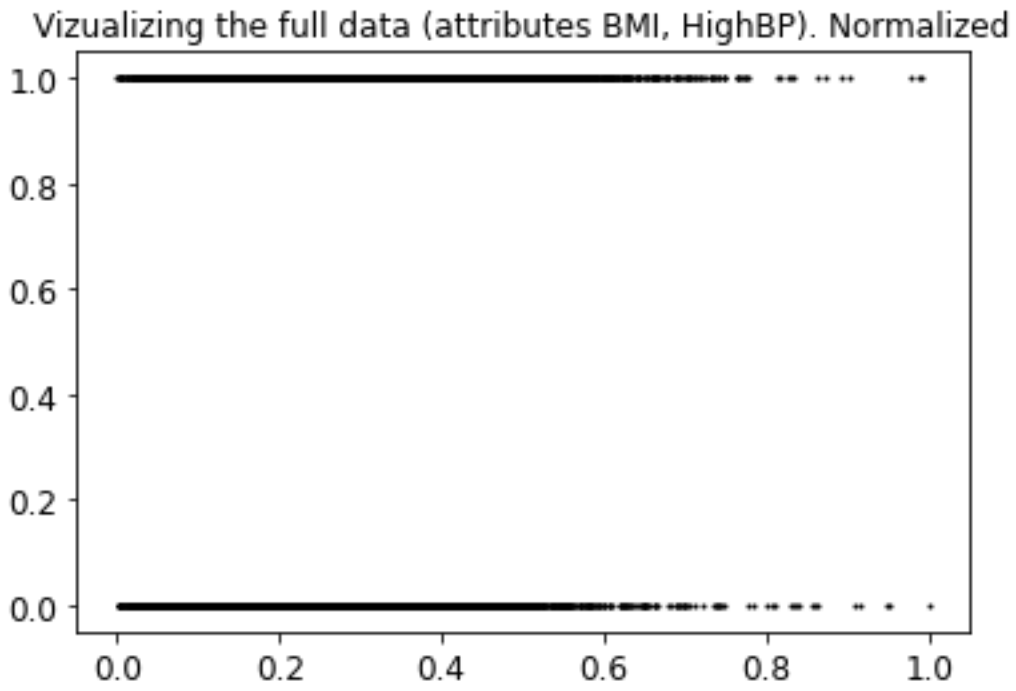
[20]:
```
# separating the target column y = Diabetes before clustering

# for sampled dataset
X_sample_df = df_sample[X_columns].values
y_sample_df = df_sample[['Diabetes']]
plot_data(X_sample_df)
plt.title("Vizualizing the  data (attributes BMI, HighBP). Not Normalized")
```

[20]:
```
Text(0.5, 1.0, 'Vizualizing the  data (attributes BMI, HighBP). Not Normalized')
```



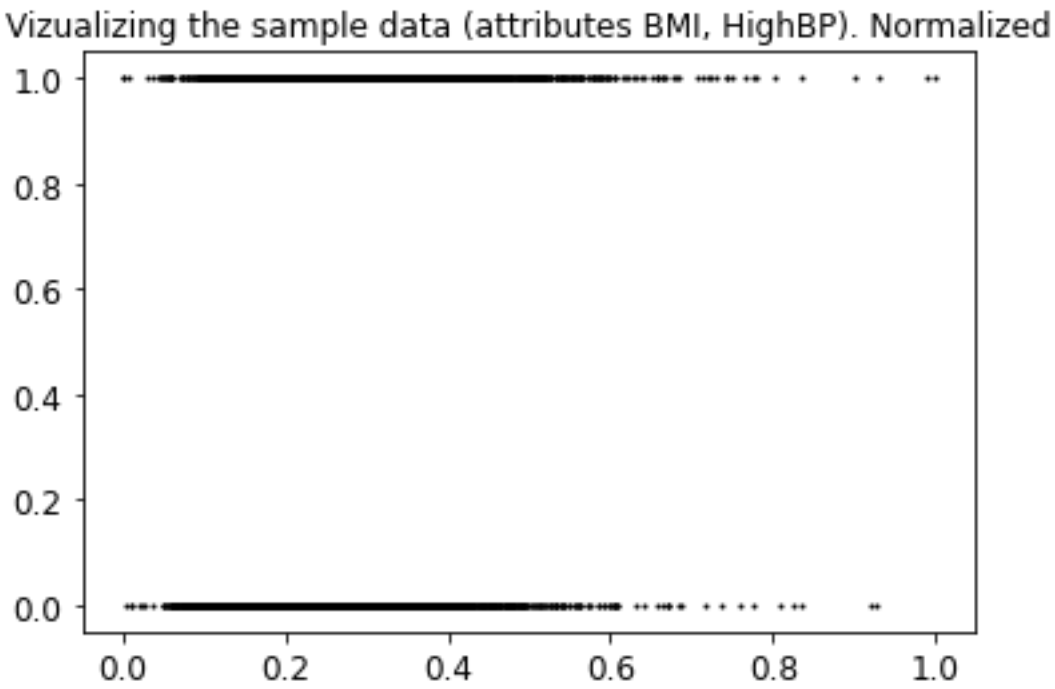
[21]:
```
# Using minmax scaler for normalization
from sklearn.preprocessing import MinMaxScaler

# normalization full dataset
X_normalized = MinMaxScaler().fit(X_df).transform(X_df)
df_normalized = pd.DataFrame(X_normalized, columns=X_columns )
plot_data(X_normalized)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Normalized")
```

[21]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Normalized')
```



[22]:
```
# normalization sample dataset
X_sample_normalized = MinMaxScaler().fit(X_sample_df).transform(X_sample_df)
df_sample_normalized = pd.DataFrame(X_sample_normalized, columns=X_columns )
plot_data(X_sample_normalized)
plt.title("Vizualizing the sample data (attributes BMI, HighBP). Normalized")


print(y_sample_df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 132646 to 192749
Data columns (total 1 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Diabetes  10000 non-null  float64
dtypes: float64(1)
memory usage: 156.2 KB
None
```



[23]:
```
# Normalized features in pandas format
df_normalized.head()
```

[23]:

	BMI	HighBP	HighChol	CholCheck	FruitConsume	VegetableConsume	\
0	0.186505	1.0	1.0	1.0	1.0	1.0	
1	0.075433	0.0	0.0	1.0	1.0	1.0	
2	0.226298	1.0	0.0	1.0	1.0	1.0	
3	0.241984	0.0	0.0	1.0	1.0	1.0	
4	0.053633	0.0	1.0	1.0	0.0	0.0	

	Smoker	HeavyDrinker	Stroke	HeartDisease	…	NoDoctorDueToCost	\
0	1.0	0.0	0.0	0.0	…	0.0	
1	0.0	0.0	0.0	0.0	…	0.0	
2	0.0	0.0	0.0	0.0	…	0.0	
3	1.0	0.0	0.0	0.0	…	0.0	
4	1.0	0.0	0.0	0.0	…	0.0	

	PhysicalActivity	GeneralHealth	PhysicalHealth	MentalHealth	\
0	0.0	0.50	0.500000	0.0	
1	1.0	0.25	0.333333	0.0	
2	1.0	0.50	0.000000	1.0	
3	1.0	0.75	1.000000	0.0	
4	0.0	0.00	0.666667	0.0	

```
   DifficultyWalking  Gender       Age  Education    Income
0                1.0     0.0  1.000000        0.4  0.285714
1                0.0     0.0  0.833333        0.8  0.571429
2                1.0     0.0  0.750000        1.0  0.857143
3                1.0     1.0  0.833333        1.0  0.857143
4                1.0     0.0  0.833333        0.2  0.285714

[5 rows x 21 columns]
```

[24]:
```
# Normalized features in numpy format
X_normalized
```

[24]:
```
array([[0.18650519, 1.        , 1.        , …, 1.        , 0.4       ,
        0.28571429],
       [0.07543253, 0.        , 0.        , …, 0.83333333, 0.8       ,
        0.57142857],
       [0.22629758, 1.        , 0.        , …, 0.75      , 1.        ,
        0.85714286],
       …,
       [0.1905421 , 0.        , 0.        , …, 0.5       , 0.4       ,
        0.        ],
       [0.227797  , 0.        , 1.        , …, 0.        , 0.6       ,
        0.28571429],
       [0.09192618, 0.        , 0.        , …, 0.33333333, 1.        ,
        1.        ]])
```

Note: The data pairs are as follows: - Full Data 1. X_df (pandas) with y_df(pandas) : not normalized full data set 2. X_normalized (numpy) with y_df(pandas) : normalized full X in numpy (easy for clustering) 3. df_normalized (pandas) with y_df(pandas) : normalized X in pandas format (easy for tracking feature names) - Sample Data of 10,000 randomly selected rows 1. X_sample_df (pandas) with y_sample_df(pandas) : not normalized sample data set 2. X_df_normalized (numpy) with y_sample_df(pandas) : normalized sample X in numpy (easy for clustering) 3. df_sample_normalized (pandas) with y_sample_df(pandas) : normalized X sample in pandas format (easy for tracking feature names)

- For all our clustering we will use only the normalized versions of the dataset.

2.3 5. Feature Importance - With Decision Tree Classifier

- We are using Decision Tree Classifier to find which feature importance to see which features are having the highest.
- We will only be using normalized data. Since it will put all features in similar range.

- We will be using tha full dataset as is . We will also we using the a balanced version of the dataset using undersampling technique to see if there is any change in the key features.

[25]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import classification_report
```

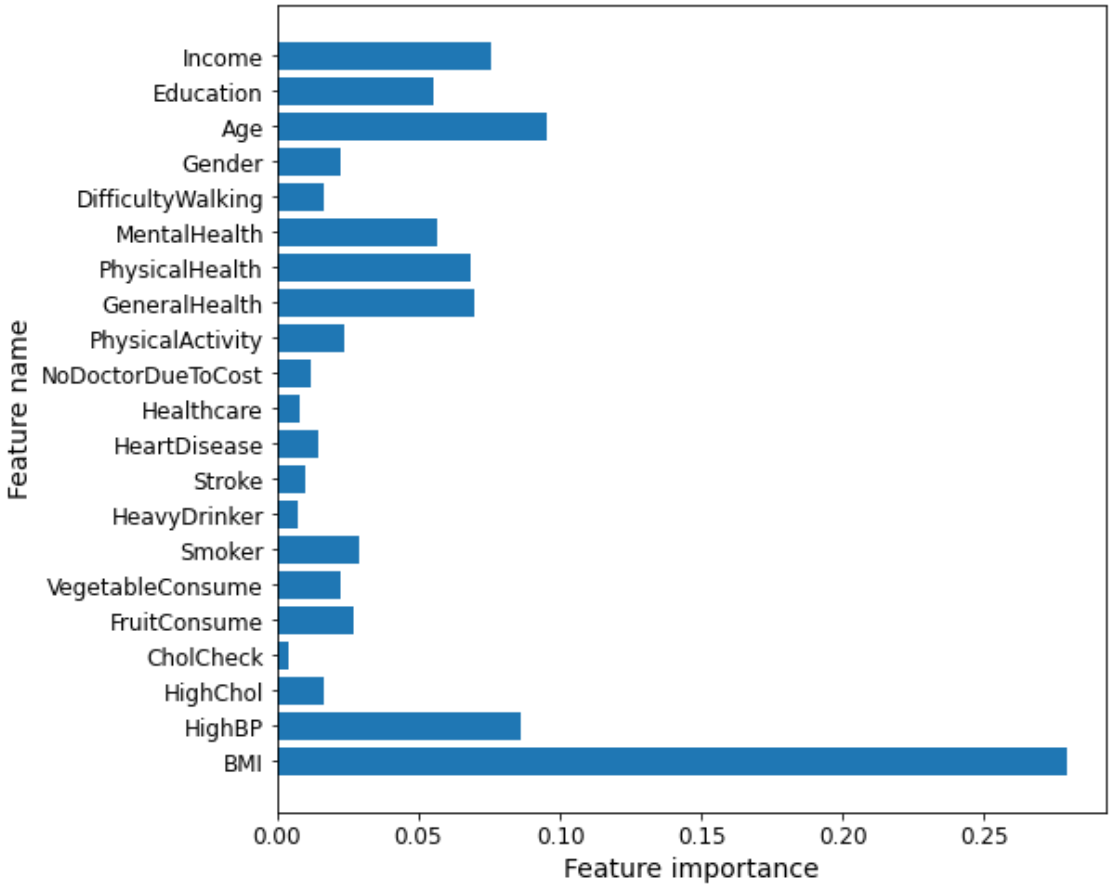
[26]:
```
X = X_normalized
y_df['Diabetes']=y_df['Diabetes'].astype('int')
y = y_df.to_numpy()
```

[27]:
```
# A simple training (1 training)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0,␣
↪test_size=0.30)
```

Using Full Dataset As Is

[28]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))

#plt.figure(figsize=(12,12), dpi=60)

# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.79
```




```
Feature importances: [0.27931751 0.08635461 0.01666336 0.00396084 0.02698384
0.02236237
 0.02883552 0.00744351 0.00987176 0.01420467 0.0082121  0.01204351
 0.02342647 0.06937859 0.06842486 0.05683353 0.01673057 0.02241037
 0.09529659 0.05551182 0.0757336 ]
```

[29]:
```
clf.score(X_test, y_test)
```

[29]:
```
0.7932900432900433
```

[30]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[30]:
```
array([[54468,  8045],
       [ 7044,  3439]], dtype=int64)
```

[31]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.89      0.87      0.88     62513
     Class 1       0.30      0.33      0.31     10483

    accuracy                           0.79     72996
   macro avg       0.59      0.60      0.60     72996
weighted avg       0.80      0.79      0.80     72996
```

Doing with a Balanced Dataset

- using random undersampler only on the training part

[32]:
```
# import RandomUndersampler
from imblearn.under_sampling import RandomUnderSampler
```

[33]:
```
X_train.shape
```

[33]: (170321, 21)

[34]:
```
under = RandomUnderSampler(sampling_strategy='auto')
X_train, y_train = under.fit_resample(X_train, y_train)
```

[35]:
```
X_train.shape
```

[35]: (49632, 21)

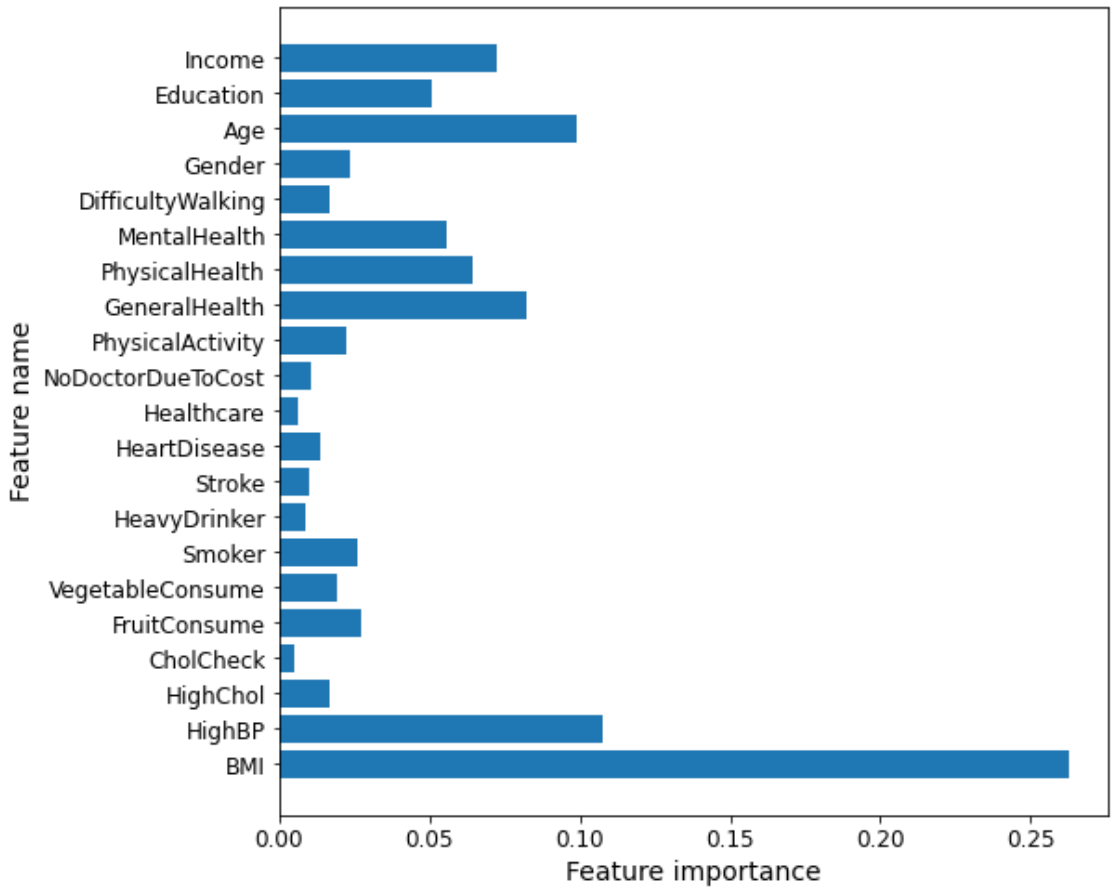
[36]:
```
unique, counts = np.unique(y_train, return_counts=True)
print ( np.asarray((unique, counts)).T)
```

```
[[    0 24816]
 [    1 24816]]
```

[37]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))
```

```
#plt.figure(figsize=(12,12), dpi=60)

# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.66
```




```
Feature importances: [0.26291005 0.10754319 0.01688605 0.00499179 0.02712826
0.01928236
 0.02616177 0.00840229 0.00992798 0.01337396 0.0062639  0.01041017
 0.02246018 0.0820788  0.06436261 0.05540956 0.01649228 0.02356799
 0.09920803 0.05057206 0.0725667 ]
```

[38]:
```
clf.score(X_test, y_test)
```

[38]:
```
0.6602553564578881
```

[39]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[39]:
```
array([[41328, 21185],
       [ 3615,  6868]], dtype=int64)
```

[40]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.92      0.66      0.77     62513
     Class 1       0.24      0.66      0.36     10483

    accuracy                           0.66     72996
   macro avg       0.58      0.66      0.56     72996
weighted avg       0.82      0.66      0.71     72996
```

Note: Looking at at the feature importance we can bar plots for both the original dataset and the balanced data set we see that the following 8 features are very important - BMI, HighBP, General Health, Physical Health, Mental Health, Age , Education and Income.

[41]:
```
important_features =␣
 ↪['BMI','HighBP','GeneralHealth','PhysicalHealth','MentalHealth','Age','Education','Income']
```

2.4 5. Principle Component Analysis

- Using the only the most important features dicovered from the decision tree model we reduce the dimensionality to 2

[42]:
```
df_normalized.head()
```

[42]:

	BMI	HighBP	HighChol	CholCheck	FruitConsume	VegetableConsume	\
0	0.186505	1.0	1.0	1.0	1.0	1.0	
1	0.075433	0.0	0.0	1.0	1.0	1.0	
2	0.226298	1.0	0.0	1.0	1.0	1.0	
3	0.241984	0.0	0.0	1.0	1.0	1.0	
4	0.053633	0.0	1.0	1.0	0.0	0.0	

```
   Smoker  HeavyDrinker  Stroke  HeartDisease  …  NoDoctorDueToCost  \
0     1.0           0.0     0.0           0.0  …                0.0
1     0.0           0.0     0.0           0.0  …                0.0
2     0.0           0.0     0.0           0.0  …                0.0
3     1.0           0.0     0.0           0.0  …                0.0
4     1.0           0.0     0.0           0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0           0.50        0.500000           0.0
1               1.0           0.25        0.333333           0.0
2               1.0           0.50        0.000000           1.0
3               1.0           0.75        1.000000           0.0
4               0.0           0.00        0.666667           0.0

   DifficultyWalking  Gender       Age  Education    Income
0                1.0     0.0  1.000000        0.4  0.285714
1                0.0     0.0  0.833333        0.8  0.571429
2                1.0     0.0  0.750000        1.0  0.857143
3                1.0     1.0  0.833333        1.0  0.857143
4                1.0     0.0  0.833333        0.2  0.285714

[5 rows x 21 columns]
```

[43]:
```
# Choose True if we are selecting only 8 top features for doing PCA else it␣
 ↪will take entire data set
select_features = True

if(select_features==True):
    df_best_features = df_normalized[important_features]
else:
    df_best_features = df_normalized
df_best_features.head()
```

[43]:
```
        BMI  HighBP  GeneralHealth  PhysicalHealth  MentalHealth       Age  \
0  0.186505     1.0           0.50        0.500000           0.0  1.000000
1  0.075433     0.0           0.25        0.333333           0.0  0.833333
2  0.226298     1.0           0.50        0.000000           1.0  0.750000
3  0.241984     0.0           0.75        1.000000           0.0  0.833333
4  0.053633     0.0           0.00        0.666667           0.0  0.833333

   Education    Income
0        0.4  0.285714
1        0.8  0.571429
2        1.0  0.857143
3        1.0  0.857143
4        0.2  0.285714
```

[44]:
```
# Dimesionality reduction to 2
from sklearn.decomposition import PCA

pca_model = PCA(n_components=2)
pca_model.fit(df_best_features) # fit the model
X_normalized_pca = pca_model.transform(df_best_features)
X_normalized_pca
```

[44]:
```
array([[ 0.8143773 ,  0.21944804],
       [-0.1775784 ,  0.39799483],
       [ 0.57868681, -0.00386865],
       …,
       [-0.3079835 ,  0.39221926],
       [-0.47663154,  0.34781812],
       [-0.51515748, -0.06059107]])
```

[45]:
```
# numpy
X_normalized_pca.shape
```

[45]:
```
(243317, 2)
```

[46]:
```
# pandafy it
df_X_normalized_pca = pd.DataFrame(X_normalized_pca,␣
 ↪columns=['Feature1','Feature2'] )
df_X_normalized_pca.head()
```

[46]:

	Feature1	Feature2
0	0.814377	0.219448
1	-0.177578	0.397995
2	0.578687	-0.003869
3	-0.225108	0.374260
4	0.051858	0.924369

2.5 6. NAIVE BAYES CLASSIFIER

- For Naive Bayes Classfier we will use the reduced features generated by PCA and also undersample the majority class (to handle the imbalanced dataset) and classify the diabetics/non-diabetics
- We only undersample the training sets because the model needs to perform with naturally imbalanced data (ie less positive diabetes cases) we leave the test sets as they are.

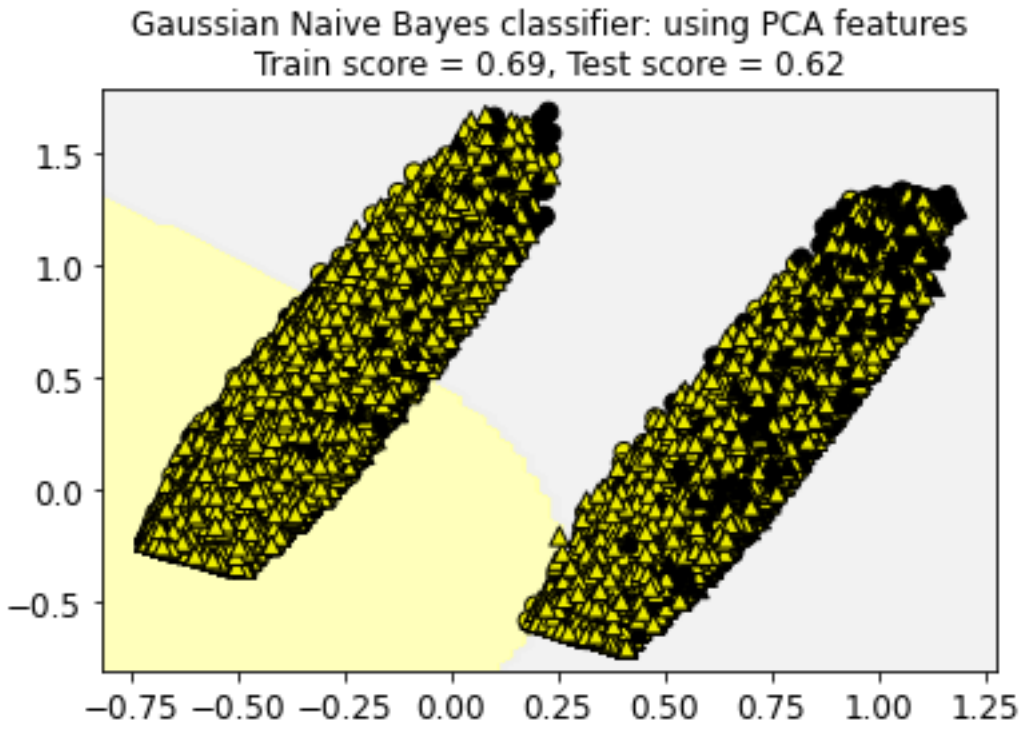
Using PCA Reduced Features (2)

[47]:
```
X = X_normalized_pca
y_df['Diabetes']=y_df['Diabetes'].astype('int')
y = y_df.to_numpy()

print(X.shape)
```

```
(243317, 2)
```

[48]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# undersampling only on the Training sets
under = RandomUnderSampler(sampling_strategy='auto')

X_train, y_train = under.fit_resample(X_train, y_train)
print(X_train.shape)

NBclf = GaussianNB().fit(X_train, y_train)

plot_class_regions_for_classifier(NBclf, X_train, y_train, X_test, y_test,␣
↪'Gaussian Naive Bayes classifier: using PCA features')
```

```
(53184, 2)
['#FFFFAA', '#EFEFEF']
```



2.6 Confusion Matrix

[49]:
```
NBclf.score(X_test, y_test)
```

[49]:
```
0.6205655104389282
```

[50]:
```
y_pred = NBclf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)

print(confusion_mat)
```

```
[[30913 21210]
 [ 1871  6836]]
```

2.7 Classification Report: Precision, Recall, F1-Score

[51]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)

print(result_metrics)
```

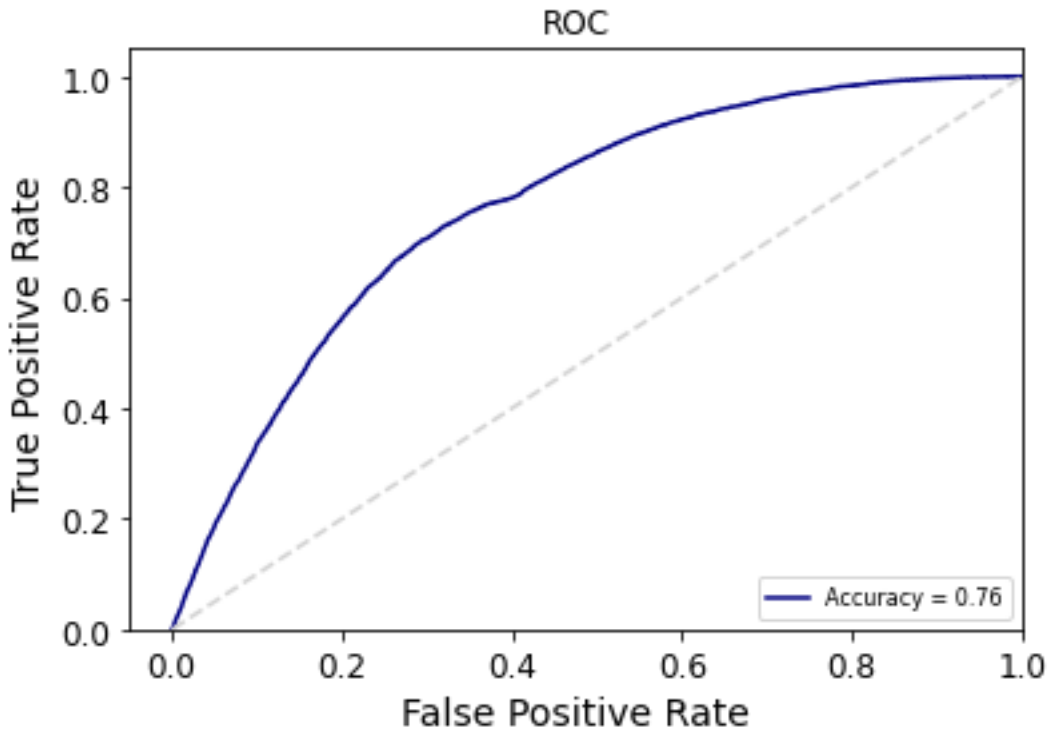
```
              precision    recall  f1-score   support

     Class 0       0.94      0.59      0.73     52123
     Class 1       0.24      0.79      0.37      8707

    accuracy                           0.62     60830
   macro avg       0.59      0.69      0.55     60830
weighted avg       0.84      0.62      0.68     60830
```

2.8 ROC Curve

[52]:
```
from sklearn.metrics import roc_curve, auc
y_score = NBclf.predict_proba(X_test)

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_score[:,1])

roc_auc = auc(false_positive_rate, true_positive_rate)
print('Accuracy = ', roc_auc)

count = 1
```

```
# Get different color each graph line
colorSet = ['navy', 'greenyellow', 'deepskyblue', 'darkviolet', 'crimson',
            'darkslategray', 'indigo', 'brown', 'orange', 'palevioletred',␣
 ↪'mediumseagreen',
            'k', 'darkgoldenrod', 'g', 'midnightblue', 'c', 'y', 'r', 'b', 'm',␣
 ↪'lawngreen'
            'mediumturquoise', 'lime', 'teal', 'drive', 'sienna', 'sandybrown']
color = colorSet[count-1]

# Plotting
plt.title('ROC')
plt.plot(false_positive_rate, true_positive_rate, c=color, label=('Accuracy =␣
 ↪%0.2f'%roc_auc))
plt.legend(loc='lower right', prop={'size':8})
plt.plot([0,1],[0,1], color='lightgrey', linestyle='--')
plt.xlim([-0.05,1.0])
plt.ylim([0.0,1.05])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
Accuracy =  0.7646406906055878
```



Precision value for Class 1 is poor, recall score 0.79 for the same class is not bad. We plotted the ROC curve to compare the TPR vs FPR. Accuracy of 0.76 is not great but since Class 1 has very low precision, we will try and change the hyper parameters and see if the performance gets better.

Naive Bayes with undersampled data and StratifiedKFold using NB params var_smoothing

[53]:
```
from sklearn.model_selection import GridSearchCV, StratifiedKFold
```

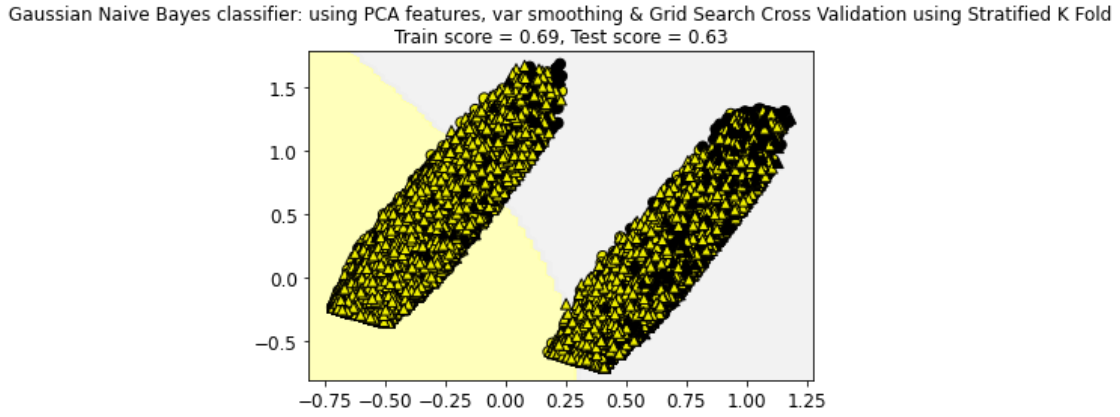
[54]:
```
nb_classifier = GaussianNB()
skf = StratifiedKFold(n_splits=7)
params_NB = {'var_smoothing': np.logspace(0,-9, num=100)}
gs_NB = GridSearchCV(estimator=nb_classifier,
                     param_grid=params_NB,
                     cv=skf,   # use any cross validation technique
                     verbose=1,
                     scoring='accuracy')
gs_NB.fit(X_train, y_train)

gs_NB.best_params_
```

```
Fitting 7 folds for each of 100 candidates, totalling 700 fits
```

[54]:
```
{'var_smoothing': 1.0}
```

[55]:
```
plot_class_regions_for_classifier(gs_NB, X_train, y_train, X_test, y_test,␣
↪'Gaussian Naive Bayes classifier: using PCA features, var smoothing & Grid␣
↪Search Cross Validation using Stratified K Fold')
```

```
['#FFFFAA', '#EFEFEF']
```



[56]:
```
gs_NB.score(X_test, y_test)
```

[56]:
```
0.6337333552523426
```

[57]:
```
y_pred = gs_NB.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)

print(confusion_mat)
```

```
[[31836 20287]
 [ 1993  6714]]
```

2.9 Classification Report: Precision, Recall, F1-Score

[58]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)

print(result_metrics)
```

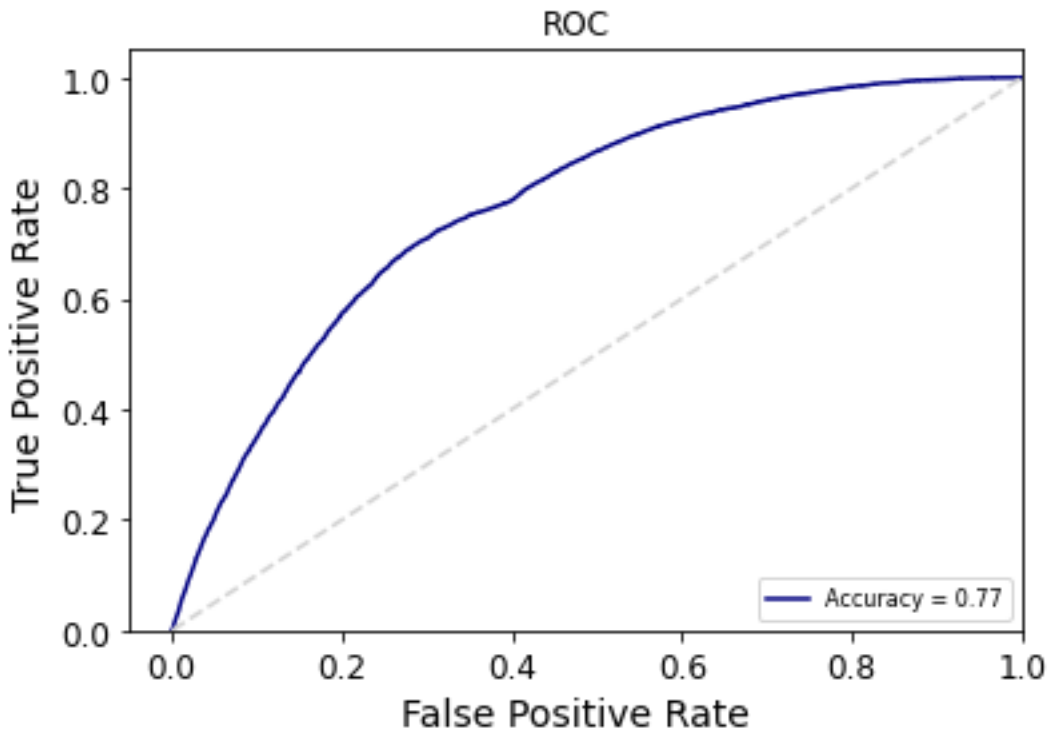
```
              precision    recall  f1-score   support

     Class 0       0.94      0.61      0.74     52123
     Class 1       0.25      0.77      0.38      8707

    accuracy                           0.63     60830
   macro avg       0.59      0.69      0.56     60830
weighted avg       0.84      0.63      0.69     60830
```

2.10 ROC Curve

[59]:
```
from sklearn.metrics import roc_curve, auc
y_score = gs_NB.predict_proba(X_test)

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_score[:,1])

roc_auc = auc(false_positive_rate, true_positive_rate)
print('Accuracy = ', roc_auc)

count = 1

# Get different color each graph line
colorSet = ['navy', 'greenyellow', 'deepskyblue', 'darkviolet', 'crimson',
            'darkslategray', 'indigo', 'brown', 'orange', 'palevioletred',␣
 ↪'mediumseagreen',
```

```
            'k', 'darkgoldenrod', 'g', 'midnightblue', 'c', 'y', 'r', 'b', 'm',␣
 ↪'lawngreen'
            'mediumturquoise', 'lime', 'teal', 'drive', 'sienna', 'sandybrown']
color = colorSet[count-1]

# Plotting
plt.title('ROC')
plt.plot(false_positive_rate, true_positive_rate, c=color, label=('Accuracy =␣
 ↪%0.2f'%roc_auc))
plt.legend(loc='lower right', prop={'size':8})
plt.plot([0,1],[0,1], color='lightgrey', linestyle='--')
plt.xlim([-0.05,1.0])
plt.ylim([0.0,1.05])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
Accuracy =  0.7685176902887391
```



The precision for class 1 improves marginally after changing the hyper parameters but the gain (0.01) is negligible. The ROC Curve looks very similar to NB without the hyper parameters change

3 Conclusion

- In the beginning, we had 21 features not all of which are important to classify the diabetes dataset.
- We first found the important features using the importances parameter of the DecisionTree Classifier.
- We then performed Principal Component Analysis and reduced the dimensions to 2
- We then undersampled the majority class to handle the data imbalance problem
- On performing Gaussian Naive Bayes on this data, we got model score as 0.76 but the precision of the Class 1 was very low (0.24), even though the ROC was not bad.
- We changed hyper parameters of Gaussian Naive Bayes, i.e., Stratified K Fold = 7 and NB params var_smoothing
- The results did not improve a lot (only 0.01) which is not promising.

We plan to train few more classifiers and then draw a conclusion on the best classifiers for the dataset.

4 End

4.1 NEXT NOTEBOOK -> project_part_4_classification_NN (Neural Networks)

project_part_4_Classification_NN

March 10, 2022

1 NOTEBOOK 6: CLASSIFICATION - Neural Network

1.0.1 Team 3

- Anjali Sebastian
- Yesha Sharma
- Rupansh Phutela

1.0.2 What this Notebook does?

After Data selection, cleaning, pre-processing, EDA and Regression Analysis, Clustering and Naive Bayes Classification we will now look at how we can perform other types of classfication on our data. Our data has target varibale y = Diabetes (Yes or No) we will try to classify the data to see the performance of different classifiers. In this Notebook we are trying various **Neural Network** Classification Models.

- Normalization of entire dataset due to varying ranges of different attributes
- Feature Importances - Identify Best Features
- Use Principle Component Analysis to reduce dimensionality of the best selected features
- Multiple Neural Network models that have different number of layer, activation functions and regularization parameters.
- Analysis of the Best Neural in terms of metrics, confusion matrix, classification report and ROC Curve
- Conclusion
- References

1.1 1. Import Packages and Setup

[1]:
```
# you need  Python 3.5
import sys
assert sys.version_info >= (3, 5)
```

[2]:
```
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
```

[3]:
```
import os
import pandas as pd
import numpy as np
```

```
import seaborn as sns
import time
import warnings
warnings.filterwarnings("ignore")
################################################
```

[4]:
```
# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)


# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "clustering_kmeans"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

# method to save figures
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.2 2. Utility Functions

[5]:
```
import matplotlib.patches as mpatches
from matplotlib.colors import ListedColormap, BoundaryNorm

def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_labelled_scatter(X, y, class_labels):
    num_labels = len(class_labels)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
    marker_array = ['o', '^', '*']
    color_array = ['#FFFF00','#00AAFF','#000000','#FF00AA','#2ca02c',␣
↪'#d62728','#9467bd','#8c564b','#e377c2']
    cmap_bold = ListedColormap(color_array)
    bnorm = BoundaryNorm(np.arange(0, num_labels + 1, 1), ncolors=num_labels)
    plt.figure()

    plt.scatter(X[:, 0], X[:, 1], s=65, c=y, cmap=cmap_bold, norm = bnorm,␣
↪alpha = 0.40, edgecolor='black', lw = 1)

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    h = []
    for c in range(0, num_labels):
        h.append(mpatches.Patch(color=color_array[c], label=class_labels[c]))
    plt.legend(handles=h)
    plt.show()
```

[6]:
```
# a function to plot a bar graph of important features
def plot_feature_importances(clf, feature_names):
    c_features = len(feature_names)
    #plt.figure(figsize=(15,4))
    plt.figure(figsize=(8,8))
    plt.barh(range(c_features), clf.feature_importances_)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature name")
    plt.yticks(np.arange(c_features), feature_names)
```

[7]:
```
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score


def plot_class_regions_for_classifier_subplot(clf, X, y, X_test, y_test, title,␣
↪subplot, target_names = None, plot_decision_regions = True):

    numClasses = np.amax(y) + 1
    color_list_light = ['#FFFFAA', '#EFEFEF', '#AAFFAA', '#AAAAFF']
    color_list_bold = ['#EEEE00', '#000000', '#00CC00', '#0000CC']
    cmap_light = ListedColormap(color_list_light[0:numClasses])
    cmap_bold  = ListedColormap(color_list_bold[0:numClasses])

    h = 0.03
    k = 0.5
    x_plot_adjust = 0.1
    y_plot_adjust = 0.1
    plot_symbol_size = 50
```

```
    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()
    x2, y2 = np.meshgrid(np.arange(x_min-k, x_max+k, h), np.arange(y_min-k,␣
↪y_max+k, h))

    P = clf.predict(np.c_[x2.ravel(), y2.ravel()])
    P = P.reshape(x2.shape)

    if plot_decision_regions:
        subplot.contourf(x2, y2, P, cmap=cmap_light, alpha = 0.8)

    subplot.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=plot_symbol_size,␣
↪edgecolor = 'black')
    subplot.set_xlim(x_min - x_plot_adjust, x_max + x_plot_adjust)
    subplot.set_ylim(y_min - y_plot_adjust, y_max + y_plot_adjust)

    if (X_test is not None):
        subplot.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
↪s=plot_symbol_size, marker='^', edgecolor = 'black')
        train_score = clf.score(X, y)
        test_score  = clf.score(X_test, y_test)
        y_pred = clf.predict(X_test)
        f1 = f1_score(y_test,y_pred)
        recall_class1 = recall_score(y_test, y_pred, average=None)[1]

        title = title + "\nTrain score = {:.2f}, Test score = {:.2f}".
↪format(train_score, test_score)
        title = title + "\nF1 score = {:.2f}, Recall Class 1 = {:.2f}".
↪format(f1,recall_class1)

    subplot.set_title(title)

    if (target_names is not None):
        legend_handles = []
        for i in range(0, len(target_names)):
            patch = mpatches.Patch(color=color_list_bold[i],␣
↪label=target_names[i])
            legend_handles.append(patch)
        subplot.legend(loc=0, handles=legend_handles)


def plot_class_regions_for_classifier(clf, X, y, X_test=None, y_test=None,␣
↪title=None, target_names = None, plot_decision_regions = True):
```

```
    numClasses = np.amax(y) + 1
    color_list_light = ['#FFFFAA', '#EFEFEF', '#AAFFAA', '#AAAAFF']
    color_list_bold = ['#EEEE00', '#000000', '#00CC00', '#0000CC']
    cmap_light = ListedColormap(color_list_light[0:numClasses])
    cmap_bold  = ListedColormap(color_list_bold[0:numClasses])

    h = 0.03
    k = 0.5
    x_plot_adjust = 0.1
    y_plot_adjust = 0.1
    plot_symbol_size = 50

    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()
    x2, y2 = np.meshgrid(np.arange(x_min-k, x_max+k, h), np.arange(y_min-k,␣
↪y_max+k, h))

    P = clf.predict(np.c_[x2.ravel(), y2.ravel()])
    P = P.reshape(x2.shape)
    plt.figure()
    if plot_decision_regions:
        plt.contourf(x2, y2, P, cmap=cmap_light, alpha = 0.8)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=plot_symbol_size,␣
↪edgecolor = 'black')
    plt.xlim(x_min - x_plot_adjust, x_max + x_plot_adjust)
    plt.ylim(y_min - y_plot_adjust, y_max + y_plot_adjust)

    if (X_test is not None):
        plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
↪s=plot_symbol_size, marker='^', edgecolor = 'black')
        train_score = clf.score(X, y)
        test_score  = clf.score(X_test, y_test)
        y_pred = clf.predict(X_test)
        f1 = f1_score(y_test,y_pred)
        recall_class1 = recall_score(y_test, y_pred, average=None)[1]

        title = title + "\nTrain score = {:.2f}, Test score = {:.2f}".
↪format(train_score, test_score)
        title = title + "\nF1 score = {:.2f},  Recall Class 1 = {:.2f}".
↪format(f1,recall_class1)

    if (target_names is not None):
        legend_handles = []
```

```
        for i in range(0, len(target_names)):
            patch = mpatches.Patch(color=color_list_bold[i],␣
↪label=target_names[i])
            legend_handles.append(patch)
        plt.legend(loc=0, handles=legend_handles)

    if (title is not None):
        plt.title(title)
    plt.show()
```

[8]:
```
# Show confusion matrix
def plot_confusion_matrix(confusion_mat, cln):
    plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.gray)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(cln)
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

1.3 3. Read Data and Display

[9]:
```
diabetes = pd.read_csv('./diabetes.csv')
```

[10]:
```
diabetes.head()
```

[10]:

	Unnamed: 0	Diabetes	BMI	State	HighBP	HighChol	CholCheck	\
0	0	0.0	28.17	AL	1.0	1.0	1.0	
1	1	0.0	18.54	AL	0.0	0.0	1.0	
2	2	1.0	31.62	AL	1.0	0.0	1.0	
3	6	1.0	32.98	AL	0.0	0.0	1.0	
4	9	1.0	16.65	AL	0.0	1.0	1.0	

	FruitConsume	VegetableConsume	Smoker	…	NoDoctorDueToCost	\
0	1.0	1.0	1.0	…	0.0	
1	1.0	1.0	0.0	…	0.0	
2	1.0	1.0	0.0	…	0.0	
3	1.0	1.0	1.0	…	0.0	
4	0.0	0.0	1.0	…	0.0	

	PhysicalActivity	GeneralHealth	PhysicalHealth	MentalHealth	\
0	0.0	3.0	15.0	0.0	
1	1.0	2.0	10.0	0.0	
2	1.0	3.0	0.0	30.0	
3	1.0	4.0	30.0	0.0	

```
4                0.0            1.0            20.0           0.0

   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 24 columns]
```

[11]:
```
#set datatypes of columns to boolean or categorical as appropriate
make_bool_int = ['Diabetes','HighBP','HighChol','CholCheck',\
                 ␣
 ↪'FruitConsume','VegetableConsume','Smoker','HeavyDrinker','Stroke','HeartDisease',\
                 ␣
 ↪'Healthcare','NoDoctorDueToCost','PhysicalActivity','DifficultyWalking','Gender']
make_categorical_int = ['GeneralHealth','Age','Education','Income']
```

[12]:
```
#drop the extra index column in datafram
diabetes=diabetes.drop(['Unnamed: 0'], axis=1)

#drop the state column in dataframe since it will not be used in the dataframe
diabetes=diabetes.drop(['State'], axis=1)
```

[13]:
```
diabetes.head()
```

[13]:
```
   Diabetes    BMI  HighBP  HighChol  CholCheck  FruitConsume  \
0       0.0  28.17     1.0       1.0        1.0           1.0
1       0.0  18.54     0.0       0.0        1.0           1.0
2       1.0  31.62     1.0       0.0        1.0           1.0
3       1.0  32.98     0.0       0.0        1.0           1.0
4       1.0  16.65     0.0       1.0        1.0           0.0

   VegetableConsume  Smoker  HeavyDrinker  Stroke  …  NoDoctorDueToCost  \
0               1.0     1.0           0.0     0.0  …                0.0
1               1.0     0.0           0.0     0.0  …                0.0
2               1.0     0.0           0.0     0.0  …                0.0
3               1.0     1.0           0.0     0.0  …                0.0
4               0.0     1.0           0.0     0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0            3.0            15.0           0.0
1               1.0            2.0            10.0           0.0
2               1.0            3.0             0.0          30.0
3               1.0            4.0            30.0           0.0
4               0.0            1.0            20.0           0.0
```

```
   DifficultyWalking  Gender   Age  Education  Income
0                1.0     0.0  13.0        3.0     3.0
1                0.0     0.0  11.0        5.0     5.0
2                1.0     0.0  10.0        6.0     7.0
3                1.0     1.0  11.0        6.0     7.0
4                1.0     0.0  11.0        2.0     3.0

[5 rows x 22 columns]
```

[14]:
```
# deep copy before next stage
df = diabetes.copy(deep = True)
```

[15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 243317 entries, 0 to 243316
Data columns (total 22 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   Diabetes            243317 non-null  float64
 1   BMI                 243317 non-null  float64
 2   HighBP              243317 non-null  float64
 3   HighChol            243317 non-null  float64
 4   CholCheck           243317 non-null  float64
 5   FruitConsume        243317 non-null  float64
 6   VegetableConsume    243317 non-null  float64
 7   Smoker              243317 non-null  float64
 8   HeavyDrinker        243317 non-null  float64
 9   Stroke              243317 non-null  float64
 10  HeartDisease        243317 non-null  float64
 11  Healthcare          243317 non-null  float64
 12  NoDoctorDueToCost   243317 non-null  float64
 13  PhysicalActivity    243317 non-null  float64
 14  GeneralHealth       243317 non-null  float64
 15  PhysicalHealth      243317 non-null  float64
 16  MentalHealth        243317 non-null  float64
 17  DifficultyWalking   243317 non-null  float64
 18  Gender              243317 non-null  float64
 19  Age                 243317 non-null  float64
 20  Education           243317 non-null  float64
 21  Income              243317 non-null  float64
dtypes: float64(22)
memory usage: 40.8 MB
```

[16]:
```
df.shape
```

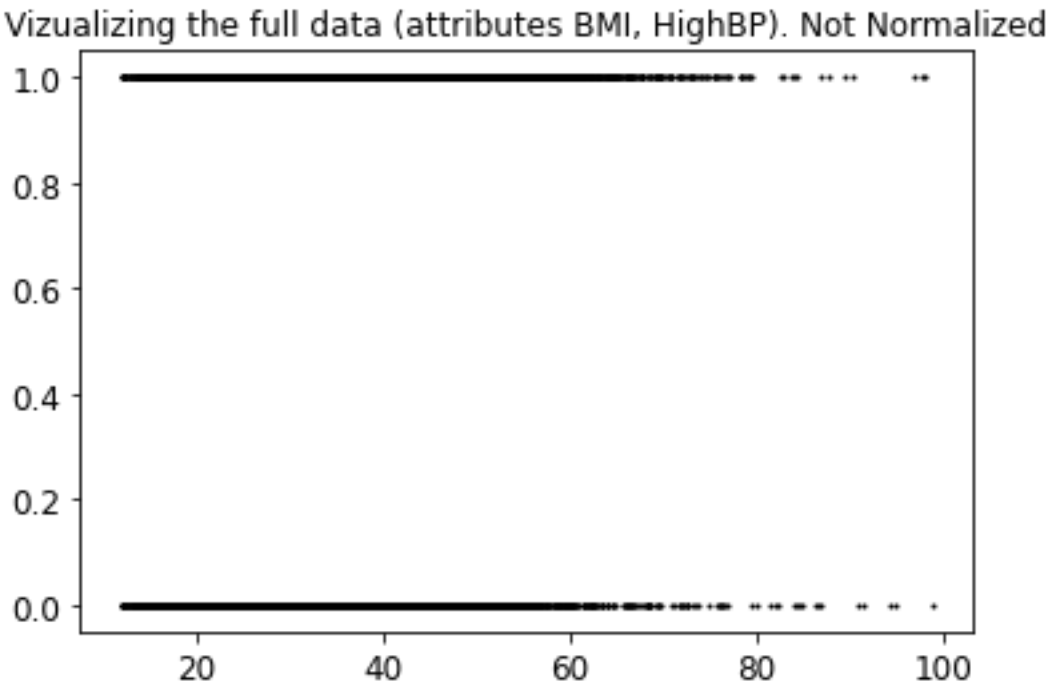
[16]: (243317, 22)

1.4 4. Normalization and Simple Vizualization

[17]:
```
X_columns = ['BMI', 'HighBP', 'HighChol', 'CholCheck', 'FruitConsume',
       'VegetableConsume', 'Smoker', 'HeavyDrinker', 'Stroke', 'HeartDisease',
       'Healthcare', 'NoDoctorDueToCost', 'PhysicalActivity', 'GeneralHealth',
       'PhysicalHealth', 'MentalHealth', 'DifficultyWalking', 'Gender', 'Age',
       'Education', 'Income']
```

[18]:
```
# separating the target column y = Diabetes before classification
# for complete dataset
X_df = df[X_columns].values
y_df = df[['Diabetes']]
plot_data(X_df)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Not Normalized")
```

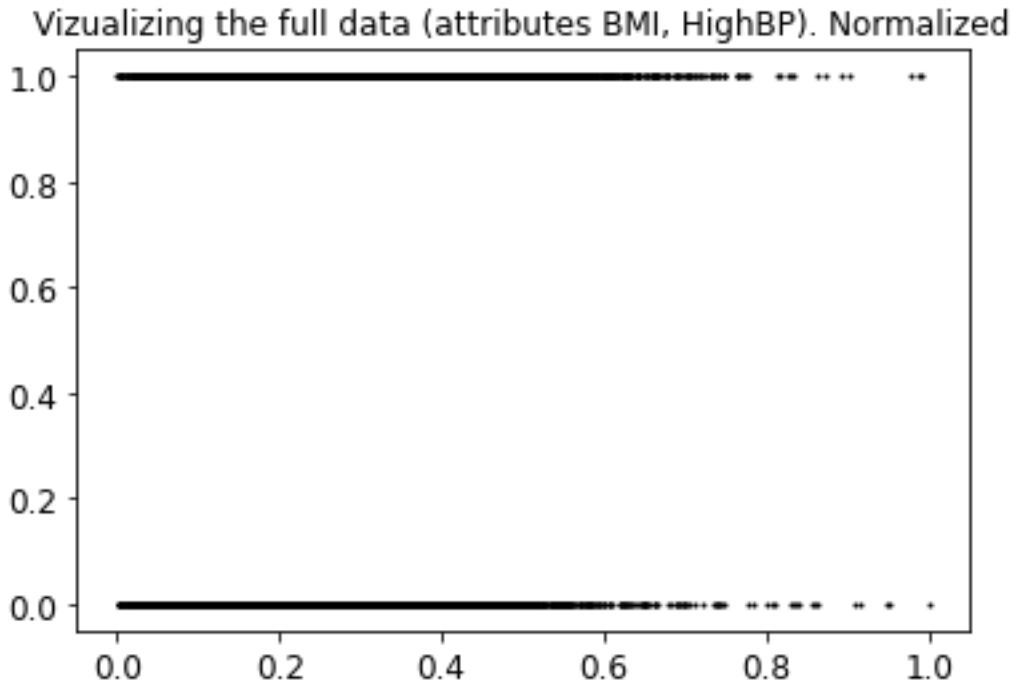
[18]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Not
Normalized')
```



[19]:
```
# Using minmax scaler for normalization
from sklearn.preprocessing import MinMaxScaler

# normalization full dataset
X_normalized = MinMaxScaler().fit(X_df).transform(X_df)
df_normalized = pd.DataFrame(X_normalized, columns=X_columns )
```

```
plot_data(X_normalized)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Normalized")
```

[19]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Normalized')
```



Note: The data pairs are as follows: - Full Data 1. X_df (pandas) with y_df(pandas) : not normalized full data set 2. X_normalized (numpy) with y_df(pandas) : normalized full X in numpy (easy for clustering) 3. df_normalized (pandas) with y_df(pandas) : normalized X in pandas format (easy for tracking feature names)

- For all our classification we will use only the normalized versions of the dataset.
- We will first pick the best features flowing which we will use PCA to reduce dataset to 2 features

1.5 5. Feature Importances - With Decision Tree Classifier

- We are using Decision Tree Classifier to find which features are more important to see which features are having the highestimpact on our target.
- We will only be using normalized data. Since it will put all features in similar range.

- We will be using the full dataset as is . We will also we using a balanced version of the dataset using undersampling technique to see if there is any change in the key features.

[20]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

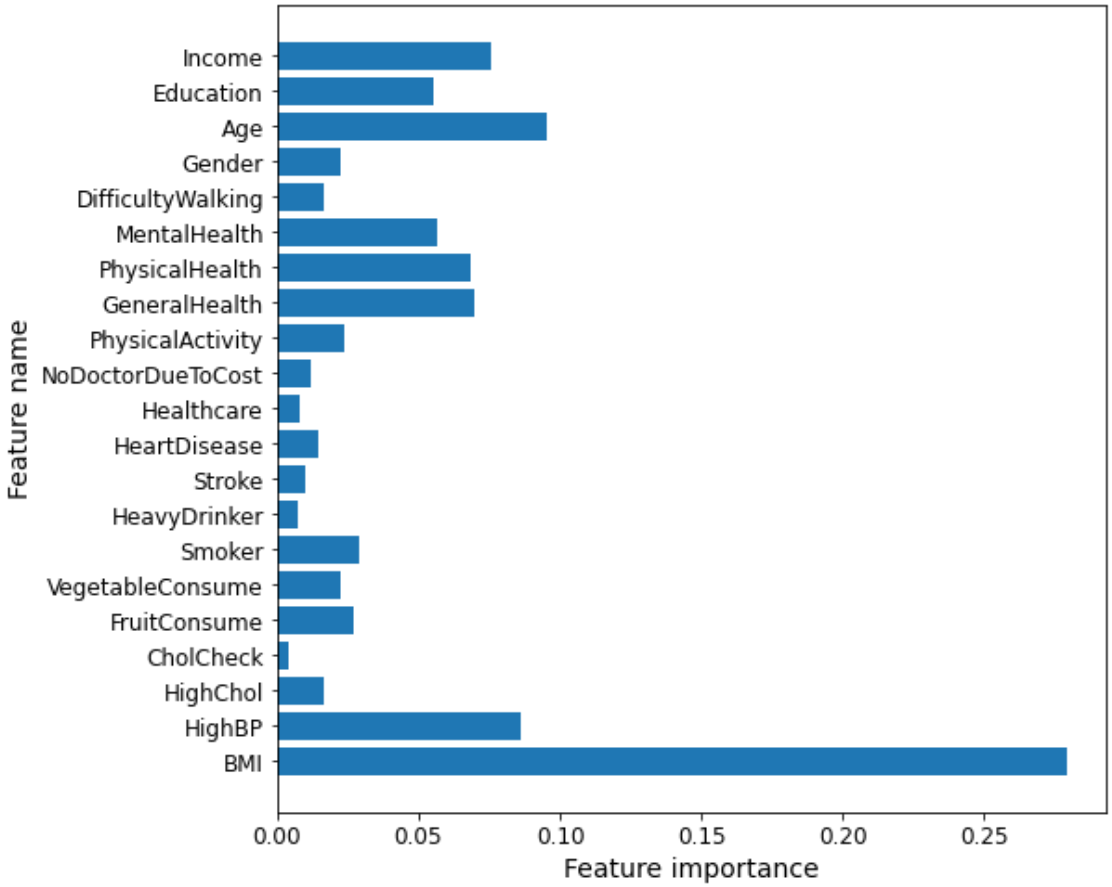
[21]:
```
X = X_normalized
y_df['Diabetes']=y_df['Diabetes'].astype('int')
y = y_df.to_numpy()
```

[22]:
```
# A simple training (1 training)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0,␣
↪test_size=0.30)
```

Using Full Dataset As Is

[23]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))

#plt.figure(figsize=(12,12), dpi=60)

# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.79
```




```
Feature importances: [0.27931751 0.08635461 0.01666336 0.00396084 0.02698384
0.02236237
 0.02883552 0.00744351 0.00987176 0.01420467 0.0082121  0.01204351
 0.02342647 0.06937859 0.06842486 0.05683353 0.01673057 0.02241037
 0.09529659 0.05551182 0.0757336 ]
```

[24]:
```
clf.score(X_test, y_test)
```

[24]:
```
0.7932900432900433
```

[25]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[25]:
```
array([[54468,  8045],
       [ 7044,  3439]], dtype=int64)
```

[26]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.89      0.87      0.88     62513
     Class 1       0.30      0.33      0.31     10483

    accuracy                           0.79     72996
   macro avg       0.59      0.60      0.60     72996
weighted avg       0.80      0.79      0.80     72996
```

Doing with a Balanced Dataset

- using random undersampler only on the training part

[27]:
```
# import RandomUndersampler
from imblearn.under_sampling import RandomUnderSampler
```

[28]:
```
X_train.shape
```

[28]:
```
(170321, 21)
```

[29]:
```
under = RandomUnderSampler(sampling_strategy='auto')
X_train, y_train = under.fit_resample(X_train, y_train)
```

[30]:
```
X_train.shape
```

[30]:
```
(49632, 21)
```

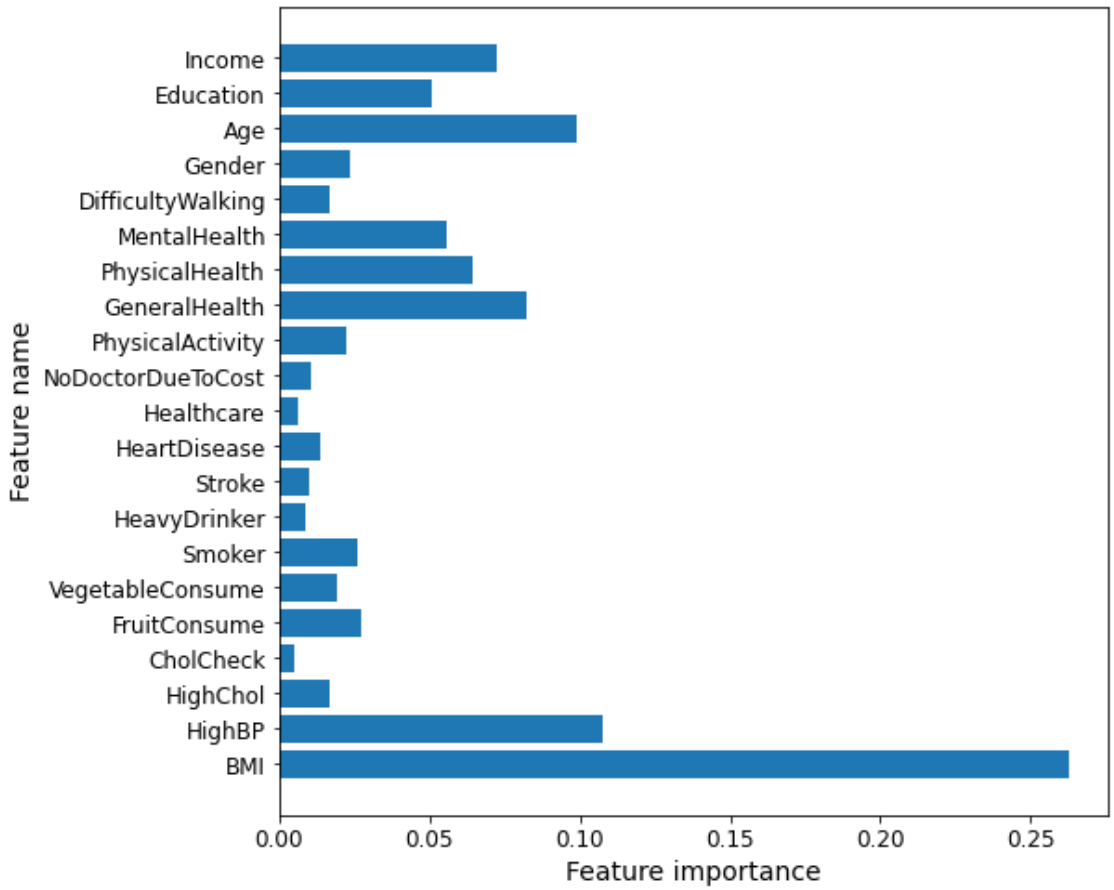
[31]:
```
unique, counts = np.unique(y_train, return_counts=True)
print ( np.asarray((unique, counts)).T)
```

```
[[    0 24816]
 [    1 24816]]
```

[32]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))
```

```
#plt.figure(figsize=(12,12), dpi=60)

# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.66
```




```
Feature importances: [0.26291005 0.10754319 0.01688605 0.00499179 0.02712826
0.01928236
 0.02616177 0.00840229 0.00992798 0.01337396 0.0062639  0.01041017
 0.02246018 0.0820788  0.06436261 0.05540956 0.01649228 0.02356799
 0.09920803 0.05057206 0.0725667 ]
```

[33]:
```
clf.score(X_test, y_test)
```

[33]:
```
0.6602553564578881
```

[34]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[34]:
```
array([[41328, 21185],
       [ 3615,  6868]], dtype=int64)
```

[35]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.92      0.66      0.77     62513
     Class 1       0.24      0.66      0.36     10483

    accuracy                           0.66     72996
   macro avg       0.58      0.66      0.56     72996
weighted avg       0.82      0.66      0.71     72996
```

Note: Looking at at the feature importance we can see that the bar plots for both the original dataset and the balanced data set are having similar patterns. We see that the following 8 features are very important - BMI, HighBP, General Health, Physical Health, Mental Health, Age , Education and Income.

[36]:
```
# Create a list of important features
important_features =␣
 ↪['BMI','HighBP','GeneralHealth','PhysicalHealth','MentalHealth','Age','Education','Income']
```

1.6 5. Principle Component Analysis

- Using the only the most important features dicovered from the decision tree model we reduce the dimensionality to 2 using Principal Component Analysis

[37]:
```
df_normalized.head()
```

[37]:

	BMI	HighBP	HighChol	CholCheck	FruitConsume	VegetableConsume	\
0	0.186505	1.0	1.0	1.0	1.0	1.0	
1	0.075433	0.0	0.0	1.0	1.0	1.0	
2	0.226298	1.0	0.0	1.0	1.0	1.0	
3	0.241984	0.0	0.0	1.0	1.0	1.0	

```
4  0.053633     0.0       1.0        1.0           0.0                0.0

   Smoker  HeavyDrinker  Stroke  HeartDisease  …  NoDoctorDueToCost  \
0     1.0           0.0     0.0           0.0  …                0.0
1     0.0           0.0     0.0           0.0  …                0.0
2     0.0           0.0     0.0           0.0  …                0.0
3     1.0           0.0     0.0           0.0  …                0.0
4     1.0           0.0     0.0           0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0           0.50        0.500000           0.0
1               1.0           0.25        0.333333           0.0
2               1.0           0.50        0.000000           1.0
3               1.0           0.75        1.000000           0.0
4               0.0           0.00        0.666667           0.0

   DifficultyWalking  Gender       Age  Education    Income
0                1.0     0.0  1.000000        0.4  0.285714
1                0.0     0.0  0.833333        0.8  0.571429
2                1.0     0.0  0.750000        1.0  0.857143
3                1.0     1.0  0.833333        1.0  0.857143
4                1.0     0.0  0.833333        0.2  0.285714

[5 rows x 21 columns]
```

[38]:
```
# Choose True if we are selecting only 8 top features for doing PCA else it␣
 ↪will take entire data set
select_features = True

if(select_features==True):
    df_best_features = df_normalized[important_features]
else:
    df_best_features = df_normalized
df_best_features.head()
```

[38]:
```
        BMI  HighBP  GeneralHealth  PhysicalHealth  MentalHealth       Age  \
0  0.186505     1.0           0.50        0.500000           0.0  1.000000
1  0.075433     0.0           0.25        0.333333           0.0  0.833333
2  0.226298     1.0           0.50        0.000000           1.0  0.750000
3  0.241984     0.0           0.75        1.000000           0.0  0.833333
4  0.053633     0.0           0.00        0.666667           0.0  0.833333

   Education    Income
0        0.4  0.285714
1        0.8  0.571429
2        1.0  0.857143
3        1.0  0.857143
```

```
4        0.2  0.285714
```

[39]:
```
# Dimesionality reduction to 2
from sklearn.decomposition import PCA

pca_model = PCA(n_components=2)
pca_model.fit(df_best_features) # fit the model
X_normalized_pca = pca_model.transform(df_best_features)
X_normalized_pca
```

[39]:
```
array([[ 0.8143773 ,  0.21944804],
       [-0.1775784 ,  0.39799483],
       [ 0.57868681, -0.00386865],
       …,
       [-0.3079835 ,  0.39221926],
       [-0.47663154,  0.34781812],
       [-0.51515748, -0.06059107]])
```

[40]:
```
# numpy
X_normalized_pca.shape
```

[40]:
```
(243317, 2)
```

[41]:
```
# panda it
df_X_normalized_pca = pd.DataFrame(X_normalized_pca,␣
 ↪columns=['Feature1','Feature2'] )
df_X_normalized_pca.head()
```

[41]:

	Feature1	Feature2
0	0.814377	0.219448
1	-0.177578	0.397995
2	0.578687	-0.003869
3	-0.225108	0.374260
4	0.051858	0.924369

Note: We have reduced our datasets dimensionality to 2 features which have just been named feature1 and feature2. Going ahead we will be using these two synthetic features to perform our classification.

1.7 6. Neural Networks

- Neural Network is very computationaly intensive so we will take only a part of the dataset to run the classification on (around 10,000)
- For Neural Neworks we will try different X variables and classify the diabetics/non-diabetics.

[42]:
```
# attach back the labels before sampling
df_normalized_pca = pd.concat([df_X_normalized_pca.reset_index(drop=True), y_df.
 ↪reset_index(drop=True)], axis= 1)
```

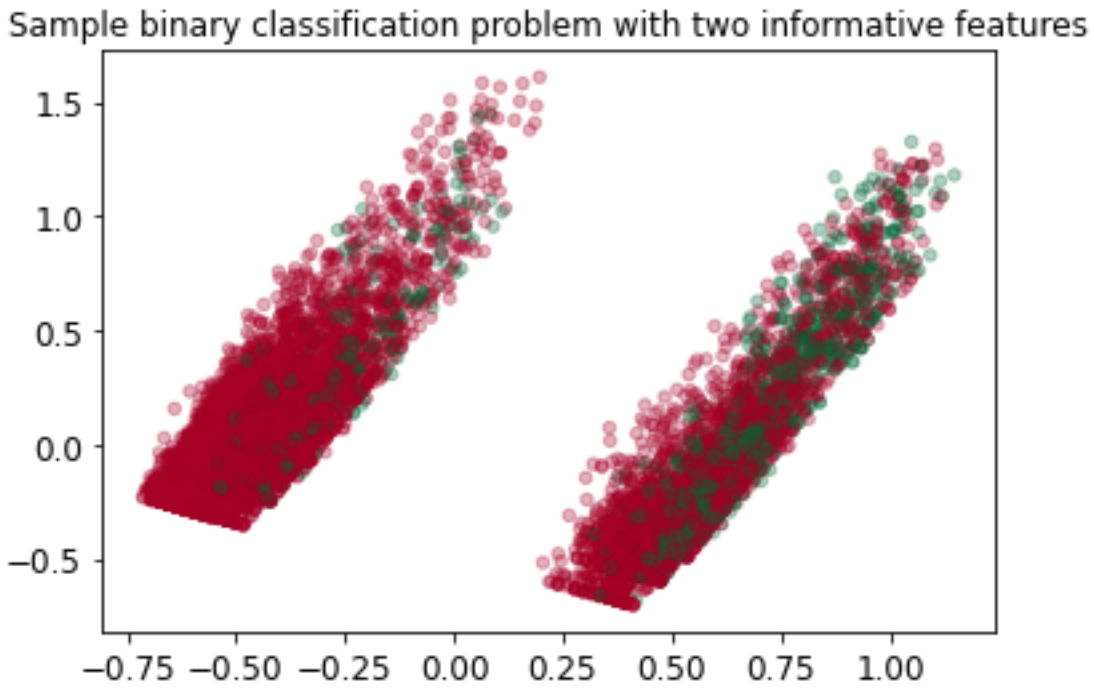
```
df_normalized_pca.head()
```

[42]:

	Feature1	Feature2	Diabetes
0	0.814377	0.219448	0
1	-0.177578	0.397995	0
2	0.578687	-0.003869	1
3	-0.225108	0.374260	1
4	0.051858	0.924369	1

[43]:
```
# Selecting a random sample for the data set
#sampling a random number of values since neural net classification of all 0.2 million datapoints is too slow
# option 10000 , 50000 etc.
number_of_samples = 10000
sample_normalized_pca = df_normalized_pca.sample(number_of_samples,random_state=42)
```

[44]:
```
# plotting the 2 attributes of PCA
plt.figure()
plt.title('Sample binary classification problem with two informative features')

#Plotting just 5000 points to not clutter the scatter plot
plt.scatter(sample_normalized_pca.iloc[:, 0], sample_normalized_pca.iloc[:, 1],alpha = 0.3,cmap=plt.cm.RdYlGn,marker= 'o', s=20, c=sample_normalized_pca.loc[:,'Diabetes'])
plt.show()
```



[45]:
```
# set up the Data
X = sample_normalized_pca.iloc[:,[0,1]].to_numpy()
y = sample_normalized_pca.iloc[:,[2]].to_numpy()
print(X.shape)
```

```
(10000, 2)
```

[46]:
```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.neural_network import MLPClassifier

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
↪random_state = 42)

# undersampling only on the Training sets
under = RandomUnderSampler(sampling_strategy='auto')
X_train, y_train = under.fit_resample(X_train, y_train)
print(X_train.shape)
```

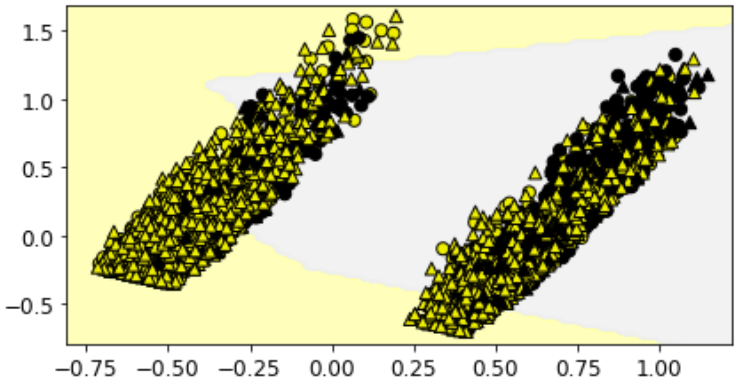
```
(2168, 2)
```

1.7.1 MODEL 1: 2 Layers with Default Activation (Relu) No Regularization

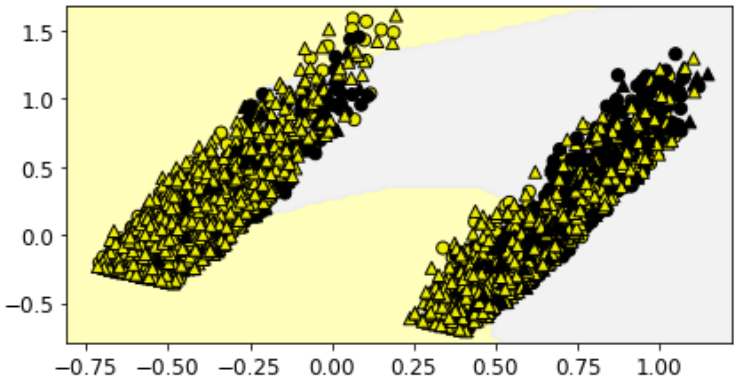
- We are using normalized data that was transformed by PCA to 2 features
- We will use under sampling technique since we are more interested in positive cases
- We only undersample the training sets because the model needs to perform with naturally imbalanced data (ie less positive diabetes cases) we leave the test sets as they are.
- Try different number of units in the layers

[47]:
```
fig, subaxes = plt.subplots(4, 1, figsize=(6, 15))
for units, axis in zip([10,20,50,100], subaxes):
    # training the data
    nnclf = MLPClassifier(hidden_layer_sizes = [units,units],
                          solver='lbfgs', random_state=42).fit(X_train,y_train)
    title = 'Normalized Data: NN classifier, No Regularization,  2 layers, {:.
↪0f}/{:.0f} units'.format(units,units)
    plot_class_regions_for_classifier_subplot(nnclf, X_train, y_train,X_test,␣
↪y_test, title, axis)
    plt.tight_layout()
```

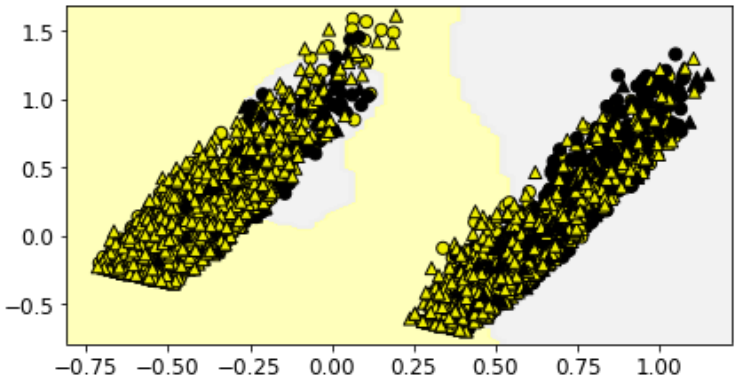
Normalized Data: NN classifier, No Regularization, 2 layers, 10/10 units
Train score = 0.73, Test score = 0.65
F1 score = 0.42, Recall Class 1 = 0.76



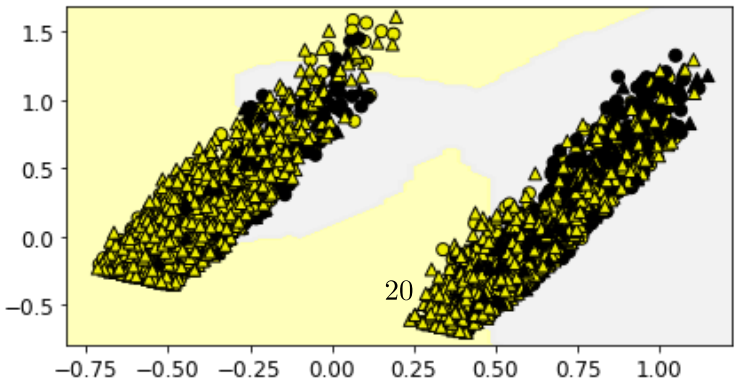
Normalized Data: NN classifier, No Regularization, 2 layers, 20/20 units
Train score = 0.74, Test score = 0.68
F1 score = 0.43, Recall Class 1 = 0.72



Normalized Data: NN classifier, No Regularization, 2 layers, 50/50 units
Train score = 0.74, Test score = 0.68
F1 score = 0.43, Recall Class 1 = 0.73



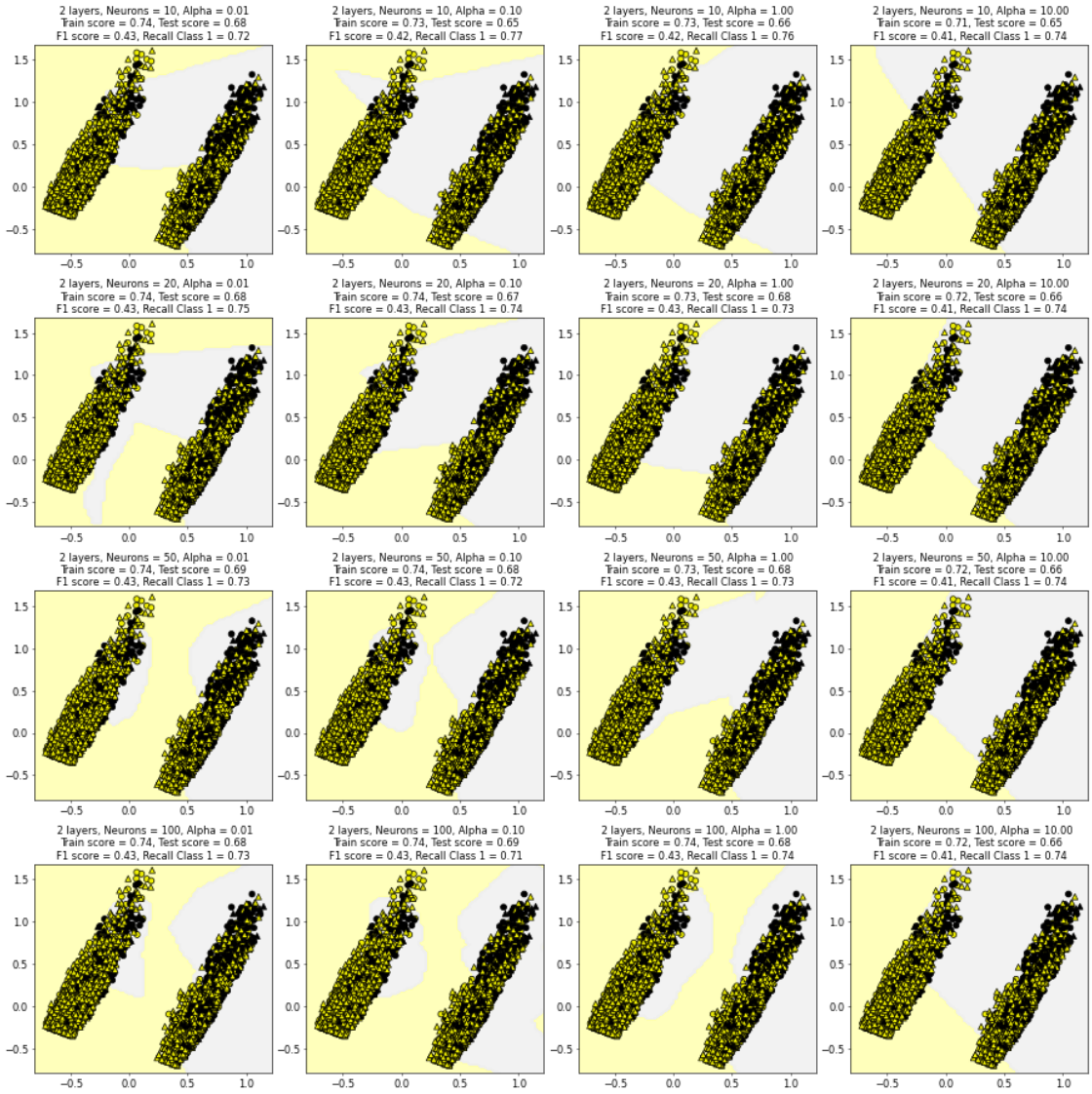
Normalized Data: NN classifier, No Regularization, 2 layers, 100/100 units
Train score = 0.74, Test score = 0.67
F1 score = 0.42, Recall Class 1 = 0.73



Notes: Best models look like two layers with either 50 units or 100 units. We need to apply regularization to see if this trend continues

1.7.2 MODEL 3: 2 Layers with Default Activation (Relu) with Different Regularization Levels

[48]:
```
fig, subaxes = plt.subplots(4, 4, figsize=(18, 18), dpi=50)

for this_unit, this_axis in zip([10, 20,50,100], subaxes):
    for this_alpha, subplot in zip([0.01, 0.1, 1, 10], this_axis):

        title = ' 2 layers, Neurons = {:.0f}, Alpha = {:.2f}'.format(this_unit,␣
↪this_alpha)
        nnclf = MLPClassifier(hidden_layer_sizes = [this_unit,this_unit], alpha␣
↪= this_alpha, solver='lbfgs', random_state=42).fit(X_train,y_train)
        plot_class_regions_for_classifier_subplot(nnclf, X_train,␣
↪y_train,X_test, y_test, title, subplot)


        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



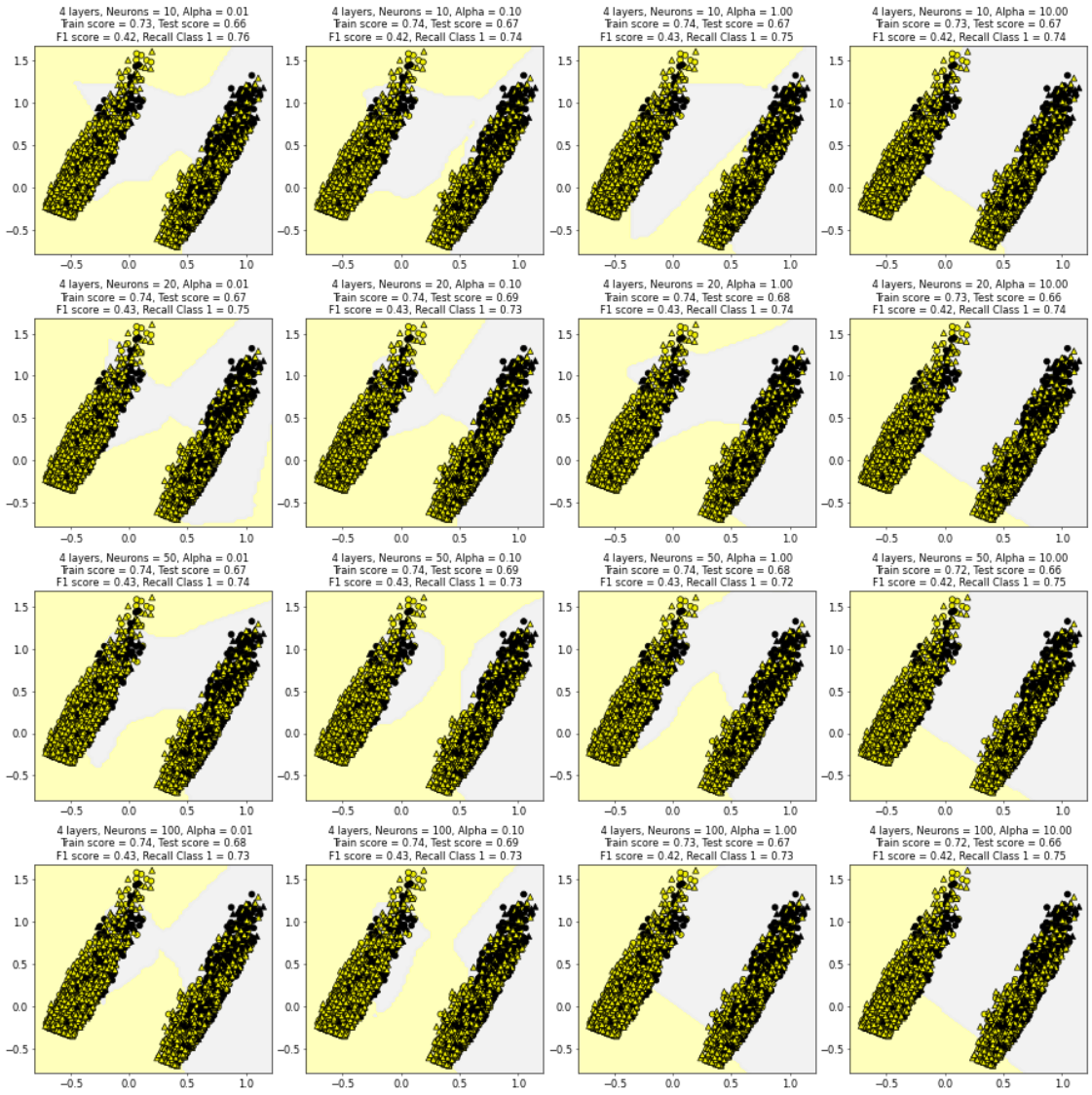
Note: Multiple models have very similar performance. WE are displaying the F1 score and Recall for class 1 to further differentiate. But there all the models here are very comparable so the least complex model would be best.

1.7.3 MODEL 4: 4 , 5 and 6 Layers with Default Activation (Relu) with Different Regularization Levels

- Do adding more layers help?

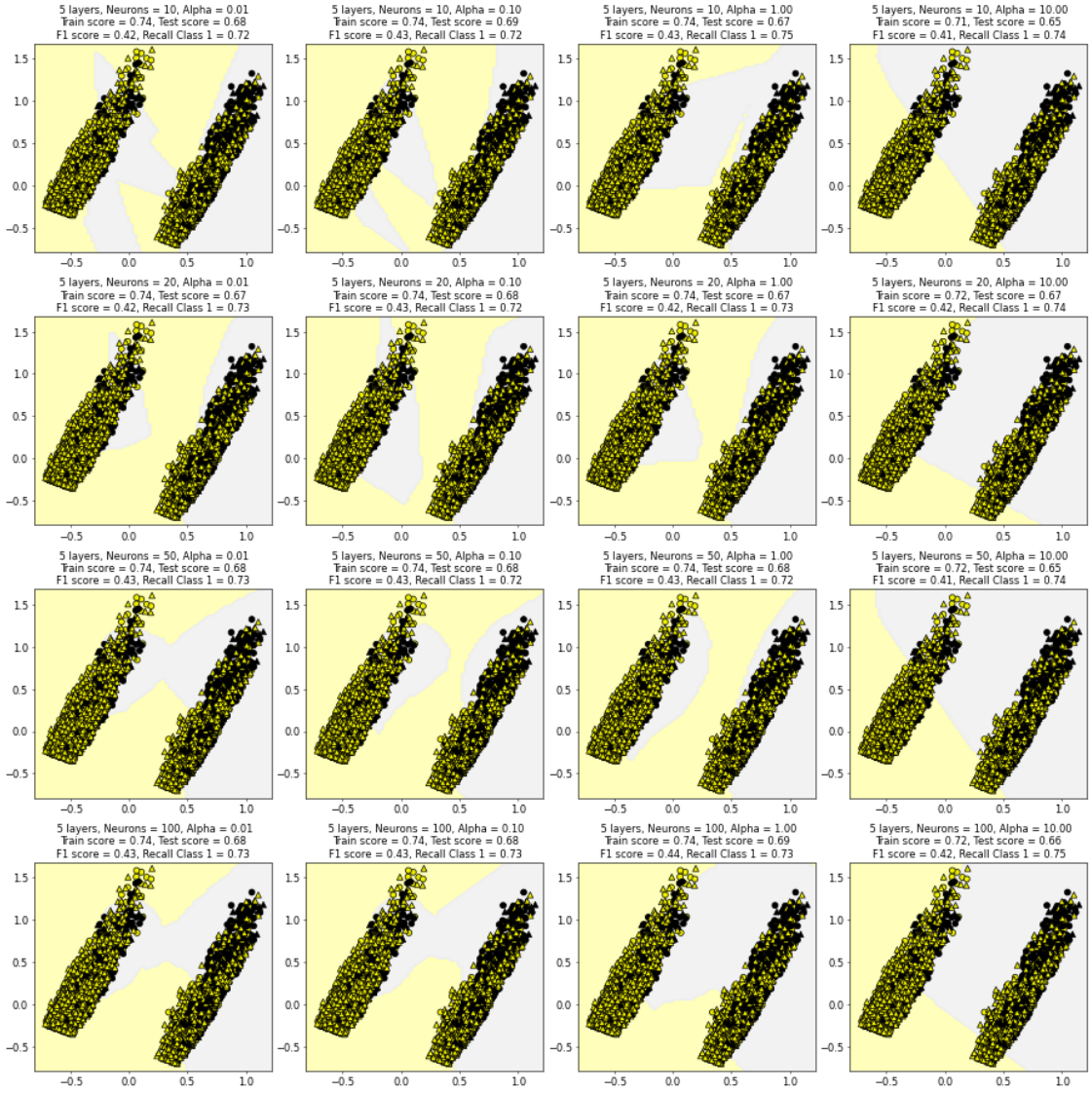
[49]:
```
fig, subaxes = plt.subplots(4, 4, figsize=(18, 18), dpi=50)

for this_unit, this_axis in zip([10, 20,50,100], subaxes):
    for this_alpha, subplot in zip([0.01, 0.1, 1, 10], this_axis):
```

```
        title = ' 4 layers, Neurons = {:.0f}, Alpha = {:.2f}'.format(this_unit,␣
↪this_alpha)
        nnclf = MLPClassifier(hidden_layer_sizes =␣
↪[this_unit,this_unit,this_unit,this_unit], alpha = this_alpha,␣
↪solver='lbfgs', random_state=42).fit(X_train,y_train)
        plot_class_regions_for_classifier_subplot(nnclf, X_train,␣
↪y_train,X_test, y_test, title, subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```

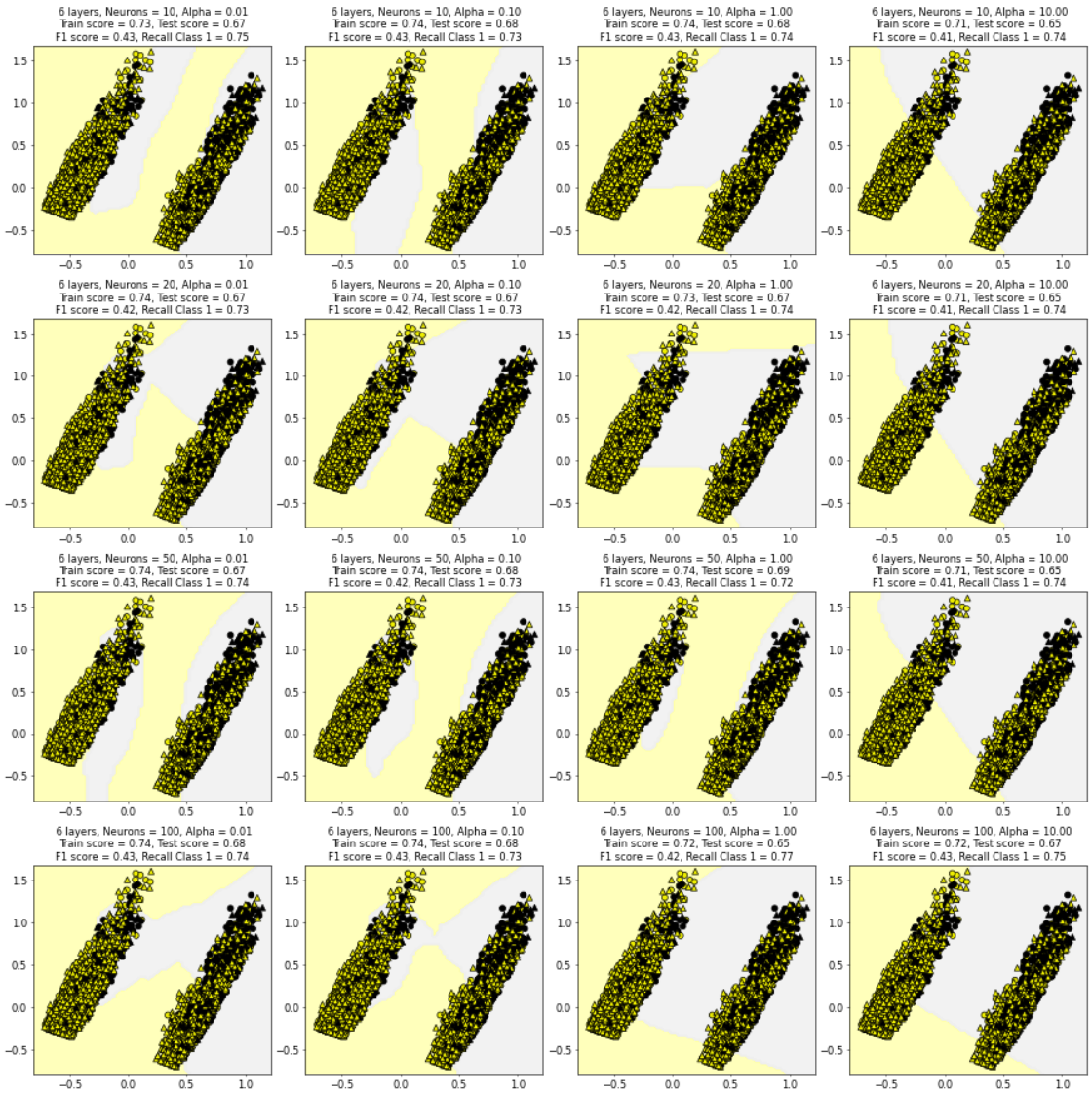


[50]:
```
fig, subaxes = plt.subplots(4, 4, figsize=(18, 18), dpi=50)
```

```
for this_unit, this_axis in zip([10, 20,50,100], subaxes):
    for this_alpha, subplot in zip([0.01, 0.1, 1, 10], this_axis):

        title = ' 5 layers, Neurons = {:.0f}, Alpha = {:.2f}'.format(this_unit,␣
↪this_alpha)
        nnclf = MLPClassifier(hidden_layer_sizes =␣
↪[this_unit,this_unit,this_unit,this_unit,this_unit], alpha = this_alpha,␣
↪solver='lbfgs', random_state=42).fit(X_train,y_train)
        plot_class_regions_for_classifier_subplot(nnclf, X_train,␣
↪y_train,X_test, y_test, title, subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



[51]:
```
fig, subaxes = plt.subplots(4, 4, figsize=(18, 18), dpi=50)

for this_unit, this_axis in zip([10, 20,50,100], subaxes):
    for this_alpha, subplot in zip([0.01, 0.1, 1, 10], this_axis):

        title = ' 6 layers, Neurons = {:.0f}, Alpha = {:.2f}'.format(this_unit,␣
↪this_alpha)
        nnclf = MLPClassifier(hidden_layer_sizes =␣
↪[this_unit,this_unit,this_unit,this_unit,this_unit,this_unit], alpha =␣
↪this_alpha, solver='lbfgs', random_state=42).fit(X_train,y_train)
        plot_class_regions_for_classifier_subplot(nnclf, X_train,␣
↪y_train,X_test, y_test, title, subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```

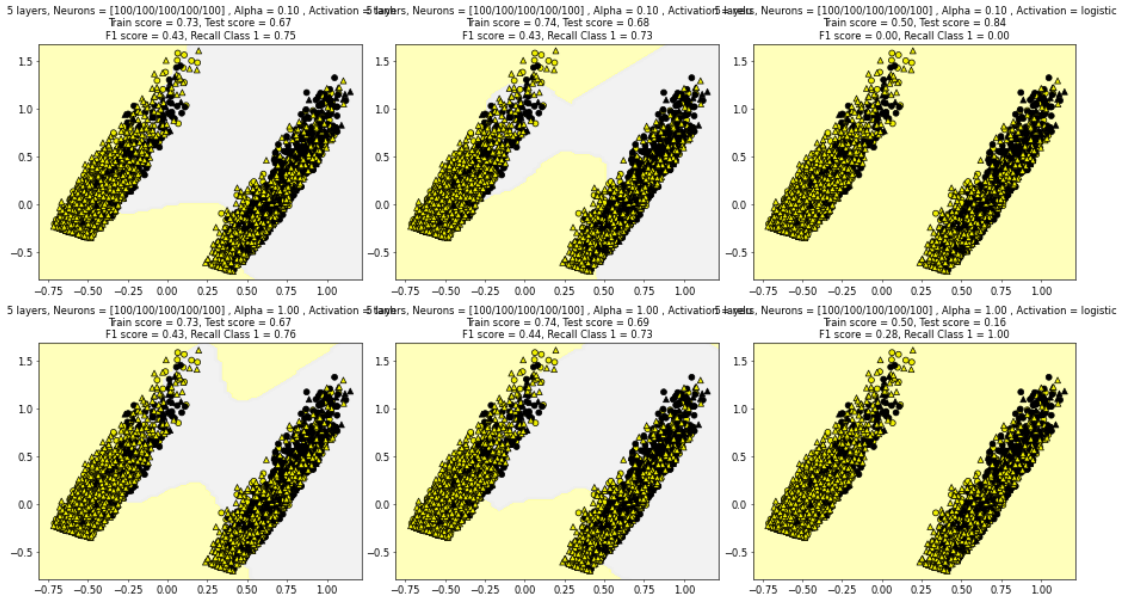


Note: Yes adding some more layers is helps upto a point. Best Model appears to be 5 layers of 100 neurons each with Alpha of 1. However increasing the layers to 6 layer is worse than 5. Therefore, we can see that the test scores are falling most likely due to overfit. The best model is at 5 layer of 100 neurons with alpha = 1 . The train score = 0.74 and test score = 0.69. F1 =0.44

1.7.4 MODEL 5: Changing Activation Function with Different Regularization Levels

- Are there better activation functions?
- We will use a 5 layer NN (100/100/100/100/100)
- alpha = 0.1 1

[52]:
```
fig, subaxes = plt.subplots(2, 3, figsize=(18, 10), dpi=50)

for this_alpha, this_axis in zip([0.1,1], subaxes):
    for this_activation, subplot in zip(['tanh', 'relu', 'logistic'],␣
 ↪this_axis):

        title = ' 5 layers, Neurons = [100/100/100/100/100] , Alpha = {:.2f} ,␣
 ↪Activation = {}'.format(this_alpha,this_activation)
        nnclf = MLPClassifier(hidden_layer_sizes = [100,100,100,100,100], alpha␣
 ↪= this_alpha, solver='lbfgs', random_state=42, activation = this_activation).
 ↪fit(X_train,y_train)
        plot_class_regions_for_classifier_subplot(nnclf, X_train,␣
 ↪y_train,X_test, y_test, title, subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



Note: Looking at all the activation functions the relu activation function sems to be performing

the best. The best model is :5 layers of 100 neurons, activation function = relu, alpha =1

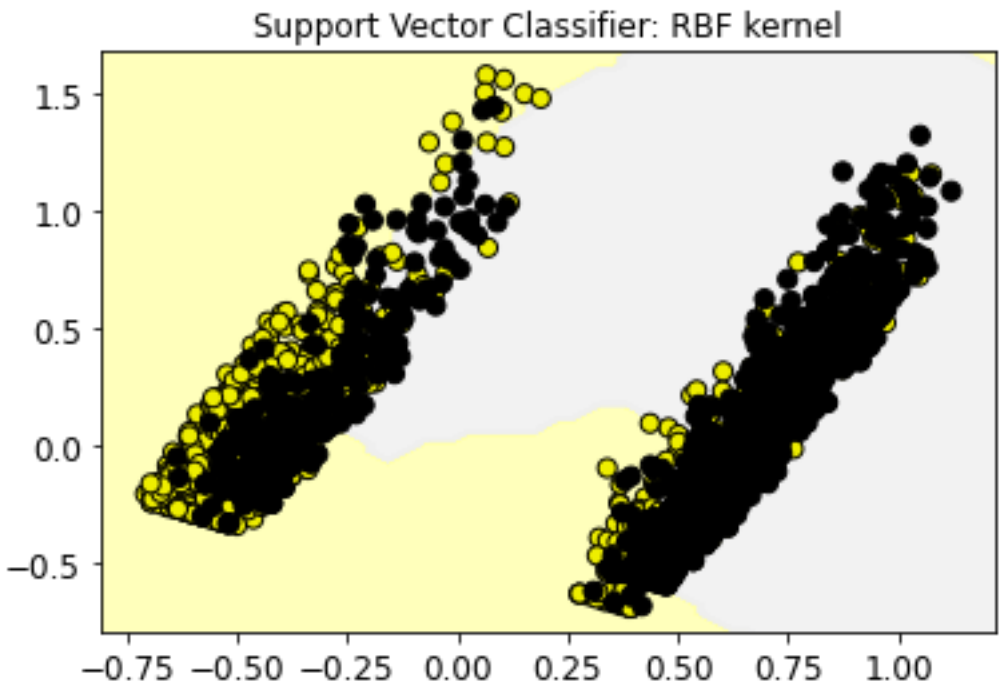
1.7.5 MODEL 6: Best Model - 5 layers of 100 neurons, activation function = relu, alpha =1

- Note we continue to use normalized, dimension reduced feature . We have takne a small sample of the total data as Neural network is very computation intensive
- Vizualize the Model
- Metrics: Score , Confusion Matrix, Classification Report
- ROC and AUC curve

[53]:
```
nn_best = MLPClassifier(hidden_layer_sizes = [100,100,100,100,100], alpha = 1,␣
↪solver='lbfgs', random_state=42, activation = 'relu').fit(X_train,y_train)

#clf_best = SVC(kernel='rbf', max_iter=10000, gamma=5, C=15, probability =True␣
↪, random_state=42).fit(X_train, y_train)


y_pred = nn_best.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print('Neural Network results\n', result_metrics)
```

```
Neural Network results
              precision    recall  f1-score   support

           0       0.93      0.68      0.79      2090
           1       0.31      0.73      0.44       410

    accuracy                           0.69      2500
   macro avg       0.62      0.71      0.61      2500
weighted avg       0.83      0.69      0.73      2500
```

[54]:
```
# Plot the classifier
plot_class_regions_for_classifier(nn_best.fit(X_train, y_train),
                                  X_train, y_train, None, None,
                                  'Support Vector Classifier: RBF kernel')
```



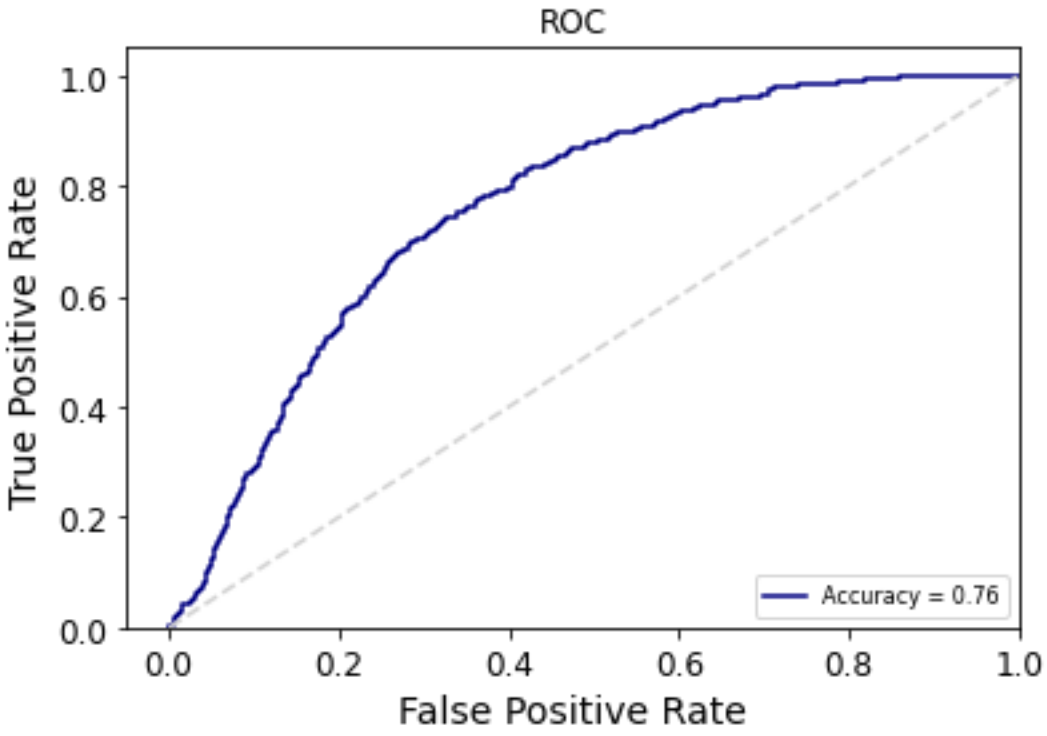
[55]:
```
#Score
print(" Score for the SVM Model = ",nn_best.score(X_test, y_test))

# Confusion Matrix
confusion_mat = confusion_matrix(y_test, y_pred)
print('\n Confusion Matrix: \n')
print(confusion_mat)

# Print classification report
target_names = ['Class 0', 'Class 1']
result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print('\n Classification Report: \n')
print(result_metrics)
```

```
 Score for the SVM Model =  0.6892

 Confusion Matrix:

[[1423  667]
 [ 110  300]]

 Classification Report:

              precision    recall  f1-score   support
```

```
     Class 0       0.93      0.68      0.79      2090
     Class 1       0.31      0.73      0.44       410

    accuracy                           0.69      2500
   macro avg       0.62      0.71      0.61      2500
weighted avg       0.83      0.69      0.73      2500
```

[56]:
```
from sklearn.metrics import roc_curve, auc
y_score = nn_best.predict_proba(X_test)

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_score[:,1])

roc_auc = auc(false_positive_rate, true_positive_rate)
print('Accuracy (AUC) = ', roc_auc)

count = 1

# Get different color each graph line
colorSet = ['navy', 'greenyellow', 'deepskyblue', 'darkviolet', 'crimson',
            'darkslategray', 'indigo', 'brown', 'orange', 'palevioletred',␣
 ↪'mediumseagreen',
            'k', 'darkgoldenrod', 'g', 'midnightblue', 'c', 'y', 'r', 'b', 'm',␣
 ↪'lawngreen'
            'mediumturquoise', 'lime', 'teal', 'drive', 'sienna', 'sandybrown']
color = colorSet[count-1]

# Plotting
plt.title('ROC')
plt.plot(false_positive_rate, true_positive_rate, c=color, label=('Accuracy =␣
 ↪%0.2f'%roc_auc))
plt.legend(loc='lower right', prop={'size':8})
plt.plot([0,1],[0,1], color='lightgrey', linestyle='--')
plt.xlim([-0.05,1.0])
plt.ylim([0.0,1.05])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
Accuracy (AUC) =  0.7630073520830902
```



2 Conclusion

- After selecting the most important features using the decision tree classifier (8 features out of 21). We carried out PCA and reduced the dimesions to two dimensions, This is similar to the analysis carried out in the Naive Bayes classifier sheet done prior.

- Neural Networks are computationally expensive and we used a sample of 10,000 from the whole dataset (0.24 million) to train and test the model.
- The best neural network model has 5 layers with 100 neurons in each layer. Regularization was applied with alpha = 1. The RELU activation function was best among the three activation functions we tried
- Our Dataset is significantly imbalanced with positive class "Diabetes = Yes" is the minority. We are interested in catching positive cases. The classification reports show that Recall for class1 = 0.73 which is good but the precision is very low at 0.31. The model tends to generate many false positives.
- We found the result of the SVM classifier to be slightly better than the neural network models. This is detailed in the following sheet.

3 REFERENCES

https://www.codecademy.com/learn/machine-learning/modules/dspath-clustering/cheatsheet

https://mclguide.readthedocs.io/en/latest/sklearn/clusterdim.html

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://www.discoverbits.in/371/sklearn-attributeerror-predict_proba-available-probability

https://www.kaggle.com/residentmario/undersampling-and-oversampling-imbalanced-data

Material from Machine Learning Course, Seattle University

Material from Introduction to Data Science, Seattle University

4 –END–

4.1 NEXT NOTEBOOK -> project_part_4_Classification_SVM (Support Vector Machines)

project_part_4_Classification_SVM

March 10, 2022

1 NOTEBOOK 6: CLASSIFICATION - SVM

1.0.1 Team 3

- Anjali Sebastian
- Yesha Sharma
- Rupansh Phutela

1.0.2 What this Notebook does?

After Data selection, cleaning, pre-processing, EDA and Regression Analysis, Clustering, and Gaussian Naive Bayes Classification and Neural Networks usinng MLP, we will now look at how we can perform SVM classfication on our data. Our data has target varibale y = Diabetes (Yes or No) we will try to classify the data to see the performance of different classifiers. In this Notebook we are trying various **Support Vector Machine** Classification Models.

- Normalization of entire dataset due to varying ranges of different attributes
- Feature Importances - Identify Best Features
- Use Principle Component Analysis to reduce dimensionality of the best selected features
- Multiple SVM models (Linear and RBF). Find good C and Gamma Parameter
- Analysis of the Best SVM Model in terms of metrics, confusion matrix, classification report and ROC Curve
- Conclusion
- References

1.1 1. Import Packages and Setup

[1]:
```
# you need  Python 3.5
import sys
assert sys.version_info >= (3, 5)
```

[2]:
```
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
```

[3]:
```
import os
import pandas as pd
import numpy as np
import seaborn as sns
```

```
import time
import warnings
warnings.filterwarnings("ignore")
################################################
```

[4]:
```
# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)


# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "clustering_kmeans"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

# method to save figures
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.2 2. Utility Functions

[5]:
```
import matplotlib.patches as mpatches
from matplotlib.colors import ListedColormap, BoundaryNorm

def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_labelled_scatter(X, y, class_labels):
    num_labels = len(class_labels)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    marker_array = ['o', '^', '*']
```

```
    color_array = ['#FFFF00','#00AAFF','#000000','#FF00AA','#2ca02c',␣
↪'#d62728','#9467bd','#8c564b','#e377c2']
    cmap_bold = ListedColormap(color_array)
    bnorm = BoundaryNorm(np.arange(0, num_labels + 1, 1), ncolors=num_labels)
    plt.figure()

    plt.scatter(X[:, 0], X[:, 1], s=65, c=y, cmap=cmap_bold, norm = bnorm,␣
↪alpha = 0.40, edgecolor='black', lw = 1)

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)

    h = []
    for c in range(0, num_labels):
        h.append(mpatches.Patch(color=color_array[c], label=class_labels[c]))
    plt.legend(handles=h)
    plt.show()
```

[6]:
```
# a function to plot a bar graph of important features
def plot_feature_importances(clf, feature_names):
    c_features = len(feature_names)
    #plt.figure(figsize=(15,4))
    plt.figure(figsize=(8,8))
    plt.barh(range(c_features), clf.feature_importances_)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature name")
    plt.yticks(np.arange(c_features), feature_names)
```

[7]:
```
def plot_class_regions_for_classifier_subplot(clf, X, y, X_test, y_test, title,␣
↪subplot, target_names = None, plot_decision_regions = True):

    numClasses = np.amax(y) + 1
    color_list_light = ['#FFFFAA', '#EFEFEF', '#AAFFAA', '#AAAAFF']
    color_list_bold = ['#EEEE00', '#000000', '#00CC00', '#0000CC']
    cmap_light = ListedColormap(color_list_light[0:numClasses])
    cmap_bold  = ListedColormap(color_list_bold[0:numClasses])

    h = 0.03
    k = 0.5
    x_plot_adjust = 0.1
    y_plot_adjust = 0.1
    plot_symbol_size = 50

    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()
```

```
    x2, y2 = np.meshgrid(np.arange(x_min-k, x_max+k, h), np.arange(y_min-k,␣
↪y_max+k, h))

    P = clf.predict(np.c_[x2.ravel(), y2.ravel()])
    P = P.reshape(x2.shape)

    if plot_decision_regions:
        subplot.contourf(x2, y2, P, cmap=cmap_light, alpha = 0.8)

    subplot.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=plot_symbol_size,␣
↪edgecolor = 'black')
    subplot.set_xlim(x_min - x_plot_adjust, x_max + x_plot_adjust)
    subplot.set_ylim(y_min - y_plot_adjust, y_max + y_plot_adjust)

    if (X_test is not None):
        subplot.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
↪s=plot_symbol_size, marker='^', edgecolor = 'black')
        train_score = clf.score(X, y)
        test_score  = clf.score(X_test, y_test)
        title = title + "\nTrain score = {:.2f}, Test score = {:.2f}".
↪format(train_score, test_score)

    subplot.set_title(title)

    if (target_names is not None):
        legend_handles = []
        for i in range(0, len(target_names)):
            patch = mpatches.Patch(color=color_list_bold[i],␣
↪label=target_names[i])
            legend_handles.append(patch)
        subplot.legend(loc=0, handles=legend_handles)


def plot_class_regions_for_classifier(clf, X, y, X_test=None, y_test=None,␣
↪title=None, target_names = None, plot_decision_regions = True):

    numClasses = np.amax(y) + 1
    color_list_light = ['#FFFFAA', '#EFEFEF', '#AAFFAA', '#AAAAFF']
    color_list_bold = ['#EEEE00', '#000000', '#00CC00', '#0000CC']
    cmap_light = ListedColormap(color_list_light[0:numClasses])
    cmap_bold  = ListedColormap(color_list_bold[0:numClasses])

    h = 0.03
    k = 0.5
    x_plot_adjust = 0.1
    y_plot_adjust = 0.1
```

```
    plot_symbol_size = 50

    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()
    x2, y2 = np.meshgrid(np.arange(x_min-k, x_max+k, h), np.arange(y_min-k,␣
↪y_max+k, h))

    P = clf.predict(np.c_[x2.ravel(), y2.ravel()])
    P = P.reshape(x2.shape)
    plt.figure()
    if plot_decision_regions:
        plt.contourf(x2, y2, P, cmap=cmap_light, alpha = 0.8)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=plot_symbol_size,␣
↪edgecolor = 'black')
    plt.xlim(x_min - x_plot_adjust, x_max + x_plot_adjust)
    plt.ylim(y_min - y_plot_adjust, y_max + y_plot_adjust)

    if (X_test is not None):
        plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
↪s=plot_symbol_size, marker='^', edgecolor = 'black')
        train_score = clf.score(X, y)
        test_score  = clf.score(X_test, y_test)
        title = title + "\nTrain score = {:.2f}, Test score = {:.2f}".
↪format(train_score, test_score)

    if (target_names is not None):
        legend_handles = []
        for i in range(0, len(target_names)):
            patch = mpatches.Patch(color=color_list_bold[i],␣
↪label=target_names[i])
            legend_handles.append(patch)
        plt.legend(loc=0, handles=legend_handles)

    if (title is not None):
        plt.title(title)
    plt.show()
```

[8]:
```
# Show confusion matrix
def plot_confusion_matrix(confusion_mat, cln):
    plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.gray)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(cln)
    plt.xticks(tick_marks, tick_marks)
```

```
    plt.yticks(tick_marks, tick_marks)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

1.3 3. Read Data and Display

[9]:
```
diabetes = pd.read_csv('./diabetes.csv')
```

[10]:
```
diabetes.head()
```

[10]:

	Unnamed: 0	Diabetes	BMI	State	HighBP	HighChol	CholCheck	\
0	0	0.0	28.17	AL	1.0	1.0	1.0	
1	1	0.0	18.54	AL	0.0	0.0	1.0	
2	2	1.0	31.62	AL	1.0	0.0	1.0	
3	6	1.0	32.98	AL	0.0	0.0	1.0	
4	9	1.0	16.65	AL	0.0	1.0	1.0	

	FruitConsume	VegetableConsume	Smoker	…	NoDoctorDueToCost	\
0	1.0	1.0	1.0	…	0.0	
1	1.0	1.0	0.0	…	0.0	
2	1.0	1.0	0.0	…	0.0	
3	1.0	1.0	1.0	…	0.0	
4	0.0	0.0	1.0	…	0.0	

	PhysicalActivity	GeneralHealth	PhysicalHealth	MentalHealth	\
0	0.0	3.0	15.0	0.0	
1	1.0	2.0	10.0	0.0	
2	1.0	3.0	0.0	30.0	
3	1.0	4.0	30.0	0.0	
4	0.0	1.0	20.0	0.0	

	DifficultyWalking	Gender	Age	Education	Income
0	1.0	0.0	13.0	3.0	3.0
1	0.0	0.0	11.0	5.0	5.0
2	1.0	0.0	10.0	6.0	7.0
3	1.0	1.0	11.0	6.0	7.0
4	1.0	0.0	11.0	2.0	3.0

[5 rows x 24 columns]

[11]:
```
#set datatypes of columns to boolean or categorical as appropriate
make_bool_int = ['Diabetes','HighBP','HighChol','CholCheck',\
                 ␣
 ↪'FruitConsume','VegetableConsume','Smoker','HeavyDrinker','Stroke','HeartDisease',\
                 ␣
 ↪'Healthcare','NoDoctorDueToCost','PhysicalActivity','DifficultyWalking','Gender']
```

```
make_categorical_int = ['GeneralHealth','Age','Education','Income']
```

[12]:
```
#drop the extra index column in datafram
diabetes=diabetes.drop(['Unnamed: 0'], axis=1)

#drop the state column in dataframe since it will not be used in the dataframe
diabetes=diabetes.drop(['State'], axis=1)
```

[13]:
```
diabetes.head()
```

[13]:

	Diabetes	BMI	HighBP	HighChol	CholCheck	FruitConsume	\
0	0.0	28.17	1.0	1.0	1.0	1.0	
1	0.0	18.54	0.0	0.0	1.0	1.0	
2	1.0	31.62	1.0	0.0	1.0	1.0	
3	1.0	32.98	0.0	0.0	1.0	1.0	
4	1.0	16.65	0.0	1.0	1.0	0.0	

	VegetableConsume	Smoker	HeavyDrinker	Stroke	…	NoDoctorDueToCost	\
0	1.0	1.0	0.0	0.0	…	0.0	
1	1.0	0.0	0.0	0.0	…	0.0	
2	1.0	0.0	0.0	0.0	…	0.0	
3	1.0	1.0	0.0	0.0	…	0.0	
4	0.0	1.0	0.0	0.0	…	0.0	

	PhysicalActivity	GeneralHealth	PhysicalHealth	MentalHealth	\
0	0.0	3.0	15.0	0.0	
1	1.0	2.0	10.0	0.0	
2	1.0	3.0	0.0	30.0	
3	1.0	4.0	30.0	0.0	
4	0.0	1.0	20.0	0.0	

	DifficultyWalking	Gender	Age	Education	Income
0	1.0	0.0	13.0	3.0	3.0
1	0.0	0.0	11.0	5.0	5.0
2	1.0	0.0	10.0	6.0	7.0
3	1.0	1.0	11.0	6.0	7.0
4	1.0	0.0	11.0	2.0	3.0

[5 rows x 22 columns]

[14]:
```
# deep copy before next stage
df = diabetes.copy(deep = True)
```

[15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 243317 entries, 0 to 243316
Data columns (total 22 columns):
```

```
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   Diabetes            243317 non-null  float64
 1   BMI                 243317 non-null  float64
 2   HighBP              243317 non-null  float64
 3   HighChol            243317 non-null  float64
 4   CholCheck           243317 non-null  float64
 5   FruitConsume        243317 non-null  float64
 6   VegetableConsume    243317 non-null  float64
 7   Smoker              243317 non-null  float64
 8   HeavyDrinker        243317 non-null  float64
 9   Stroke              243317 non-null  float64
 10  HeartDisease        243317 non-null  float64
 11  Healthcare          243317 non-null  float64
 12  NoDoctorDueToCost   243317 non-null  float64
 13  PhysicalActivity    243317 non-null  float64
 14  GeneralHealth       243317 non-null  float64
 15  PhysicalHealth      243317 non-null  float64
 16  MentalHealth        243317 non-null  float64
 17  DifficultyWalking   243317 non-null  float64
 18  Gender              243317 non-null  float64
 19  Age                 243317 non-null  float64
 20  Education           243317 non-null  float64
 21  Income              243317 non-null  float64
dtypes: float64(22)
memory usage: 40.8 MB
```

[16]:
```
df.shape
```

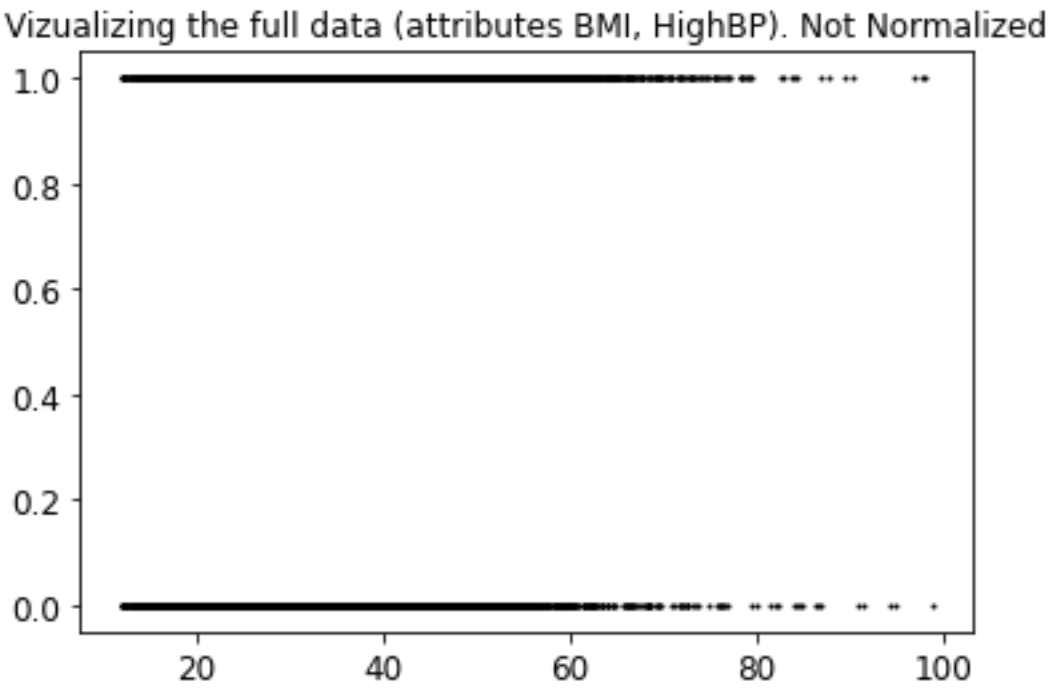
[16]:
```
(243317, 22)
```

1.4 4. Normalization and Simple Vizualization

[17]:
```
X_columns = ['BMI', 'HighBP', 'HighChol', 'CholCheck', 'FruitConsume',
       'VegetableConsume', 'Smoker', 'HeavyDrinker', 'Stroke', 'HeartDisease',
       'Healthcare', 'NoDoctorDueToCost', 'PhysicalActivity', 'GeneralHealth',
       'PhysicalHealth', 'MentalHealth', 'DifficultyWalking', 'Gender', 'Age',
       'Education', 'Income']
```

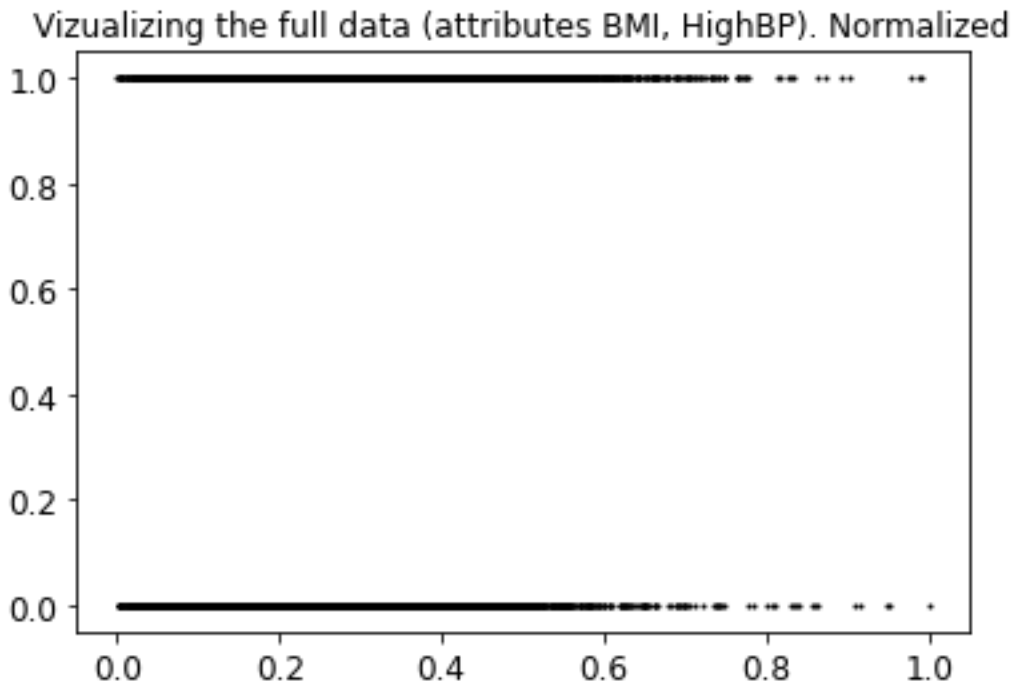
[18]:
```
# separating the target column y = Diabetes before classification
# for complete dataset
X_df = df[X_columns].values
y_df = df[['Diabetes']]
plot_data(X_df)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Not Normalized")
```

[18]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Not
Normalized')
```



[19]:
```
# Using minmax scaler for normalization
from sklearn.preprocessing import MinMaxScaler

# normalization full dataset
X_normalized = MinMaxScaler().fit(X_df).transform(X_df)
df_normalized = pd.DataFrame(X_normalized, columns=X_columns )
plot_data(X_normalized)
plt.title("Vizualizing the full data (attributes BMI, HighBP). Normalized")
```

[19]:
```
Text(0.5, 1.0, 'Vizualizing the full data (attributes BMI, HighBP). Normalized')
```



Note: The data pairs are as follows: - Full Data 1. X_df (pandas) with y_df(pandas) : not normalized full data set 2. X_normalized (numpy) with y_df(pandas) : normalized full X in numpy (easy for clustering) 3. df_normalized (pandas) with y_df(pandas) : normalized X in pandas format (easy for tracking feature names)

- For all our classification we will use only the normalized versions of the dataset.
- We will first pick the best features flowing which we will use PCA to reduce dataset to 2 features

1.5 5. Feature Importances - With Decision Tree Classifier

- We are using Decision Tree Classifier to find which features are more important to see which features are having the highestimpact on our target.
- We will only be using normalized data. Since it will put all features in similar range.

- We will be using the full dataset as is . We will also we using a balanced version of the dataset using undersampling technique to see if there is any change in the key features.

[20]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

[21]:
```
X = X_normalized
y_df['Diabetes']=y_df['Diabetes'].astype('int')
```

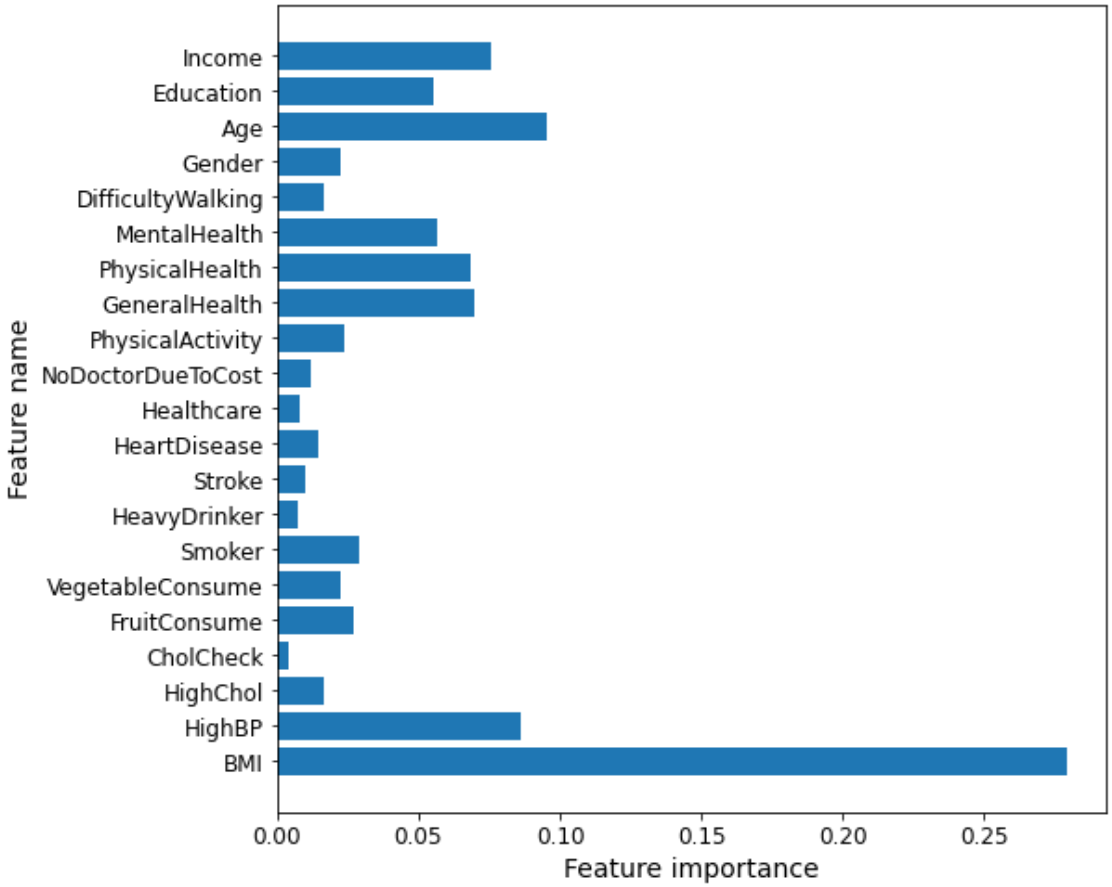
```
y = y_df.to_numpy()
```

[22]:
```
# A simple training (1 training)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0,␣
↪test_size=0.30)
```

Using Full Dataset As Is

[23]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))


# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.79
```




```
Feature importances: [0.27931751 0.08635461 0.01666336 0.00396084 0.02698384
0.02236237
 0.02883552 0.00744351 0.00987176 0.01420467 0.0082121  0.01204351
 0.02342647 0.06937859 0.06842486 0.05683353 0.01673057 0.02241037
 0.09529659 0.05551182 0.0757336 ]
```

[24]:
```
clf.score(X_test, y_test)
```

[24]:
```
0.7932900432900433
```

[25]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[25]:
```
array([[54468,  8045],
       [ 7044,  3439]], dtype=int64)
```

[26]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.89      0.87      0.88     62513
     Class 1       0.30      0.33      0.31     10483

    accuracy                           0.79     72996
   macro avg       0.59      0.60      0.60     72996
weighted avg       0.80      0.79      0.80     72996
```

Doing with a Balanced Dataset

- using random undersampler only on the training part

[27]:
```
# import RandomUndersampler
from imblearn.under_sampling import RandomUnderSampler
```

[28]:
```
X_train.shape
```

[28]:
```
(170321, 21)
```

[29]:
```
under = RandomUnderSampler(sampling_strategy='auto')
X_train, y_train = under.fit_resample(X_train, y_train)
```

[30]:
```
X_train.shape
```

[30]:
```
(49632, 21)
```

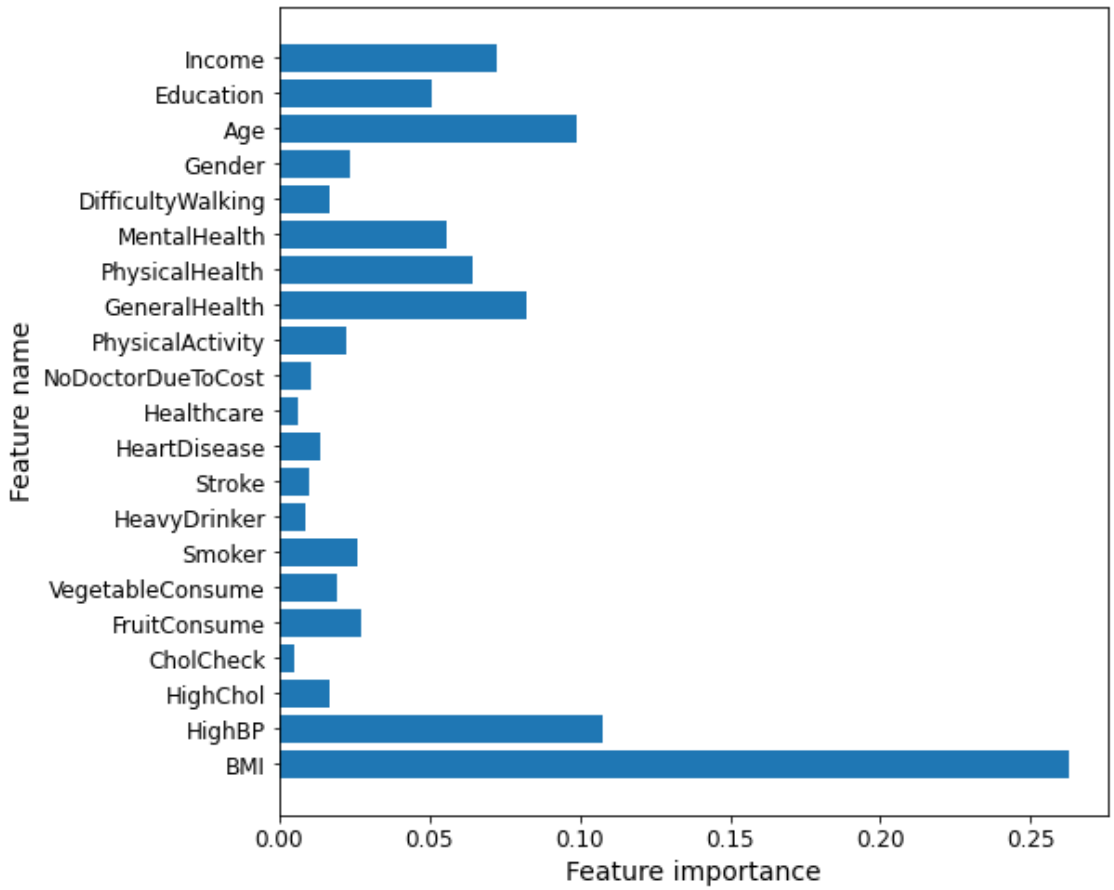
[31]:
```
unique, counts = np.unique(y_train, return_counts=True)
print ( np.asarray((unique, counts)).T)
```

```
[[    0 24816]
 [    1 24816]]
```

[32]:
```
clf = DecisionTreeClassifier(criterion='entropy').fit(X_train, y_train)

train_score = clf.score(X_train, y_train)
test_score = clf.score(X_test, y_test)

print('Accuracy of DT classifier on training set: {:.2f}'.format(train_score))
print('Accuracy of DT classifier on test set: {:.2f}'.format(test_score))
```

```
#plt.figure(figsize=(12,12), dpi=60)

# import features (call the function above)
plot_feature_importances(clf, df_normalized.columns)

plt.show()

print('Feature importances: {}'.format(clf.feature_importances_))
```

```
Accuracy of DT classifier on training set: 1.00
Accuracy of DT classifier on test set: 0.66
```




```
Feature importances: [0.26291005 0.10754319 0.01688605 0.00499179 0.02712826
0.01928236
 0.02616177 0.00840229 0.00992798 0.01337396 0.0062639  0.01041017
 0.02246018 0.0820788  0.06436261 0.05540956 0.01649228 0.02356799
 0.09920803 0.05057206 0.0725667 ]
```

[33]:
```
clf.score(X_test, y_test)
```

[33]:
```
0.6602553564578881
```

[34]:
```
y_pred = clf.predict(X_test)

#  confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)
confusion_mat
```

[34]:
```
array([[41328, 21185],
       [ 3615,  6868]], dtype=int64)
```

[35]:
```
# Print classification report
target_names = ['Class 0', 'Class 1']

result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print(result_metrics)
```

```
              precision    recall  f1-score   support

     Class 0       0.92      0.66      0.77     62513
     Class 1       0.24      0.66      0.36     10483

    accuracy                           0.66     72996
   macro avg       0.58      0.66      0.56     72996
weighted avg       0.82      0.66      0.71     72996
```

Note: Looking at at the feature importance we can see that the bar plots for both the original dataset and the balanced data set are having similar patterns. We see that the following 8 features are very important - BMI, HighBP, General Health, Physical Health, Mental Health, Age , Education and Income.

[36]:
```
# Create a list of important features
important_features =␣
 ↪['BMI','HighBP','GeneralHealth','PhysicalHealth','MentalHealth','Age','Education','Income']
```

1.6 5. Principle Component Analysis

- Using the only the most important features dicovered from the decision tree model we reduce the dimensionality to 2 using Principal Component Analysis

[37]:
```
df_normalized.head()
```

[37]:

	BMI	HighBP	HighChol	CholCheck	FruitConsume	VegetableConsume	\
0	0.186505	1.0	1.0	1.0	1.0	1.0	
1	0.075433	0.0	0.0	1.0	1.0	1.0	
2	0.226298	1.0	0.0	1.0	1.0	1.0	
3	0.241984	0.0	0.0	1.0	1.0	1.0	

```
4  0.053633     0.0       1.0        1.0           0.0                0.0

   Smoker  HeavyDrinker  Stroke  HeartDisease  …  NoDoctorDueToCost  \
0     1.0           0.0     0.0           0.0  …                0.0
1     0.0           0.0     0.0           0.0  …                0.0
2     0.0           0.0     0.0           0.0  …                0.0
3     1.0           0.0     0.0           0.0  …                0.0
4     1.0           0.0     0.0           0.0  …                0.0

   PhysicalActivity  GeneralHealth  PhysicalHealth  MentalHealth  \
0               0.0           0.50        0.500000           0.0
1               1.0           0.25        0.333333           0.0
2               1.0           0.50        0.000000           1.0
3               1.0           0.75        1.000000           0.0
4               0.0           0.00        0.666667           0.0

   DifficultyWalking  Gender       Age  Education    Income
0                1.0     0.0  1.000000        0.4  0.285714
1                0.0     0.0  0.833333        0.8  0.571429
2                1.0     0.0  0.750000        1.0  0.857143
3                1.0     1.0  0.833333        1.0  0.857143
4                1.0     0.0  0.833333        0.2  0.285714

[5 rows x 21 columns]
```

[38]:
```
# Choose True if we are selecting only 8 top features for doing PCA else it␣
 ↪will take entire data set
select_features = True

if(select_features==True):
    df_best_features = df_normalized[important_features]
else:
    df_best_features = df_normalized
df_best_features.head()
```

[38]:
```
        BMI  HighBP  GeneralHealth  PhysicalHealth  MentalHealth       Age  \
0  0.186505     1.0           0.50        0.500000           0.0  1.000000
1  0.075433     0.0           0.25        0.333333           0.0  0.833333
2  0.226298     1.0           0.50        0.000000           1.0  0.750000
3  0.241984     0.0           0.75        1.000000           0.0  0.833333
4  0.053633     0.0           0.00        0.666667           0.0  0.833333

   Education    Income
0        0.4  0.285714
1        0.8  0.571429
2        1.0  0.857143
3        1.0  0.857143
```

```
4        0.2  0.285714
```

[39]:
```
# Dimesionality reduction to 2
from sklearn.decomposition import PCA

pca_model = PCA(n_components=2)
pca_model.fit(df_best_features) # fit the model
X_normalized_pca = pca_model.transform(df_best_features)
X_normalized_pca
```

[39]:
```
array([[ 0.8143773 ,  0.21944804],
       [-0.1775784 ,  0.39799483],
       [ 0.57868681, -0.00386865],
       …,
       [-0.3079835 ,  0.39221926],
       [-0.47663154,  0.34781812],
       [-0.51515748, -0.06059107]])
```

[40]:
```
# numpy
X_normalized_pca.shape
```

[40]:
```
(243317, 2)
```

[41]:
```
# panda it
df_X_normalized_pca = pd.DataFrame(X_normalized_pca,␣
 ↪columns=['Feature1','Feature2'] )
df_X_normalized_pca.head()
```

[41]:

	Feature1	Feature2
0	0.814377	0.219448
1	-0.177578	0.397995
2	0.578687	-0.003869
3	-0.225108	0.374260
4	0.051858	0.924369

Note: We have reduced our datasets dimensionality to 2 features which have just been named feature1 and feature2. Going ahead we will be using these two synthetic features to perform our classification.

1.7 6. Support Vector Machines

- SVM is a very slow algorithmn since it is very computationaly intensive so we will take only a part of the dataset to run the classification on (around 10,000)
- For Support Vector Machines we will try different X variables and classify the diabetics/non-diabetics.

[42]:
```
# attach back the labels before sampling
```

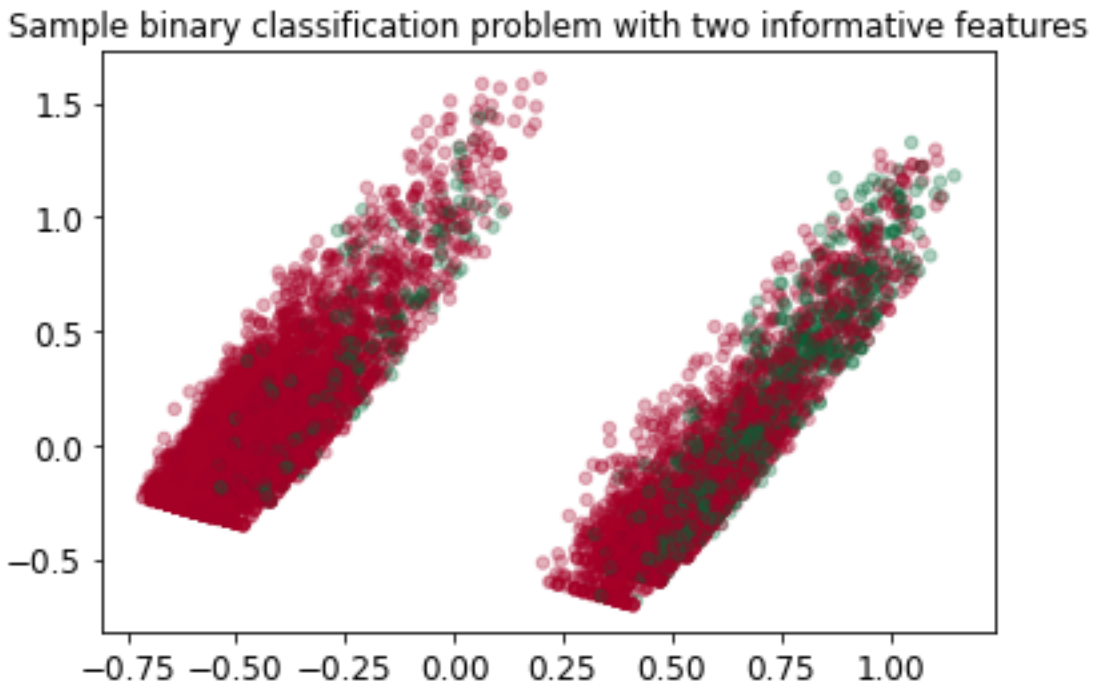
```
df_normalized_pca = pd.concat([df_X_normalized_pca.reset_index(drop=True), y_df.
 ↪reset_index(drop=True)], axis= 1)
df_normalized_pca.head()
```

[42]:

	Feature1	Feature2	Diabetes
0	0.814377	0.219448	0
1	-0.177578	0.397995	0
2	0.578687	-0.003869	1
3	-0.225108	0.374260	1
4	0.051858	0.924369	1

[43]:
```
# Selecting a random sample for the data set
#sampling a random number of values since sum classification of all 0.2 million␣
 ↪datapoints is too slow
# option 10000 , 50000 etc.
number_of_samples = 10000
sample_normalized_pca = df_normalized_pca.sample(number_of_samples,␣
 ↪random_state=42)
```

[44]:
```
# plotting the 2 attributes of PCA
plt.figure()
plt.title('Sample binary classification problem with two informative features')

#Plotting just sample points to not clutter the scatter plot
plt.scatter(sample_normalized_pca.iloc[:, 0], sample_normalized_pca.iloc[:, 1],␣
 ↪alpha = 0.3,cmap=plt.cm.RdYlGn,marker= 'o', s=20, c=sample_normalized_pca.
 ↪loc[:,'Diabetes'])
plt.show()
```



[45]:
```
# set up the Data
X = sample_normalized_pca.iloc[:,[0,1]].to_numpy()
y = sample_normalized_pca.iloc[:,[2]].to_numpy()
print(X.shape)
```

```
(10000, 2)
```

[46]:
```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state = 0)

# undersampling only on the Training sets
under = RandomUnderSampler(sampling_strategy='auto')
X_train, y_train = under.fit_resample(X_train, y_train)
print(X_train.shape)
```

```
(2194, 2)
```

1.7.1 MODEL 1: Linear Kernal

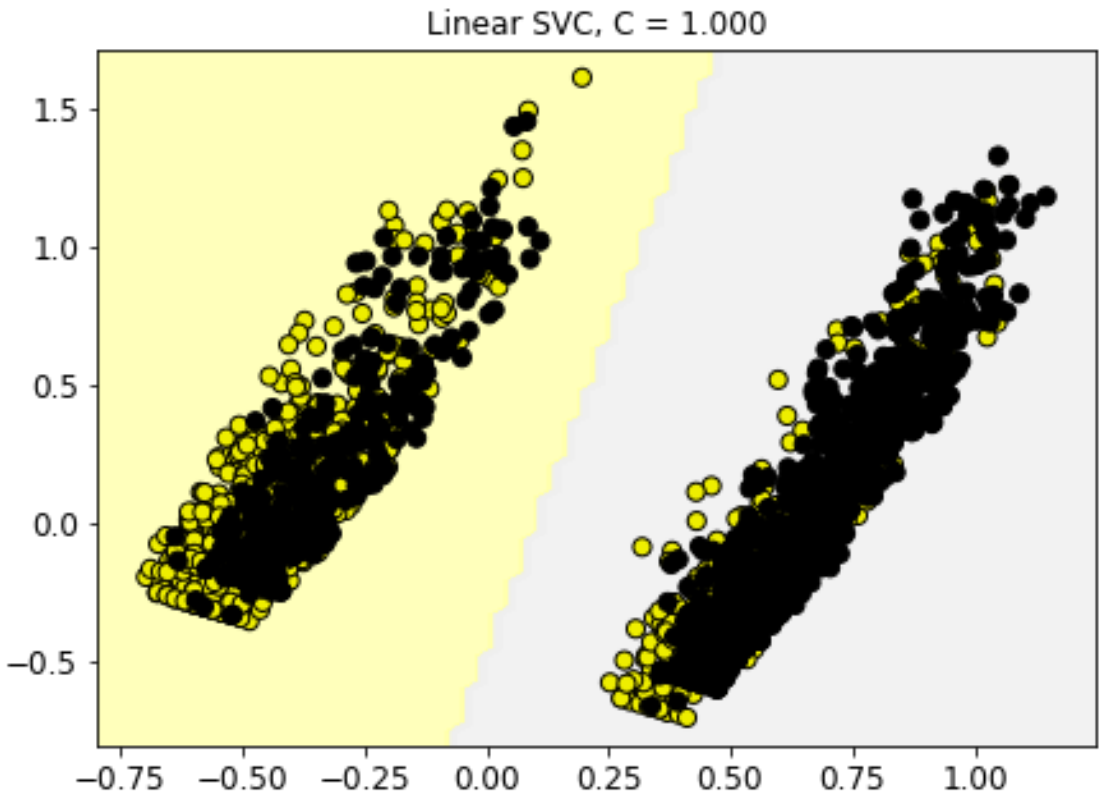
- We are using normalized data that was transformed by PCA to 2 features
- We will use under sampling technique since we are more interested in positive cases

- We only undersample the training sets because the model needs to perform with naturally imbalanced data (ie less positive diabetes cases) we leave the test sets as they are.

[47]:
```
clf1 = SVC(kernel = 'linear', C=1.0,random_state=42).fit(X_train, y_train)

y_pred = clf1.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print(result_metrics)
```

```
              precision    recall  f1-score   support

           0       0.94      0.64      0.76      2103
           1       0.29      0.77      0.42       397

    accuracy                           0.66      2500
   macro avg       0.61      0.71      0.59      2500
weighted avg       0.83      0.66      0.70      2500
```

[48]:
```
# plot the classifier
fig, subaxes = plt.subplots(1, 1, figsize=(7, 5))

title = 'Linear SVC, C = {:.3f}'.format(1.0)
plot_class_regions_for_classifier_subplot(clf1, X_train, y_train, None, None,␣
 ↪title, subaxes)
```



Note: As previously seen during EDA and Regression. Linear Models are not able to generate good results. We will try with the RBF kernal next.

1.7.2 MODEL 2: RBF Kernal

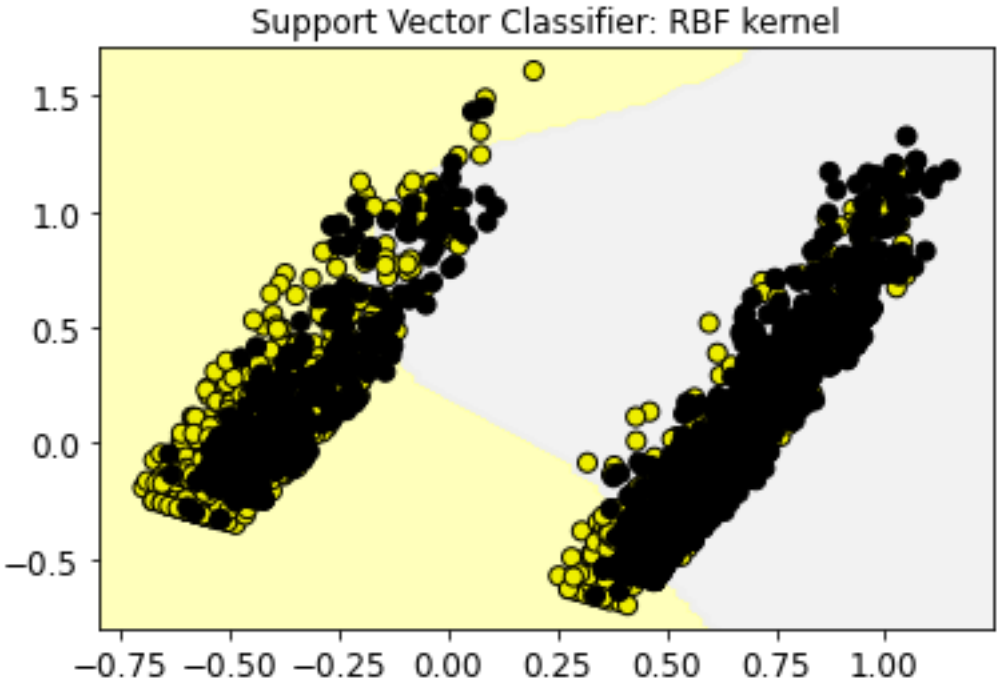
- We are using normalized data that was transformed by PCA to 2 features
- We will use under sampling technique since we are more interested in positive cases

[49]:
```
clf2 = SVC(kernel='rbf', max_iter=10000,random_state=42).fit(X_train, y_train)
y_pred = clf2.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print('RBF kernel (Gaussian) results\n', result_metrics)
```

```
RBF kernel (Gaussian) results
              precision    recall  f1-score   support

           0       0.94      0.71      0.81      2103
           1       0.33      0.74      0.45       397

    accuracy                           0.72      2500
   macro avg       0.63      0.73      0.63      2500
```

```
weighted avg       0.84      0.72      0.75      2500
```

[50]:
```
# The default SVC kernel is radial basis function (RBF)
plot_class_regions_for_classifier(clf2.fit(X_train, y_train),
                                  X_train, y_train, None, None,
                                  'Support Vector Classifier: RBF kernel')
```

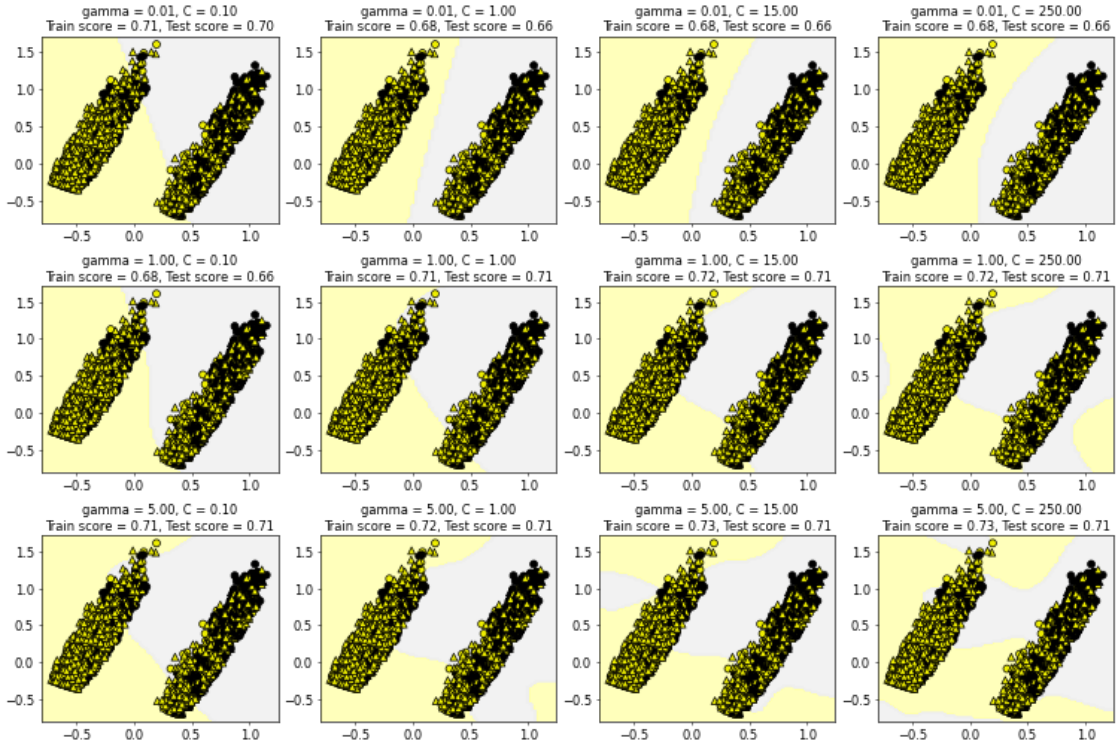


1.7.3 MODEL 3: RBF Kernal with varying C and Gamma Parameter

- Apply SVM RBF kernel using varying C and gamma parameter values.
- Use C= 0.1, 1, 15, 250. Use gamma= 0.01, 1, 5.
- Hence, 12 subplots, similar to the above example, should be drawn.
- Note we continue to use normalized, dimension reduced feature . We have take a small sample of the total data as SVM is very computation intensive

[51]:
```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

fig, subaxes = plt.subplots(3, 4, figsize=(15, 10), dpi=50)

for this_gamma, this_axis in zip([0.01, 1, 5], subaxes):

    for this_C, subplot in zip([0.1, 1, 15, 250], this_axis):
        title = 'gamma = {:.2f}, C = {:.2f}'.format(this_gamma, this_C)
```

```
        clf = SVC(kernel = 'rbf', gamma = this_gamma,
                  C = this_C,random_state=42).fit(X_train, y_train)
        plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                                  X_test, y_test, title,
                                                  subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



Note: Looking at various Gamma and C values the following look best - gamma =5 , c= 1 - gamma = 5 , c=15 Too high value of C is causes possible overfit and even with high train score the test score is not better.

[52]:
```
clf3 = SVC(kernel='rbf', max_iter=10000, gamma=5,C=1,random_state=42 ).
 ↪fit(X_train, y_train)
y_pred = clf3.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print('RBF kernel (Gaussian) results\n', result_metrics)
```

```
RBF kernel (Gaussian) results
              precision    recall  f1-score   support

           0       0.94      0.70      0.80      2103
           1       0.32      0.75      0.45       397
```

```
    accuracy                           0.71      2500
   macro avg       0.63      0.72      0.62      2500
weighted avg       0.84      0.71      0.74      2500
```

[53]:
```
clf4 = SVC(kernel='rbf', max_iter=10000, gamma=5,C=15,random_state=42  ).
 ↪fit(X_train, y_train)
y_pred = clf4.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print('RBF kernel (Gaussian) results\n', result_metrics)
```

```
RBF kernel (Gaussian) results
               precision    recall  f1-score   support

           0       0.94      0.70      0.80      2103
           1       0.32      0.76      0.45       397

    accuracy                           0.71      2500
   macro avg       0.63      0.73      0.63      2500
weighted avg       0.84      0.71      0.74      2500
```

Note: Gamma =5 and C=15 looks like the best option. The precision value class 1 across all models is poor. However recall for class 1 diabetes = 76 is ok. We will look at this model in more detail in the next section including plotting its ROC curve.

1.7.4 MODEL 4: BEST SVM MODEL - RBF Kernal with C =15 and Gamma = 5

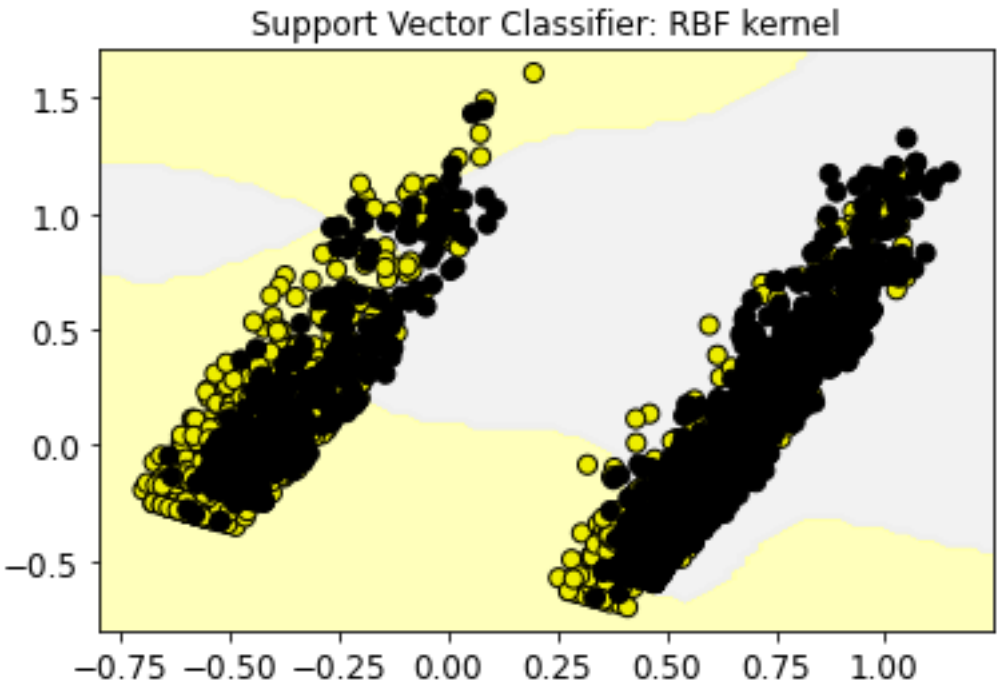
- Note we continue to use normalized, dimension reduced feature . We have take a small sample of the total data as SVM is very computation intensive
- Vizualize the Model
- Metrics: Score , Confusion Matrix, Classification Report
- ROC and AUC curve

[54]:
```
clf_best = SVC(kernel='rbf', max_iter=10000, gamma=5, C=15, probability =True ,␣
 ↪random_state=42).fit(X_train, y_train)
y_pred = clf_best.predict(X_test)

result_metrics = classification_report(y_test, y_pred)
print('RBF kernel (Gaussian) results\n', result_metrics)
```

```
RBF kernel (Gaussian) results
               precision    recall  f1-score   support

           0       0.94      0.70      0.80      2103
           1       0.32      0.76      0.45       397

    accuracy                           0.71      2500
```

```
   macro avg       0.63      0.73      0.63      2500
weighted avg       0.84      0.71      0.74      2500
```

[55]:
```
# Plot the classifier
plot_class_regions_for_classifier(clf_best.fit(X_train, y_train),
                                  X_train, y_train, None, None,
                                  'Support Vector Classifier: RBF kernel')
```



[56]:
```
#Score
print(" Score for the SVM Model = ",clf_best.score(X_test, y_test))

# Confusion Matrix
confusion_mat = confusion_matrix(y_test, y_pred)
print('\n Confusion Matrix: \n')
print(confusion_mat)

# Print classification report
target_names = ['Class 0', 'Class 1']
result_metrics = classification_report(y_test, y_pred,␣
 ↪target_names=target_names)
print('\n Classification Report: \n')
print(result_metrics)
```

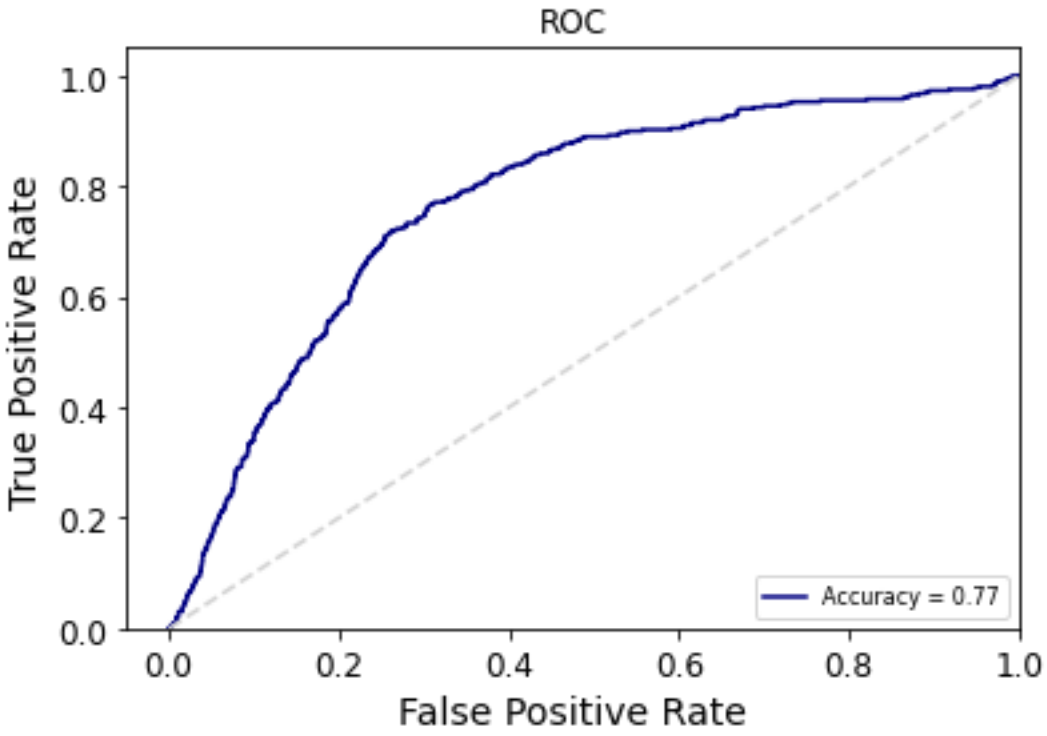
```
 Score for the SVM Model =  0.7072
```

```
 Confusion Matrix:

[[1468  635]
 [  97  300]]

 Classification Report:

              precision    recall  f1-score   support

     Class 0       0.94      0.70      0.80      2103
     Class 1       0.32      0.76      0.45       397

    accuracy                           0.71      2500
   macro avg       0.63      0.73      0.63      2500
weighted avg       0.84      0.71      0.74      2500
```

[57]:
```
from sklearn.metrics import roc_curve, auc
y_score = clf_best.predict_proba(X_test)

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_score[:,1])

roc_auc = auc(false_positive_rate, true_positive_rate)
print('Accuracy (AUC) = ', roc_auc)

count = 1

# Get different color each graph line
colorSet = ['navy', 'greenyellow', 'deepskyblue', 'darkviolet', 'crimson',
            'darkslategray', 'indigo', 'brown', 'orange', 'palevioletred',␣
 ↪'mediumseagreen',
            'k', 'darkgoldenrod', 'g', 'midnightblue', 'c', 'y', 'r', 'b', 'm',␣
 ↪'lawngreen'
            'mediumturquoise', 'lime', 'teal', 'drive', 'sienna', 'sandybrown']
color = colorSet[count-1]

# Plotting
plt.title('ROC')
plt.plot(false_positive_rate, true_positive_rate, c=color, label=('Accuracy =␣
 ↪%0.2f'%roc_auc))
plt.legend(loc='lower right', prop={'size':8})
plt.plot([0,1],[0,1], color='lightgrey', linestyle='--')
plt.xlim([-0.05,1.0])
plt.ylim([0.0,1.05])
plt.ylabel('True Positive Rate')
```

```
plt.xlabel('False Positive Rate')
plt.show()
```

```
Accuracy (AUC) =  0.7685051102479247
```



1.8 8. Relating SVM Classifier and Mini Batch KMeans Clustering

Among all three classifiers ie (Neural Networks, Naive Bayes and SVM) SVM classifier with RBF produced the best results. We will use the SVM classifier with RBF Kernal, C = gamma=5, C=15 and compare it with our KMeans Mini Batch Cluster Model to see if we can relate any results between the two.

[58]:
```
best_classifier = clf_best
```

[59]:
```
df_normalized_pca.head()
```

[59]:

	Feature1	Feature2	Diabetes
0	0.814377	0.219448	0
1	-0.177578	0.397995	0
2	0.578687	-0.003869	1
3	-0.225108	0.374260	1
4	0.051858	0.924369	1

[60]:
```
#Use the classifier to predict over the entire dataset
classifier_predict  = best_classifier.predict(df_normalized_pca.iloc[:,[0,1]].
 ↪to_numpy())
```

[61]:
```
classifier_predict.shape
```

[61]:
```
(243317,)
```

[62]:
```
classifier_predict
```

[62]:
```
array([1, 1, 1, …, 0, 0, 0])
```

[63]:
```
# concat the classifier labels
df_temp = pd.DataFrame(classifier_predict, columns=["Classifier_Labels"] )
df_compare = pd.concat([df_normalized_pca.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
df_compare.head()
```

[63]:

	Feature1	Feature2	Diabetes	Classifier_Labels
0	0.814377	0.219448	0	1
1	-0.177578	0.397995	0	1
2	0.578687	-0.003869	1	1
3	-0.225108	0.374260	1	1
4	0.051858	0.924369	1	1

[64]:
```
# Do clustering of the entire dataset using minibatch kmeans , batch size = 32,␣
 ↪n_cluster = 2
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=32,␣
 ↪max_iter=100).fit(X_normalized)
kmeans.fit(X_normalized)
```

[64]:
```
MiniBatchKMeans(batch_size=32, n_clusters=2, random_state=0)
```

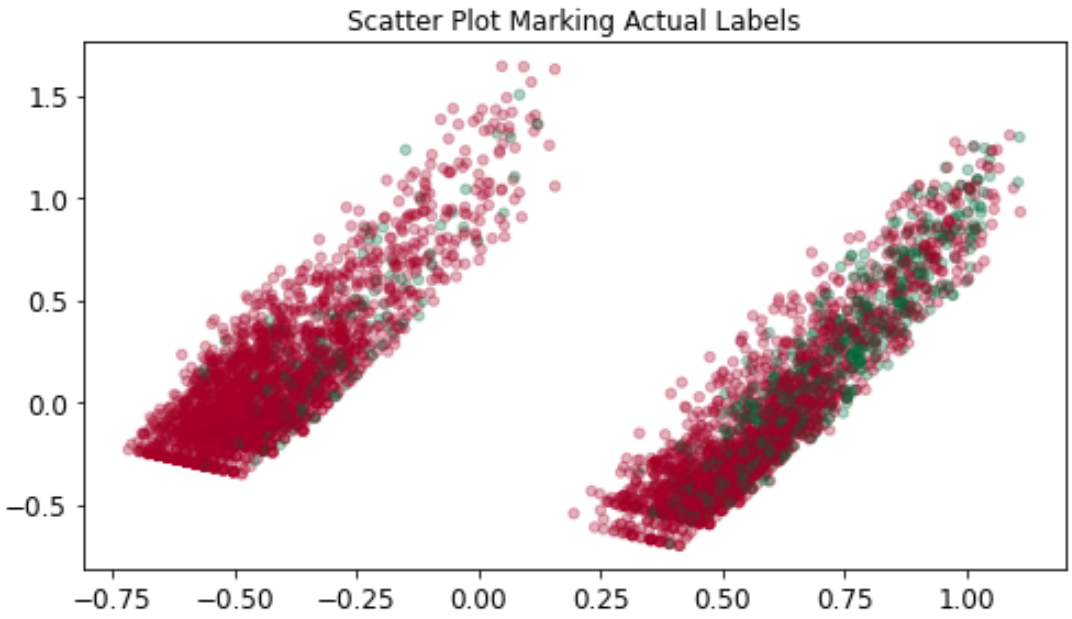
[65]:
```
# concat the cluster labels
df_temp = pd.DataFrame(kmeans.labels_, columns=["Cluster_Labels"] )
df_compare = pd.concat([df_compare.reset_index(drop=True), df_temp.
 ↪reset_index(drop=True)], axis= 1)
df_compare.head()
```

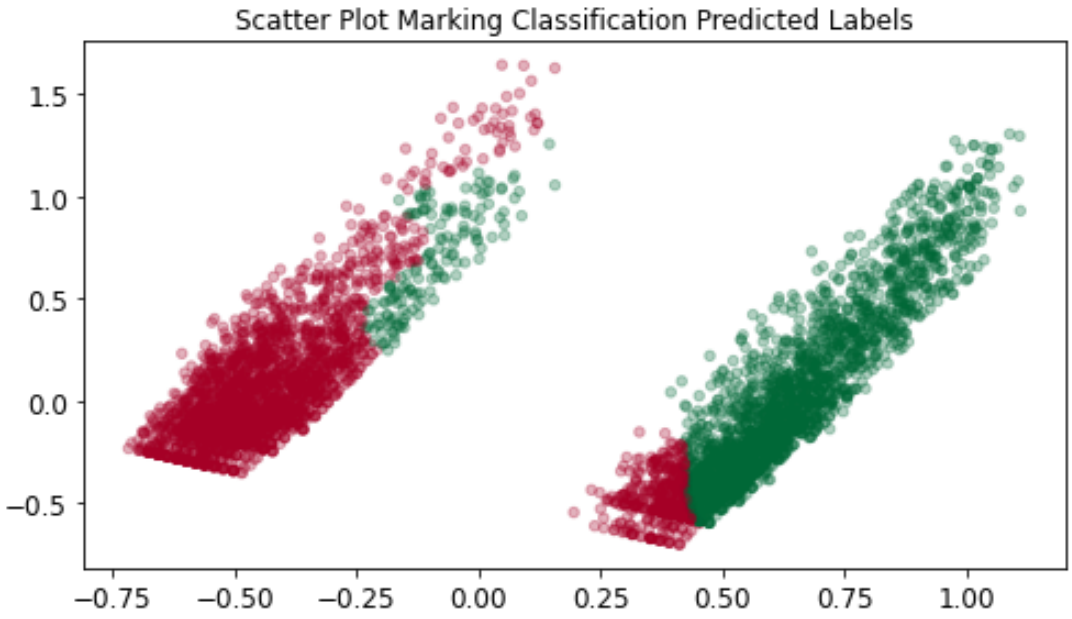
[65]:

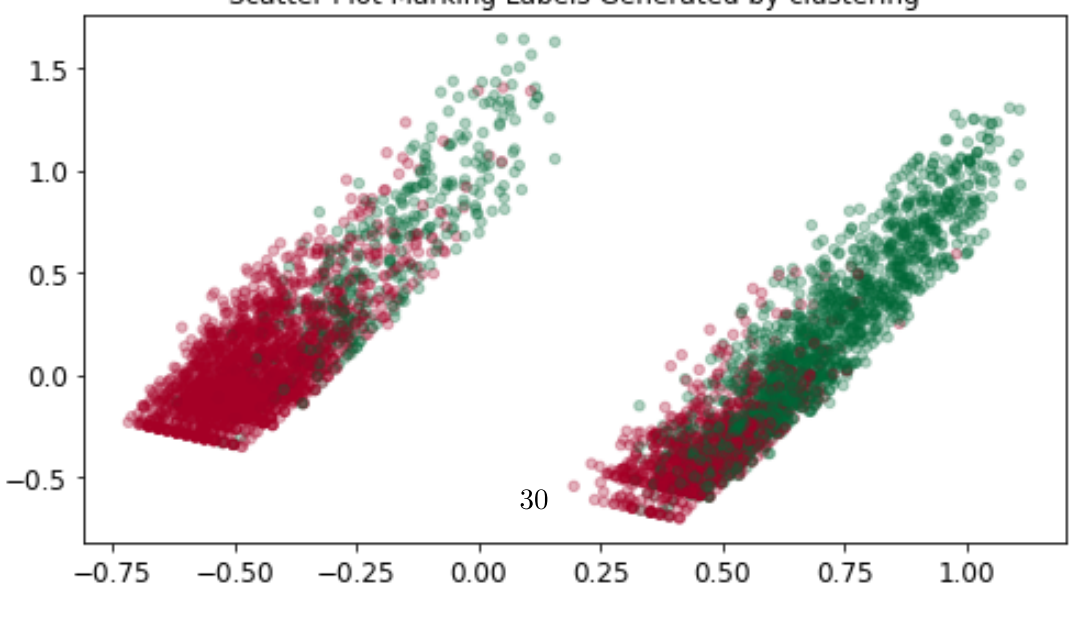
	Feature1	Feature2	Diabetes	Classifier_Labels	Cluster_Labels
0	0.814377	0.219448	0	1	1
1	-0.177578	0.397995	0	1	0
2	0.578687	-0.003869	1	1	1
3	-0.225108	0.374260	1	1	1
4	0.051858	0.924369	1	1	1

[66]:
```
fig, [ax1,ax2,ax3] = plt.subplots(3,1, figsize = (8,15))

ax1.scatter(df_compare.iloc[:5000, 0], df_compare.iloc[:5000, 1], alpha = 0.
 ↪3,cmap=plt.cm.RdYlGn,marker= 'o',
            s=20, c=df_compare.loc[:4999,'Diabetes'])

ax1.set_title("Scatter Plot Marking Actual Labels")


ax2.scatter(df_compare.iloc[:5000, 0], df_compare.iloc[:5000, 1], alpha = 0.
 ↪3,cmap=plt.cm.RdYlGn,marker= 'o',
            s=20, c=df_compare.loc[:4999,'Classifier_Labels'])
ax2.set_title("Scatter Plot Marking Classification Predicted Labels")


ax3.scatter(df_compare.iloc[:5000, 0], df_compare.iloc[:5000, 1], alpha = 0.
 ↪3,cmap=plt.cm.RdYlGn,marker= 'o',
            s=20, c=df_compare.loc[:4999,'Cluster_Labels'])
ax3.set_title("Scatter Plot Marking Labels Generated by clustering")
```

[66]:
```
Text(0.5, 1.0, 'Scatter Plot Marking Labels Generated by clustering')
```


Scatter Plot Marking Actual Labels
1.5
1.0
0.5
0.0
−0.5
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


Scatter Plot Marking Classification Predicted Labels
1.5
1.0
0.5
0.0
−0.5
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


Scatter Plot Marking Labels Generated by clustering
1.5
1.0
0.5
0.0
−0.5
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

1.8.1 Quanitfying the results of accuracy

[67]:
```
correct_positive =  df_compare.loc[(df_compare['Diabetes'] ==␣
↪df_compare['Cluster_Labels'])&(df_compare['Diabetes'] == 1) ,:]
total_known_positive = df_compare.loc[df_compare['Diabetes'] == 1 ,:]
percentage_positive = correct_positive.shape[0]/total_known_positive.shape[0]
print("Total Percentage of positive over actuals for clusterts",␣
↪percentage_positive)


#Comparison of Classifier Labels

correct_positive_cls =  df_compare.loc[(df_compare['Diabetes'] ==␣
↪df_compare['Classifier_Labels'])&(df_compare['Diabetes'] == 1) ,:]
total_known_positive_cls = df_compare.loc[df_compare['Diabetes'] == 1 ,:]
percentage_positive = correct_positive_cls.shape[0]/total_known_positive_cls.
↪shape[0]
print("Total Percentage of positive over actuals for classifier",␣
↪percentage_positive)
```

```
Total Percentage of positive over actuals for clusterts 0.648941896371002
Total Percentage of positive over actuals for classifier 0.7454035525085696
```

As we can observe there is a little difference of ~ 10 percent between the results of classifier i.e 0.65 and the results of clustering i.e. 0.745 (Classifier being the better one). But it somehow confirms our assumption of the label during the clustering. This is a good sign that the classifier and the clustering algorithm are trying to derive the labels with some level of accuracy.

2 Conclusion

- SVM is a very slow algorithmn since it is very computationaly intensive so we will take only a part of the dataset to run the classification on (around 10,000)
- We decided to use PCA since our dataset had 21 dimensions and although 21 dimensions is not very large by reducing to two dimensions we can understand and visualize the classifier better.
- We decided to not use all the 21 features since some features clearly were not contributing to better models as seen in Regression and Scatter Plot . So to decide which features to keep we used the decision tree classifier. We used this classifier since it is computationaly faster and is more explainable. The 8 key features ['BMI','HighBP','GeneralHealth','PhysicalHealth','MentalHealth','Age','Education','Income'] were then reduced to 2 using Principal Component Analysis
- Our Dataset is significantly imbalanced with positive class "Diabetes = Yes" is the minority. We are interested in catching positive cases. The classification reports show that Recall for class1 = 0.76 which is good but the precision is very low at 0.32. The model tends to generate many false positives.

- We are only showing models trained using undersampling since models without any sort of undersampling yield very low precision/ recall for class 1.

- Linear Kernel SVM is not able to generate good results
- SVM classifier is able to model non-linear decision boundaries. The model has a reasonable AUC = 0.77. Best Model has hyperparameters C= 15 Gamma = 5 with RBF Kernal.
- Finally we tried to compare the classifier and clustering results which shows that the clustering performance is around 0.65 and the classifier performance is about 0.745.

3 Classification Conclusion

For all the algorithms we studied, We began the classification with 21 features not all of which are meaningful to the classification. Hence we decided to get the feature importances using Decision Tree classifier first and reduced the features to be analyzed as 6. We further used Principal component analysis and reduced it to 2 features. Further, facing another issue of an imbalanced dataset i.e. 90 percent of non-diabetic and 10 percent of diabetic, we undersampled the majority class to bring it in terms with the minority class. All the algorithms were applied to the outcome of undersampled data.

Gaussian Naive Bayes got us a model a model with AUC as 0.76 which looked promising but the precision of the model was way less at 0.24 hence a lot of false negatives. Even with tuning hyper parameters such as Stratified K Fold and var_smoothing, there wasn't significant improvement. Hence we went to another classifier based on neural networks

The next two algorithms were computationally expensive and only a part of data was used to do further analysis.

We tried tuning the MLP Classifier for the diabetes dataset. The best neural network model turned out to be the one with 5 layers with 100 neurons in each layer, with Regularization alpha = 1 using The RELU activation function. The classifier reported Recall score for class 1 (minority) = 0.73 but the precision is very low at 0.31 tending to generate many false positives.

The last classifier in our tests was the Support Vector Classifier. SVM classifier is able to model non-linear decision boundaries. The model has a reasonable AUC = 0.77. Best Model has hyperparameters C= 15 Gamma = 5 with RBF Kernal.The classifier reported Recall for class 1 = 0.76 which is good but the precision is very low at 0.32.

Though there wasn't a lot to choose from the MLP but SVM performed the best among the various permutations and combinations of algorithms, our dataset and the hyper parameter tuning that we tried.

Also we observe that the comparison of best SVM model and best Clustering (Mini Batch K-Means) predicting the labels have some level of accuracy (Classifier - 0.745, CLustering - 0.65) and somehow confirms our assumption in the Clustering sheet.

4 REFERENCES

https://www.codecademy.com/learn/machine-learning/modules/dspath-clustering/cheatsheet https://mclguide.readthedocs.io/en/latest/sklearn/clusterdim.html https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://www.discoverbits.in/371/sklearn-attributeerror-predict_proba-available-probability
https://www.kaggle.com/residentmario/undersampling-and-oversampling-imbalanced-data

Material from Machine Learning Course, Seattle University

Material from Introduction to Data Science, Seattle University

5 –END PROJECT–