ECE 657A: Data and Knowledge Modelling and Analysis

Assignment 1: Data Cleaning and Classification

CM1 (DATA CLEANING)

Dataset 1 – Iris Dataset

1. Reading the data and observing the feature description

```
[5] #Loading the data
    iris = pd.read_csv('C:/Users/SAM/OneDrive - University of Waterloo/Spring 2021/ECE657A/Assignment 1/iris_dataset_missing.csv')
    df_iris = pd.DataFrame(iris)
    #Describing the data
    df_iris.describe()
```

	sepal_length	sepal_width	petal_length	petal_width
count	105.000000	101.000000	97.000000	105.000000
mean	5.858909	3.059083	3.812370	1.199708
std	0.861638	0.455116	1.793489	0.787193
min	4.344007	1.946010	1.033031	-0.072203
25%	5.159145	2.768688	1.545136	0.333494
50%	5.736104	3.049459	4.276817	1.331797
75%	6.435413	3.290318	5.094427	1.817211
max	7.795561	4.409565	6.768611	2.603123

We find the Iris dataset to have five columns (sepal_length sepal_width petal_length petal_width and species).

```
df_iris.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 105 entries, 0 to 104
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  105 non-null    float64
 1   sepal_width   101 non-null    float64
 2   petal_length  97 non-null     float64
 3   petal_width   105 non-null    float64
 4   species       105 non-null    object
dtypes: float64(4), object(1)
memory usage: 4.2+ KB
```

The data type of all is found to be float, except that of “species” feature which is an object.

```
[7] #numeric encoding the categorical species variables for easy data exploration
    df_iris.species = df_iris.species.replace({'Iris-versicolor':0,'Iris-setosa' : 1, 'Iris-virginica' : 2})
    df_iris.head(5)
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.045070	2.508203	3.018024	1.164924	0
1	6.325517	2.115481	4.542052	1.413651	0
2	5.257497	3.814303	1.470660	0.395348	1
3	6.675168	3.201700	5.785461	2.362764	2
4	5.595237	2.678166	4.077750	1.369266	0

As seen above, we have encoded the categorical species variables for easy data exploration.

2. <u>Checking for duplicate, unique and missing values</u>

```
[8] # rows that contain duplicate value
    dup_iris = df_iris.duplicated()
    print(dup_iris.any())

    #columns with single values
    print(df_iris.nunique())

    # Finding missing values in the data sets
    print(df_iris.isnull().sum())
```

```
False
sepal_length    105
sepal_width     101
petal_length     97
petal_width     105
species           3
dtype: int64
sepal_length    0
sepal_width     4
petal_length    8
petal_width     0
species         0
dtype: int64
```

No duplicate values are found, however a few missing values are detected in sepal_width and petal_length.

3. <u>Filling the missing values in the Iris dataset</u>

```
#introducing index column to sort back the data to its original order after interpolation.
df_iris.reset_index(level=0, inplace=True)
#sorting data based on species column
df_iris = df_iris.sort_values(by='species')
#Using interpolation
df_iris = df_iris.interpolate()
#sorting the data back to its original order
df_iris = df_iris.sort_values(by='index')
df_iris.drop(["index"], axis = 1, inplace = True)
#df_iris.head(25)
print(df_iris.isnull().sum())
```

```
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
species         0
dtype: int64
```

The interpolation method was implemented for filling in the missing values in the sepal_width and petal_length columns. Upon checking for missing values after this method, none were found.

4. Noise reduction

The main source of noise detected in the Iris dataset was the negative values present in the petal_width column. Since a width measure can't have a negative value, we converted the negative values into it's absolute value.
We also rounded off the decimal values of the features to their first decimal place.

```
#removing negative values
df_iris['petal_width'] = df_iris['petal_width'].abs()
# rounding off the values of each columns.
df_iris['sepal_length'] = df_iris['sepal_length'].round(decimals=1)
df_iris['sepal_width'] = df_iris['sepal_width'].round(decimals=1)
df_iris['petal_length'] = df_iris['petal_length'].round(decimals=1)
df_iris['petal_width'] = df_iris['petal_width'].round(decimals=1)
df_iris.head(10)
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.0	2.5	3.0	1.2	0
1	6.3	2.1	4.5	1.4	0
2	5.3	3.8	1.5	0.4	1
3	6.7	3.2	5.8	2.4	2
4	5.6	2.7	4.1	1.4	0
5	6.7	3.1	5.0	2.4	2
6	4.8	3.0	1.5	0.0	1
7	5.2	NaN	1.7	0.1	1
8	4.4	2.9	1.4	0.4	1
9	6.8	3.1	5.9	2.2	2

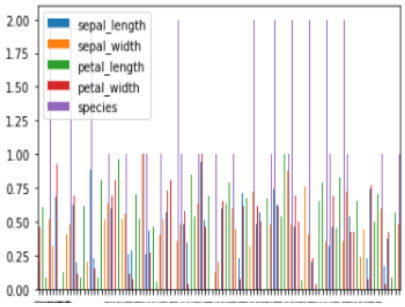
Further noise reduction is done by the normalizing technique, as normalizing helps reduce any unwanted noise while preserving the original data.

5. Normalization using min-max and Z-score methods

(i) Using the min-max normalization

```
#---------- MIN-MAX TECHNIQUE ----------#

# copy the data and keeping the target attribute away from normalization.
species = df_iris['species'].tolist()
df_min_max_scaled_iris = df_iris.iloc[:,0:4].copy()

# apply normalization max-min feature scaling technique
for column in df_min_max_scaled_iris.columns:
    df_min_max_scaled_iris[column] = (df_min_max_scaled_iris[column] - df_min_max_scaled_iris[column].min()) / (df_min_max_scaled_iris[column].max() - df_min_max_scaled_iris[column].min())

df_min_max_scaled_iris['species'] = species

# view normalized data
df_min_max_scaled_iris.plot(kind='bar')
print(df_min_max_scaled_iris)
```

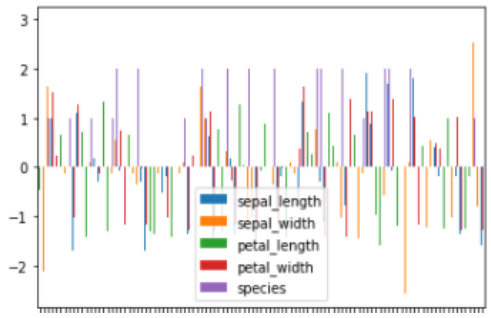
```
     sepal_length  sepal_width  petal_length  petal_width  species
0        0.200000         0.24      0.344828     0.461538        0
1        0.571429         0.08      0.603448     0.538462        0
2        0.285714         0.76      0.086207     0.153846        1
3        0.685714         0.52      0.827586     0.923077        2
4        0.371429         0.32      0.534483     0.538462        0
..            ...          ...           ...          ...      ...
100      0.171429         0.52      0.103448     0.038462        1
101      0.371429         0.36      0.431034     0.423077        0
102      0.342857         0.92      0.086207     0.076923        1
103      0.342857         0.32      0.568966     0.500000        0
104      0.057143         0.48      0.086207     0.076923        1

[105 rows x 5 columns]
```



(ii) Using the Z-score method

```
#----------- Z-SCORE TECHNIQUE -----------#
# copy the data and keeping the target attribute away from normalization.
df_z_scaled_iris = df_iris.copy()
columns_to_scale = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

# Using StandardScaler() method from sklearn.preprocessing package for normalizing the data using Z-Score normalization technique.
stds = StandardScaler()
df_z_scaled_iris[columns_to_scale] = stds.fit_transform(df_z_scaled_iris[columns_to_scale])


# view normalized data
df_z_scaled_iris.plot(kind='bar')
print(df_z_scaled_iris)
```

```
     sepal_length  sepal_width  petal_length  petal_width  species
0       -0.993164    -1.229456     -0.456238    -0.008468        0
1        0.521550    -2.113265      0.384049     0.245561        0
2       -0.643615     1.642921     -1.296526    -1.024583        1
3        0.987616     0.317208      1.112298     1.515706        2
4       -0.294065    -0.787552      0.159972     0.245561        0
..            ...          ...           ...          ...      ...
100     -1.109680     0.317208     -1.240507    -1.405626        1
101     -0.294065    -0.566600     -0.176143    -0.135482        0
102     -0.410582     2.526729     -1.296526    -1.278612        1
103     -0.410582    -0.787552      0.272011     0.118547        0
104     -1.575746     0.096256     -1.296526    -1.278612        1

[105 rows x 5 columns]
```



Dataset 2 – Heat Disease dataset

1. Reading the data and observing the feature description

```
df_hd= pd.read_csv("heart_disease_missing.csv")
df_hd.describe()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
count	212.000000	212.000000	212.000000	205.000000	202.000000	212.000000	207.000000	208.000000	212.000000	200.000000	210.000000	212.000000	211.000000	212.000000
mean	54.311321	0.688679	0.957547	131.784610	244.133256	0.132075	0.560386	149.647978	0.344340	1.113106	1.423810	0.731132	2.349112	0.542453
std	9.145339	0.464130	1.022537	18.057222	46.444257	0.339374	0.535149	22.076206	0.476277	1.255908	0.623622	1.038762	0.602117	0.499374
min	29.000000	0.000000	0.000000	93.944184	126.085811	0.000000	0.000000	88.032613	0.000000	-0.185668	0.000000	0.000000	0.858554	0.000000
25%	47.000000	0.000000	0.000000	119.968114	211.969594	0.000000	0.000000	135.946808	0.000000	0.050778	1.000000	0.000000	1.949795	0.000000
50%	55.000000	1.000000	1.000000	130.010256	241.467023	0.000000	1.000000	151.939216	0.000000	0.726060	1.000000	0.000000	2.078759	1.000000
75%	61.000000	1.000000	2.000000	139.965470	272.484222	0.000000	1.000000	165.260092	1.000000	1.816733	2.000000	1.000000	2.970842	1.000000
max	77.000000	1.000000	3.000000	192.020200	406.932689	1.000000	2.000000	202.138041	1.000000	6.157114	2.000000	4.000000	3.277466	1.000000

```
df_hd.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 212 entries, 0 to 211
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       212 non-null    int64
 1   sex       212 non-null    int64
 2   cp        212 non-null    int64
 3   trestbps  205 non-null    float64
 4   chol      202 non-null    float64
 5   fbs       212 non-null    int64
 6   restecg   207 non-null    float64
 7   thalach   208 non-null    float64
 8   exang     212 non-null    int64
 9   oldpeak   200 non-null    float64
 10  slope     210 non-null    float64
 11  ca        212 non-null    int64
 12  thal      211 non-null    float64
 13  target    212 non-null    int64
dtypes: float64(7), int64(7)
memory usage: 23.3 KB
```

The heart disease dataset consists of 14 feature columns, (age, sex, cp, trestbps, chol, fbs, restecg,thalach,exang, oldpeak, slope, ca, thal and target)

2. Checking for duplicate, unique and missing values

```
# rows that contain duplicate value
dup_hd = df_hd.duplicated()
print(dup_hd.any())

#columns with single values
print(df_hd.nunique())

# Finding missing values in the data sets
print(df_hd.isnull().sum())
```

```
False
age          41
sex           2
cp            4
trestbps    205
chol        202
fbs           2
restecg       3
thalach     208
exang         2
oldpeak     200
slope         3
ca            5
thal        211
target        2
dtype: int64
age          0
sex          0
cp           0
trestbps     7
chol        10
fbs          0
restecg      5
thalach      4
exang        0
oldpeak     12
slope        2
ca           0
thal         1
target       0
```

No duplicate values are found to be present in the dataset. However a missing values were detected in the following columns: (‘trestbbps’, ‘chol’, ‘restecg’, ‘thalach’, ‘oldpeak’, ‘slope’ and ‘thal’).

3. Filling the missing values in the Heart disease dataset

The missing values are present in seven columns as mentioned above, which consist of 3 categorical and 4 numerical features.

Categorical Columns: restecg, slope, thal

Numeric Columns: trestbps, chol, thalach, oldpeak

For the numerical columns, we used the interpolate method for replacing the missing values.

```
# Filling in the missing values in NUMERICAL Features using interpolation.

df_hd['trestbps'] = df_hd['trestbps'].interpolate()
df_hd['chol'] = df_hd['chol'].interpolate()
df_hd['thalach'] = df_hd['thalach'].interpolate()
df_hd['oldpeak'] = df_hd['oldpeak'].interpolate()
```

For the categorical columns, we follow a different approach.

-The Thal feature consists of a single missing value and is a noisy feature. Hence, we drop the entire row with the missing value in the Thal column.

-For the restecg & slope features, we are training a classifier over the said feature columns with missing values as a dependent variable against other features of our data set. We then impute based on the newly trained classifier.

Predicting the missing values of the said features, using this classifier. This will ensure that the missing values being filled are taken from similar rows.

A logistic regression model is used for the said classifier model.

After conducting the aforementioned steps, we find no null values in our dataset.

```
#-----1. FEATURE : thal
df_hd = df_hd.dropna( how='any', subset=['thal'])

#-----2. FEATURE : restecg
#making the 'restecg' column as the first column.
df_hd = df_hd.reindex(columns=['restecg', 'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target'])

#Make non-missing records as our Training data.
df_hd_train = df_hd.dropna()
#Make missing records as our Testing data.
df_hd_test=pd.concat([df_hd_train,df_hd]).drop_duplicates(keep=False)

#Separate Dependent and Independent features.
x_train = df_hd_train.iloc[:, 1:14]
y_train = df_hd_train.iloc[:, 0]
#Fit our Logistic Regression model.
lr = LogisticRegression()
lr.fit(x_train,y_train)
#predicting the missing data in 'restecg' column.
i=0
for i in range(df_hd_test.shape[0]):
    if math.isnan(df_hd_test.iloc[i,0]):
        input = [df_hd_test.iloc[i,1:14].tolist()]
        x = lr.predict(input)
        df_hd_test.iloc[i,0] = x
#Combining Dependent and Independent features.
df_hd =pd.concat([df_hd_train,df_hd_test])
```

```
#-----3. FEATURE : slope
#making the 'slope' column as the first column.
df_hd = df_hd.reindex(columns=['slope', 'restecg', 'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'thalach', 'exang', 'oldpeak', 'ca', 'thal', 'target'])

#Make non-missing records as our Training data.
df_hd_train = df_hd.dropna()
#Make missing records as our Testing data.
df_hd_test=pd.concat([df_hd_train,df_hd]).drop_duplicates(keep=False)

#Separate Dependent and Independent variables.
x_train = df_hd_train.iloc[:, 1:14]
y_train = df_hd_train.iloc[:, 0]
#Fit our Logistic Regression model.
lr1 = LogisticRegression()
lr1.fit(x_train,y_train)
#predicting the missing data in 'slope' column.
i=0
for i in range(df_hd_test.shape[0]):
    if math.isnan(df_hd_test.iloc[i,0]):
        input = [df_hd_test.iloc[i,1:14].tolist()]
        x = lr1.predict(input)
        df_hd_test.iloc[i,0] = x
#Combining Dependent and Independent features.
df_hd =pd.concat([df_hd_train,df_hd_test])
```

```
print('number of missing values in each column after the preprocessing: /n')
df_hd.isnull().sum()
```

```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
ca          0
thal        0
target      0
dtype: int64
```

4. <u>Noise reduction</u>

   The main source of noise detected in the Heart disease dataset were the thal and oldpeak features. The thal feature is a categorical value consisting of values – 0,1,2 and 3. However in the dataset the thal values are of float type. Hence the we have rounded off the thal values.

   The oldpeak feature consisted of negative values, which were corrected by taking their absolute values.

```
# Rounding of the thal feature to interget values because it is a categorical value and cannot contain deimal values.
df_hd['thal'] = round(df_hd['thal'])
# Taking absolute values of all the values in oldpeak feature to remove the negative values.
df_hd['oldpeak'] = df_hd['oldpeak'].abs()
df_hd.head(5)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	76	0	2	140.102822	197.105970	0	2.0	115.952071	0	1.284822	1.0	0	2.0	1
1	43	0	0	132.079599	341.049462	1	0.0	135.970028	1	3.110483	1.0	0	3.0	0
2	47	1	2	107.899290	242.822816	0	1.0	152.210039	0	0.023723	2.0	0	2.0	0
3	51	1	2	99.934001	288.887585	0	1.0	143.049207	1	1.195082	1.0	0	2.0	1
4	57	1	0	110.103508	334.952353	0	1.0	143.099327	1	3.082052	1.0	1	3.0	0

Further noise reduction is done by the normalizing technique, as normalizing helps reduce any unwanted noise while preserving the original data.

5. Normalization using min-max and Z-score methods

(i) Using the min-max normalization

```
#---------- MIN-MAX TECHNIQUE ----------#

# copy the data and keeping the target attribute away from normalization.
df_mm_scaled_hd = pd.get_dummies(df_hd,columns = ['sex' , 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal'])

# apply normalization techniques
for column in columns_to_scale:
    df_mm_scaled_hd[columns_to_scale] = (df_mm_scaled_hd[columns_to_scale] - df_mm_scaled_hd[columns_to_scale].min()) / (df_mm_scaled_hd[columns_to_scale].max() - df_mm_scaled_hd[columns_to_scale].min())

# view normalized data
print(df_mm_scaled_hd)
```

```
          age  trestbps      chol   thalach  ...  ca_4  thal_1.0  thal_2.0  thal_3.0
0    0.979167  0.470641  0.252879  0.244681  ...     0         0         1         0
1    0.291667  0.388835  0.765412  0.420115  ...     0         0         0         1
2    0.375000  0.142289  0.415661  0.562440  ...     0         0         1         0
3    0.458333  0.061073  0.579682  0.482156  ...     0         0         1         0
4    0.583333  0.164763  0.743703  0.482595  ...     0         0         0         1
..        ...       ...       ...       ...  ...   ...       ...       ...       ...
62   0.833333  0.471328  0.403103  0.550832  ...     0         0         1         0
155  0.354167  0.073027  0.252335  0.594882  ...     0         0         0         1
160  0.625000  0.449179  0.516254  0.824216  ...     0         0         1         0
17   0.687500  0.367388  0.487383  0.077133  ...     0         0         0         1
26   0.708333  0.468111  0.216629  0.489821  ...     0         0         0         1
```

(ii) Using the Z-score method

```
#----------- Z-SCORE TECHNIQUE -----------#

#applying one hot encoding to convert all the categorical variable to dummy variables for scaling them.
df_z_scaled_hd = pd.get_dummies(df_hd,columns = ['sex' , 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal'])

# copy the data and keeping the target attribute away from normalization.
columns_to_scale = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

stds = StandardScaler()
df_z_scaled_hd[columns_to_scale] = stds.fit_transform(df_z_scaled_hd[columns_to_scale])

print(df_z_scaled_hd)
```

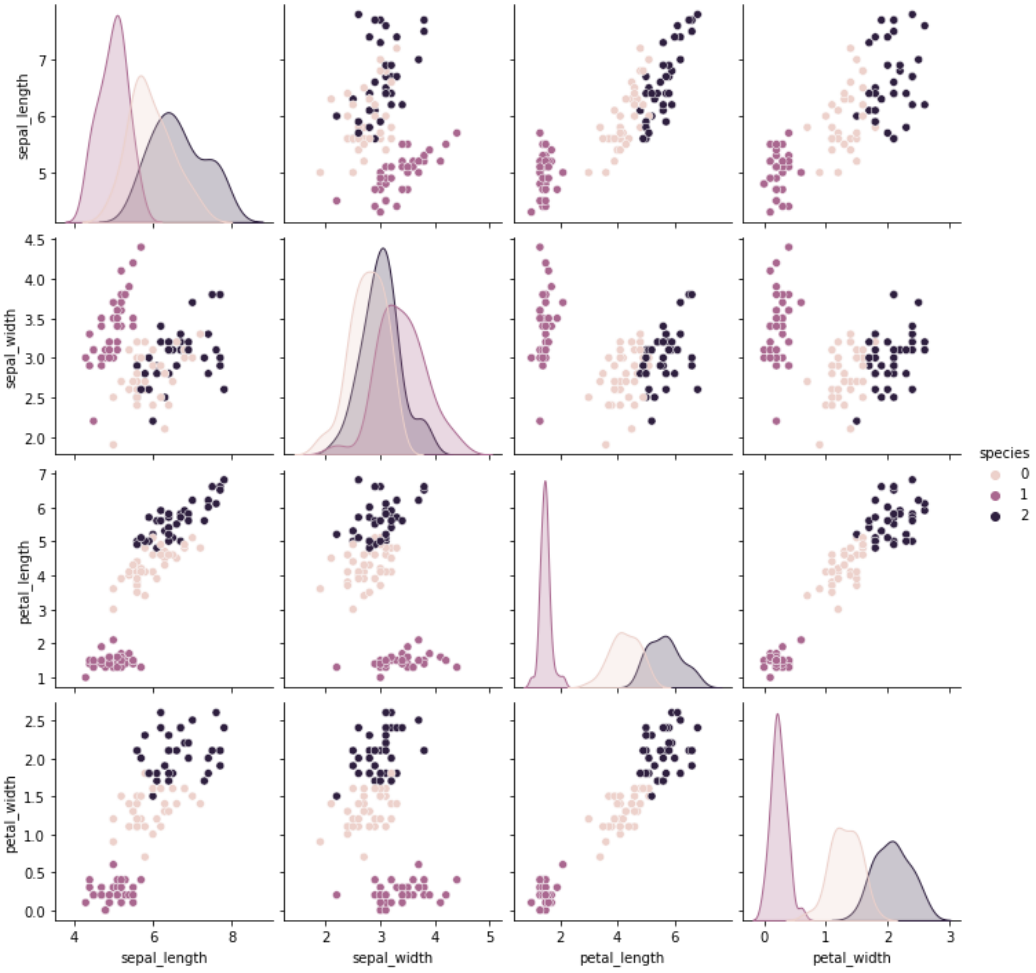
```
          age  trestbps      chol   thalach  ...  ca_4  thal_1.0  thal_2.0  thal_3.0
0    2.370992  0.446721 -1.016974 -1.556113  ...     0         0         1         0
1   -1.237580  0.003822  2.111762 -0.636946  ...     0         0         0         1
2   -0.800177 -1.330981 -0.023279  0.108748  ...     0         0         1         0
3   -0.362775 -1.770681  0.977978 -0.311890  ...     0         0         1         0
4    0.293329 -1.209303  1.979236 -0.309589  ...     0         0         0         1
..        ...       ...       ...       ...  ...   ...       ...       ...       ...
62   1.605537  0.450438 -0.099938  0.047929  ...     0         0         1         0
155 -0.909528 -1.705966 -1.020294  0.278727  ...     0         0         0         1
160  0.512031  0.330524  0.590785  1.480297  ...     0         0         1         0
17   0.840083 -0.112293  0.414546 -2.433965  ...     0         0         0         1
26   0.949433  0.433020 -1.238257 -0.271728  ...     0         0         0         1
```

CM2 (Pair-plot Visualizations)

Dataset 1 – Iris Dataset

1. Using the seaborn library, a pair-plot is plotted for the features of the dataset.

```
#pairplot for iris dataset
sns.pairplot(df_iris,hue="species")
```



The species are denoted as follows :

0- Iris-versicolor
1- Iris-setosa
2- Iris-virginica

The key observations from the scatter-plot are –

- In all the pair plots it can be seen that the versicolor and virginica species are relatively similar to one another and consequently their features display homogenous correlations.
- The petal length and sepal length features appear to have a positive correlation especially for the versicolor and virginica species.
- The petal length and petal width are also positively correlated for the versicolor and virginica species.
- The setosa species displays a positive correlation between it's sepal width and sepal length feature. While also having a very slight positive correlation between the petal length and petal width.

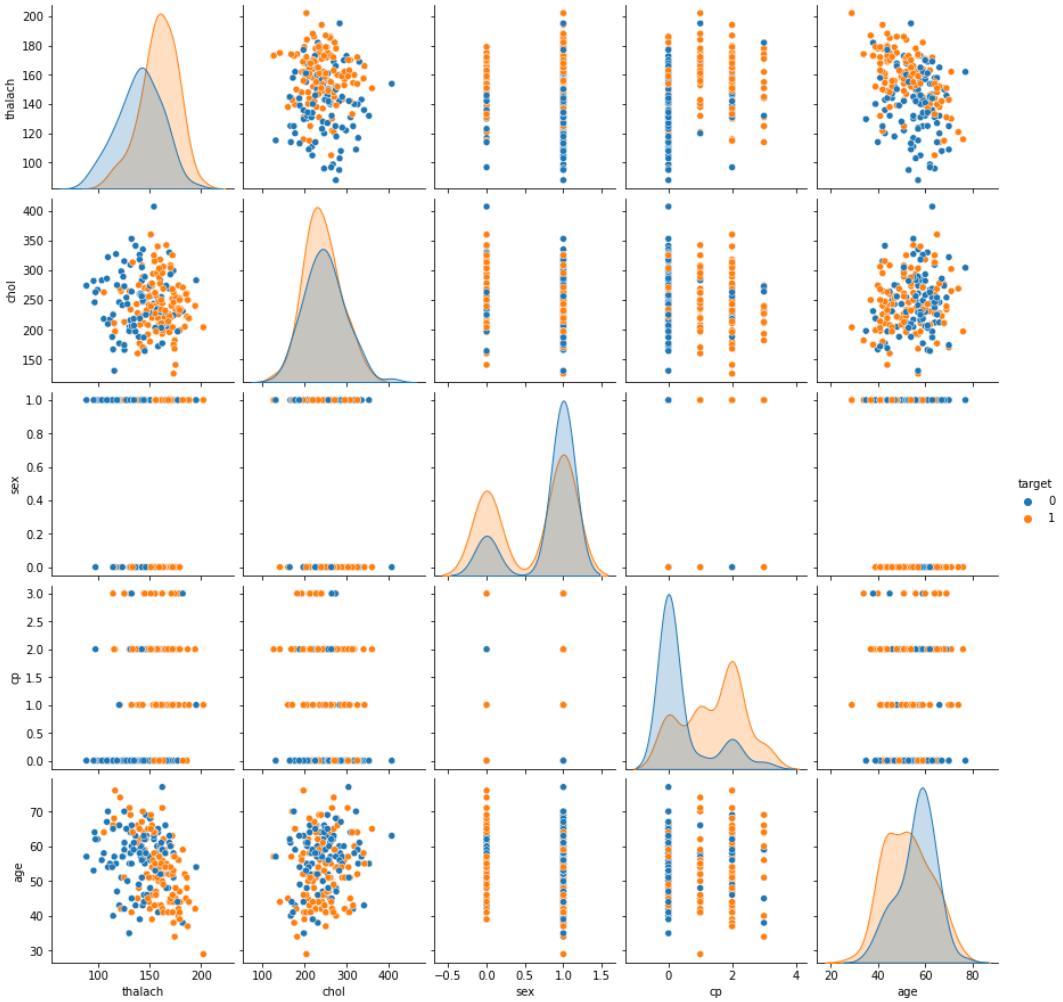
Dataset 2 – Heart Disease dataset

1. Using the seaborn library, a pair-plot is plotted for a select few features from the heart disease dataset.

   The features selected are –
   Age
   Thalach (Maximum heart rate achieved during thalium stress test)
   Cp (Chest pain type (0 = Asymptomatic angina; 1 = Atypical angina; 2 = Non-angina; 3 = Typical angina)
   Sex (0 = female; 1 = male)
   Chol (Serum cholesterol)

   These features are colour encoded by the target variable (1 = heart disease; 0 = no heart disease)

```
#pairplot for iris dataset
sns.pairplot(df_hd, vars = ['thalach', 'chol', 'sex', 'cp','age'],
             hue = 'target')
```



The key observations from the obtained pair plot are –

- Individuals with cholesterol level in the range of 200 – 300 mg/dl are found to have the highest cases of heart disease.
- Thalach heart rate is observed to be highly correlated with the target variable, indicating high cases of heart disease for thalach value greater than 150.
- The age feature provides an interesting insight. It can be observed that higher cases of heart disease seem to occur for individuals aged between 30 and 60, whereas the age group between 50 and 80 have the least cases of heart disease. If an ind has crossed a certain age approx50 w/o heart dis, the likelihood of having heart disease reduces drastically in coming ages.
- From the sex feature it is clearly visible that males have a higher tendency of having heart disease as compared to the females. However, for our dataset the ratio of males with heart disease to males without heart disease is much lower as compared to the ratio of females with heart disease to females without heart disease.
- The chest pain feature is found to indicate the lowest cases of heart disease for cp=0 (asymptomatic angina) which is quite a common insight. However, the feature depicts the highest risk of heart disease for cp=2 (non-angina) instead of cp=3 (typical angina). This is a valuable insight as it implies that individuals with non-angina are found to be more susceptible to heart disease than people with typical angina(which is normally a primary symptom of heart disease)

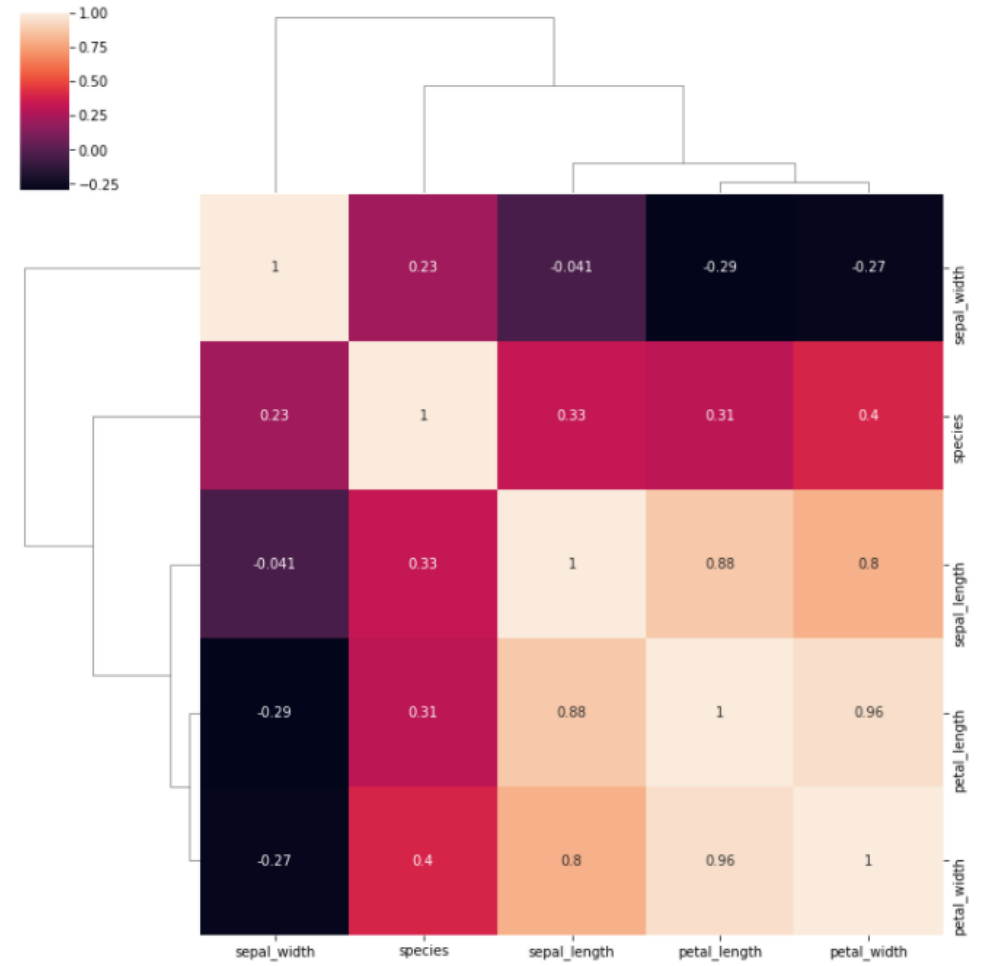
[CM3] Correlation and Statistical Measures

Dataset 1 – Iris Dataset

1. Using the seaborn library, a cluster map is plotted for observing the correlation between the features of the dataset and providing the correlation values as well.

```
#clustermap for iris dataset
sns.clustermap(df_z_scaled_iris.corr(),annot=True)
```

```
<seaborn.matrix.ClusterGrid at 0x7fb345d6a750>
```



-It can be observed that petal length and petal width have the highest correlation in the data, while sepal width and petal length are highly negatively correlated.

- The sepal width is the only feature to not have a significant correlation with the other features.

2. Calculating the mean, variance, kurtosis and skew of the iris dataset.

```
#Calculating mean,variance,kurtosis and skew
```

```
df_iris.mean()
```

```
sepal_length    5.852381
sepal_width     3.056436
petal_length    3.814433
petal_width     1.206667
species         1.000000
dtype: float64
```

```
df_iris.var()
```

```
sepal_length    0.743672
sepal_width     0.206883
petal_length    3.219790
petal_width     0.625821
species         0.673077
dtype: float64
```

```
df_iris.kurt()
```

```
sepal_length   -0.523430
sepal_width     0.480089
petal_length   -1.390350
petal_width    -1.331128
species        -1.514563
dtype: float64
```

```
df_iris.skew()
```

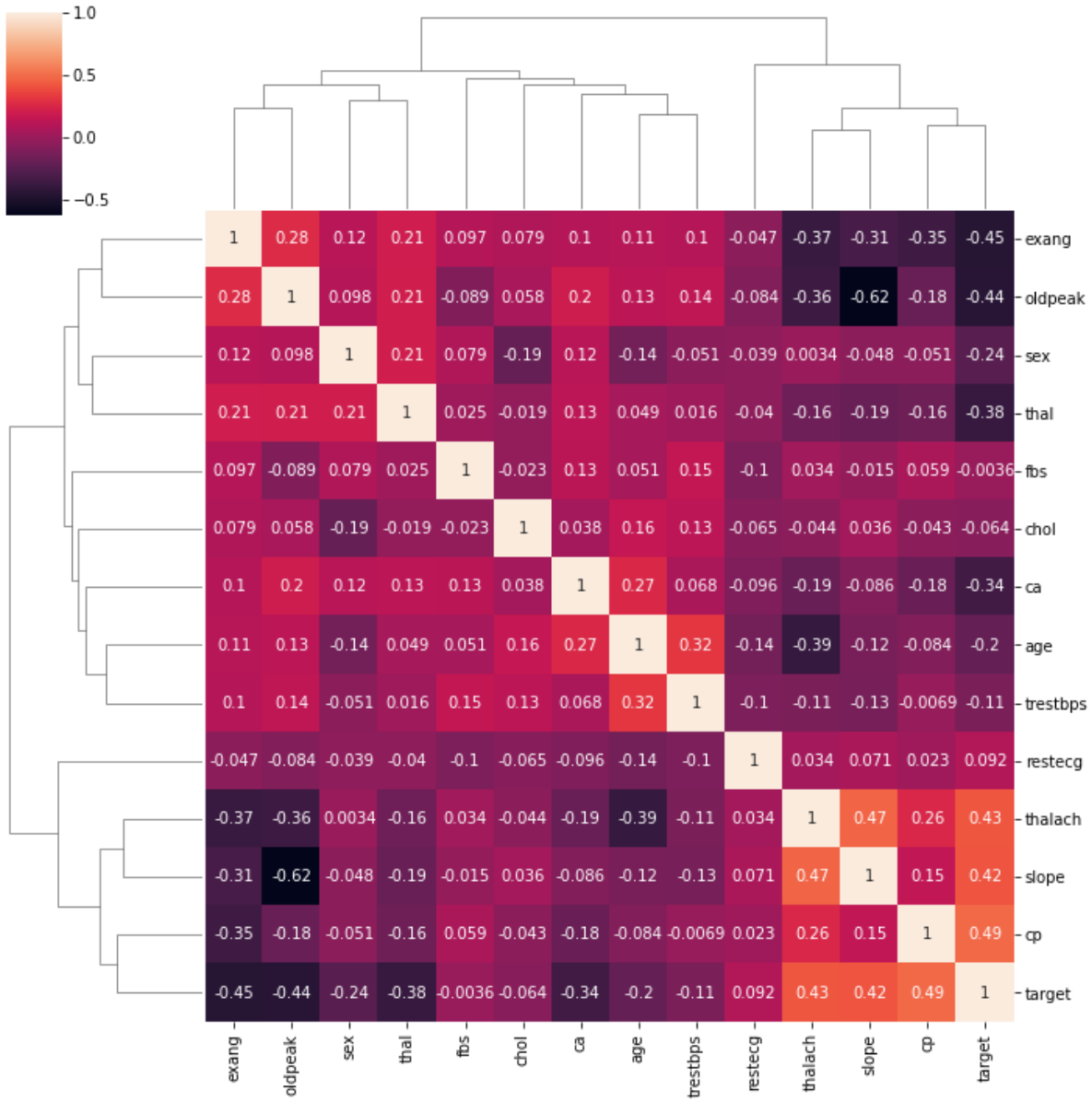
```
sepal_length    0.389386
sepal_width     0.308140
petal_length   -0.251411
petal_width    -0.060679
```

- The mean of the sepal length being greater than the sepal width and the petal measurements, indicates the longer size of the sepal.
- Almost 0 variance for majority of the features indicates an approximate normal distribution
- The kurtosis values for all the features except sepal width indicate a platykurtic distribution (k <0) which signifies the presence of small outliers in the data. However, the value is quite small in this case to hold any significance.
- The skewness is a measure of symmetry within the dataset. Since all the skew values in our observation lie in the range between -1 and 0.5, the data is moderately skewed.

Dataset 2 – Heart disease dataset

1. Using the seaborn library a cluster map is plotted for observing the correlation between the features of the dataset.

```
#clustermap of heart disease dataset
sns.clustermap(df_hd.corr(),annot=True)
```



From the clustermap, we can find three features to be highly positively correlated with our target variable (denoting heart disease).
These features are – “cp”, “slope” and “thalach”
This denotes that chest pain and the heart rate during the thalium stress test(thalach), are the greatest indicators of heart disease cases.

Similarly, three features are found to have a high negative correlation with our target variable.
These features are – “oldpeak”, “exang” and “thal".
This denotes that with lower values or occurrences of the above features, there is a higher chance of heart disease.

Interestingly, features such as “chol” (cholesterol level) and fbs(fasting blood sugar level) have negligible linear relation with respect to the target variable, implying very low significance for determining heart disease cases.

2. **Calculating the mean, variance, kurtosis and skew of the heart disease dataset.**

```
#Calculating mean,variance,kurtosis and skew
```

```
df_hd.mean()
```

```
age          54.317536
sex           0.691943
cp            0.952607
trestbps    132.010357
chol        243.893813
fbs           0.132701
restecg       0.568720
thalach     149.841674
exang         0.345972
oldpeak       1.133002
slope         1.417062
ca            0.734597
thal          2.355450
target        0.540284
dtype: float64
```

```
df_hd.var()
```

```
age          84.027262
sex           0.214173
cp            1.045362
trestbps    329.725518
chol       2126.715688
fbs           0.115640
restecg       0.284541
thalach     476.556094
exang         0.227353
oldpeak       1.457800
slope         0.387136
ca            1.081607
thal          0.344482
target        0.249560
dtype: float64
```

```
df_hd.kurt()
```

```
age        -0.572421
sex        -1.311205
cp         -1.227403
trestbps    0.472414
chol        0.252379
fbs         2.782522
restecg    -1.191437
thalach    -0.142336
exang      -1.589724
oldpeak     1.689021
slope      -0.585027
ca          1.000649
thal       -0.676980
target     -1.992466
dtype: float64
```

```
df_hd.skew()
```

```
age        -0.107835
sex        -0.837445
cp          0.473343
trestbps    0.634186
chol        0.323569
fbs         2.180880
restecg     0.102549
thalach    -0.410495
exang       0.652254
oldpeak     1.307114
slope      -0.579313
ca          1.371002
thal       -0.276223
target     -0.162823
dtype: float64
```

- The high variance of 2126.715688 indicates that a large number of cholesterol data points lie away from the mean of the distribution, implying a scattered distribution.
- The kurtosis level of the majority of the features are less than 0 , indicating a platykurtic distribution. Hence they consist of small outliers in the dataset.
  The kurtosis for the “fbs” feature is quite high, with a leptokurtic (k>0) distribution. This implies the presence of a large number of outliers for this feature.

[CM4]: Dividing the data into a training set and a test set. Train the model with the classifier's default parameters

Dataset 1- Iris Dataset

1. The data is divided into a training set and testing set.

```
# We are using input as Z scaled version of the cleaned data, i.e., df_z_scaled_iris.
# Independent features
x = df_z_scaled_iris.drop(['species'], axis=1)
# Dependent features
y = df_z_scaled_iris['species']
knn_scores = []
# Checking the accuracy for different sizes of the train-test sets, starting for 95%-5% to 55%-45% (train%-test%)
for i in arange(0.05,0.50,0.05):
    # Splitting the data
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = i, random_state = 98)
    # KNeighborsClassifier with default parameters.
    knn1 = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
    knn1.fit(x_train, y_train)
    yhat = knn1.predict(x_test)
    # Storing the accuracy scores in an array.
    knn_scores.append(accuracy_score(y_test, yhat))
    print(knn_scores)
```

```
[1.0]
[1.0, 1.0]
[1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9761904761904762]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9761904761904762, 0.9791666666666666]
```

The Z-score normalized version of the data is used. The “species” variable is our dependent variable, with the remaining features as the independent variables.

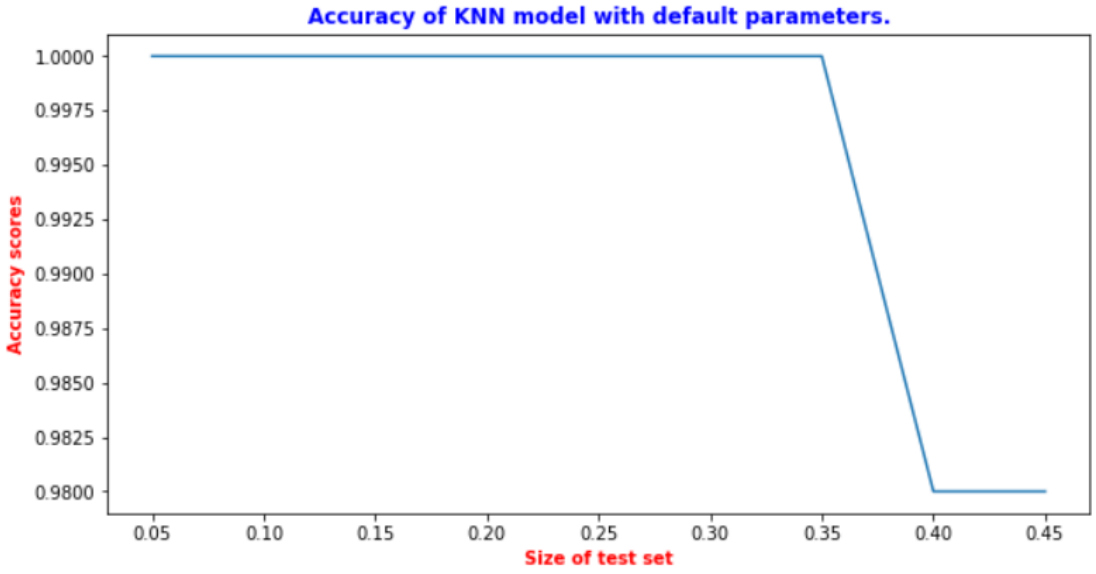
We initially check the accuracy of our knn classifier model for varying train-test sizes.

The data is then split and the training set is fit in our knn model, for which we have taken the default parameters.

The model prediction is then calculated using the test data set and the resulting knn accuracy scores are stored in an array.

```
# Rounding off the scores to 2 decimal places.
knn_scores = [ '%.2f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]

# Plotting the graph of accuracy scores against the size of test set.
x_axis = ['0.05', '0.10', '0.15', '0.20', '0.25', '0.30', '0.35', '0.40', '0.45']
f, ax = plt.subplots(1, 1, figsize = (10, 5))
plt.plot(x_axis, knn_scores)
plt.xlabel('Size of test set',color='Red',weight='bold')
plt.ylabel('Accuracy scores',color='Red',weight='bold')
plt.title('Accuracy of KNN model with default parameters.',color='Blue',weight='bold')
print(knn_scores)
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.98, 0.98]
```



We can see that, for the provided dataset, the knn classifier with default parameters is giving the maximum accuracy of 100% for if the train set size is at least 65%, i.e., the test set size is at least 35%.

If the train test size is taken below 65% then the model will not show its best performance. hence, we infer that the size of training set should always be considerably larger than that of the testing set.

The accuracy scores are reported as well, which shows an accuracy of 100% for the train-test split ratio of 80%-20% (followed as per the guideline provided in the assignment)

As shown in the code snippet below, for further use in the assignment, we will be taking test size = 0.20, i.e., training set and a test set (80%, 20%)

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 98)
# shape of train and test sets,
print(' Size of training (80%) is - {0},{1}'.format(x_train.shape, y_train.shape))
print(' Size of testing (20%) is - {0},{1}'.format(x_test.shape, y_test.shape))
```

```
 Size of training (80%) is - (84, 4),(84,)
 Size of testing (20%) is - (21, 4),(21,)
```

Dataset 2 – Heart Disease Dataset

1. The data is divided into a training set and testing set of size 80% and 20% respectively (as per the assignment guidelines)
2. The Z-score normalized version of the dataset is used
3. We now train the knn model with the default parameters.
4. Initially we run the model on the raw data (unprocessed and unclean), to provide a comparison later on with the model run on the processed data.

```
#Training the classifier on RAW data BEFORE preprocessing.
#Slpiting the data into a training set and a test set.
x = df_hd.drop(['target'], axis=1)
y = df_hd['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 98)

# KNeighborsClassifier with default parameters.
knn1 = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
knn1.fit(x_train, y_train)
yhat = knn1.predict(x_test)
print('Accuracy Score of classifier on RAW data BEFORE preprocessing:', accuracy_score(y_test, yhat))
```

```
Accuracy Score of classifier on RAW data BEFORE preprocessing: 0.7209302325581395
```

5. Now we run the model on the processed and Z-score normalized data.

```
# We are using input as Z scaled version of the cleaned data, i.e., df_z_scaled_hd.
#Training the classifier on data AFTER preprocessing.
#Slpiting the data into a training set and a test set.
x = df_z_scaled_hd.drop(['target'], axis=1)
y = df_z_scaled_hd['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 98)

# KNeighborsClassifier with default parameters.
knn0 = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
knn0.fit(x_train, y_train)
yhat = knn0.predict(x_test)
print('Accuracy Score of classifier on data AFTER preprocessing:', accuracy_score(y_test, yhat))
```

```
Accuracy Score of classifier on data AFTER preprocessing: 0.813953488372093
```

We can observe that the KNN classifier is showing an accuracy of 81.3% when taken with the default parameters for the Z-score normalized data which is significantly better that the one obtained with the raw data before the pre-processing, i.e., 72.1%

[CM5]: Tuning the Classifier

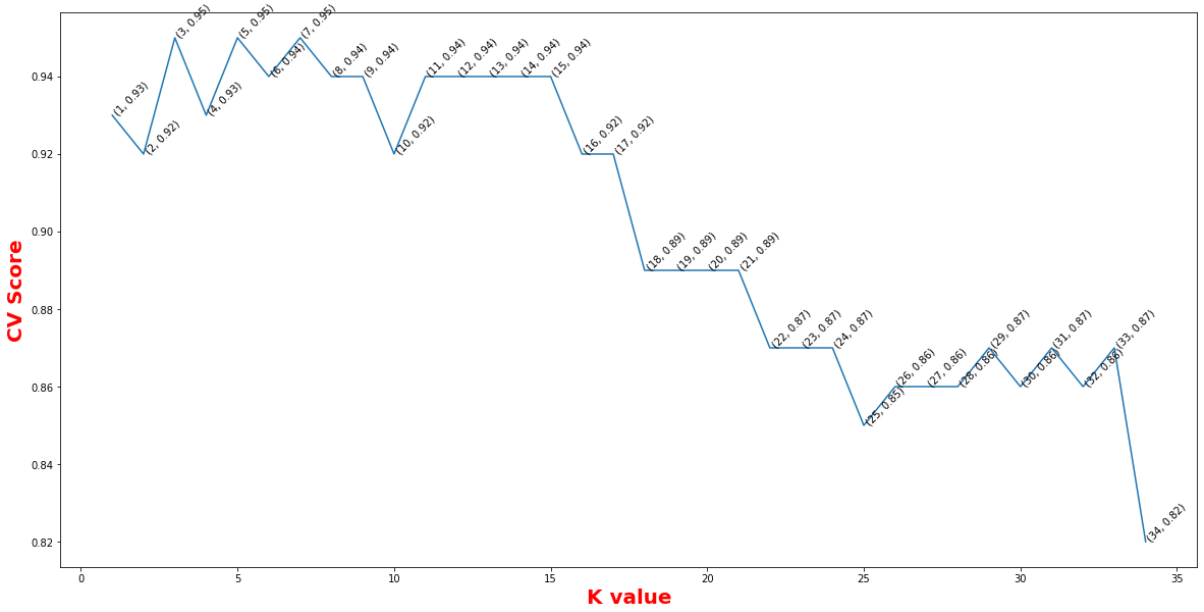
Dataset 1- Iris Dataset

1. For the iris dataset we make use of the 5-fold cross validation for tuning the classifier.

```
# Using 5-fold cross validation on the training set to train the classifer and produce accuracy scores.
cv_scores = []
# k values
neighbors = list(np.arange(1,35,1))
#training the knn classifier for different values of k along with k-fold cross validation (for tuning).
for n in neighbors:
    knn2 = KNeighborsClassifier(n_neighbors = n)
    cross_val = cross_val_score(knn2,x_train,y_train,cv = 5 , scoring = 'accuracy')
    cv_scores.append(cross_val.mean())

# Rounding off the scores to 2 decimal places.
cv_scores = [ '%.2f' % elem for elem in cv_scores ]
cv_scores = [float(line) for line in cv_scores]

# Plotting the graph of accuracy scores against k values.
f, ax = plt.subplots(1, 1, figsize = (20, 10))
plt.xlabel("K value",color='Red',weight='bold',fontsize='20')
plt.ylabel("CV Score",color='Red',weight='bold',fontsize='20')
for i in range(1,35):
    plt.text(i, cv_scores[i-1], (i, cv_scores[i-1]), fontsize=10, rotation=45)
plt.plot(neighbors, cv_scores)
```

The classifier is trained and the subsequent accuracy scores are stored in an array and then plotted for k-values ranging from 1 to 35.

The output of the above code is given below.



We can see that, for the provided dataset, the knn classifier with k-fold cross validation (tuning approach) is giving the maximum accuracy of 95% for k = 3. We can also observe that the accuracy is showing a decreasing trend with the increasing values of k.

Although this observed trend is not strictly decreasing. Therefore, the accuracy is not always affected the same way with an increase of k.

Further the variance is calculated as well.

```
#Calculating standard deviation of VC accuaracies to see the degree of variance in the obtained results.
cv_scores = np.array(cv_scores)
print(cv_scores.var())
```

```
0.0013615916955017297
```

The variance is being quite low, implies that the predictions which our model is making is not by chance and it will perform similarly on all test sets.

Dataset 2- Heart Disease Dataset

1. For the heart disease dataset, we will be splitting the data into train and test set. We will then split the train set into a new train and validate set (90%-10% respectively).

```
# we will be using the output of z-score as the input data now onwards.
#Slpiting the data into a training set and a test set.
x = df_z_scaled_hd.drop(['target'], axis=1)
y = df_z_scaled_hd['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 98)
# dividing the training set into an (90%,10%) split into train and validate sets.
x2_train, x_val, y2_train, y_val = train_test_split(x_train, y_train, test_size = 0.10, random_state = 98)
print('shape of original training set', x_train.shape, y_train.shape)
print('shape of second training set', x2_train.shape, y2_train.shape)
print('shape of validation set', x_val.shape, y_val.shape)
print('shape of testing set', x_test.shape, y_test.shape)
```

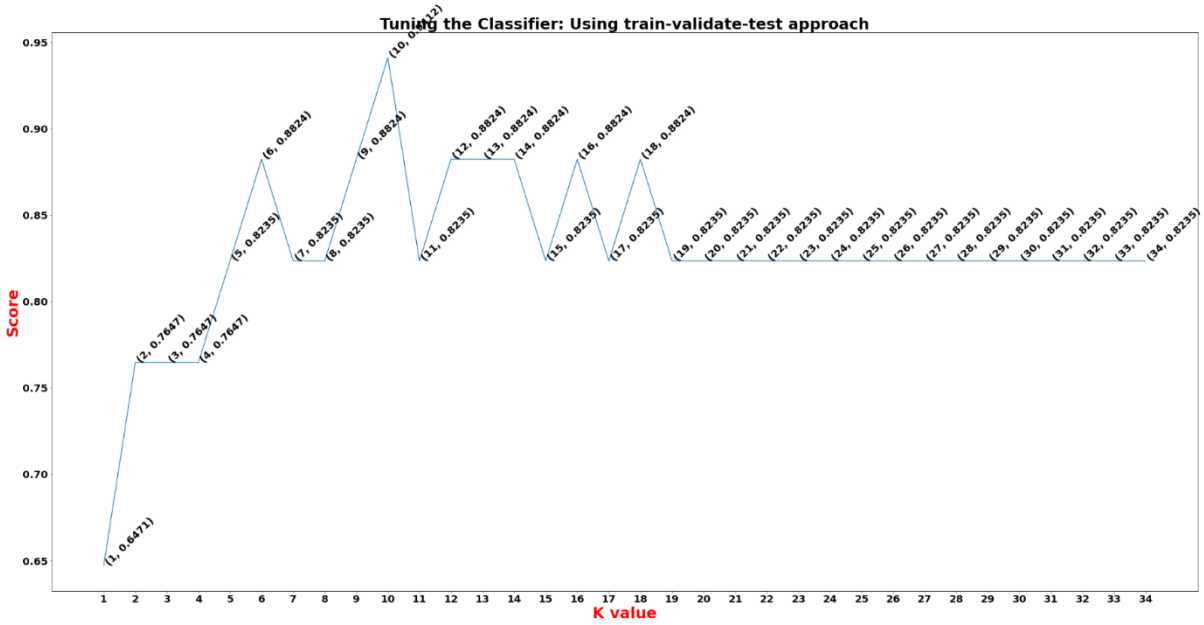
2. Now we will train our knn classifier model on this new train set and test in on the validation set. The accuracy scores for the same will be stored in an array.

```
##-------TRAIN_VALIDATE sets-------##
# Training the classifier on second training set and then testing in the Vaidation set.
knn_scores = []
for k in range(1,35):
    knn2 = KNeighborsClassifier(n_neighbors = k)
    knn2.fit(x2_train, y2_train)
    knn_scores.append(knn2.score(x_val, y_val))

knn_scores = [ '%.4f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]
print(knn_scores)
```

```
[0.6471, 0.7647, 0.7647, 0.7647, 0.8235, 0.8824, 0.8235, 0.8235, 0.8824, 0.9412, 0.8235, 0.8824, 0.8824, 0.8824, 0.8235, 0.8824,
```

3. Using these scores, we plot a graph of “accuracy vs k-values”

```
#ploting the graph
plt.rc('font', size=20)
f, ax = plt.subplots(1, 1, figsize = (40, 20))
plt.plot(range(1,35), knn_scores)
for i in range(1,35):
    plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=20, rotation=45 )
plt.xticks(np.arange(1,35,1))
plt.title("Tuning the Classifier: Using train-validate-test approach", weight='bold',fontsize='30')
plt.xlabel("K value",color='Red',weight='bold',fontsize='30')
plt.ylabel("Score",color='Red',weight='bold',fontsize='30')
plt.show()
plt.rcParams["font.weight"]= "bold"
plt.rcParams["axes.labelweight"] = "bold"
```



We can observe that the accuracy has boosted up to a whopping 94% from 81% just by using the Train-Validate set approach to train the classifier. We also observe that the maximum accuracy of 94% is found for the value of k = 10. Hence, the best value for k is set to 10.

Further it can be noted that with an increase in the value of k, the accuracy decreases significantly.

[CM6]: Fitting the model using the best found parameter

Dataset 1- Iris Dataset

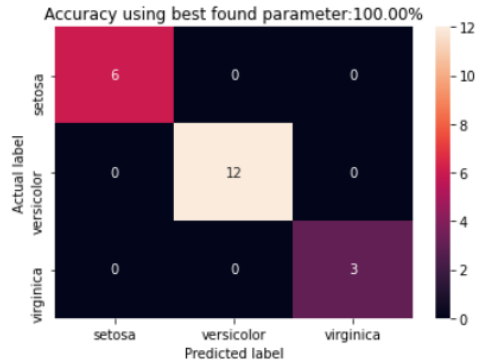
1. We now use our optimal K-value parameter (k=3) to fit the model on the training dataset. The accuracy score for the optimal parameter is noted.

```
# Using the best found parameter, fitting the model on the entire training set and predict the target on the test set.

# best found value of k
optimal_n = 3
knn2 = KNeighborsClassifier(n_neighbors = optimal_n)
# fitting the model on the entire training set
knn2.fit(x_train,y_train)
# predict the target on the test set
y_pred = knn2.predict(x_test)
# storing the accuracy and reporting it
acc = accuracy_score(y_test,y_pred)*100
print("The accuracy for optimal k = {0} is {1}".format(optimal_n,acc))
```

```
The accuracy for optimal k = 3 is 100.0
```

2. A confusion matrix is plotted for the same

```
# Creates a confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Transform to dataframe for easier plotting
cm_df = pd.DataFrame(cm, index = ['setosa','versicolor','virginica'], columns = ['setosa','versicolor','virginica'])
# Plotting the matrix
sns.heatmap(cm_df, annot=True)
plt.title('Accuracy using best found parameter:{0:.2f}%'.format(acc))
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
```

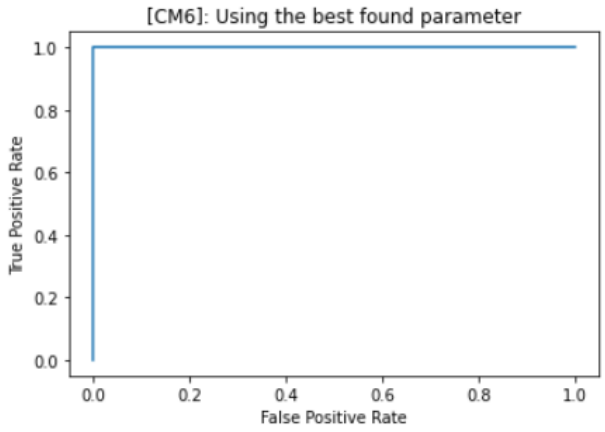


We can see from the confusion matrix that the predicted and the actual labels have the same values. Hence showing a 100% accuracy of the classification model.

3. Further we report the AUC, f score and a classification report

```
# Reporting the AUC, f-score and other classification report scores.
print("classification_report")
print(classification_report(y_test,y_pred))
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred, pos_label=2)
metrics.auc(fpr, tpr)
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('[CM6]: Using the best found parameter')
plt.show()
```

```
classification_report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      1.00      1.00        12
           2       1.00      1.00      1.00         3

    accuracy                           1.00        21
   macro avg       1.00      1.00      1.00        21
weighted avg       1.00      1.00      1.00        21
```



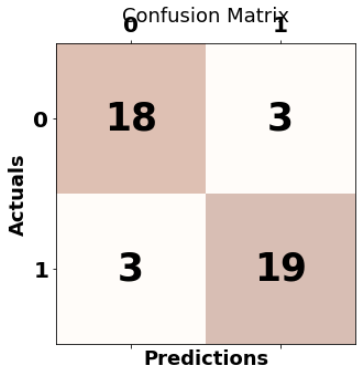
We can see that, for the provided dataset, the knn classifier showed the best accuracy for the value of k=3 which was the outcome of the tuning process performed in [CM5].

The precision, recall and f-scores for all three categories of the target variable (0,1,2) are found to be 100%

Dataset-2 Heart Disease Dataset

1. We now use our optimal K-value parameter (k=10) to fit the model on the training dataset.
2. The accuracy score is calculated for the same and a confusion matrix is plotted.

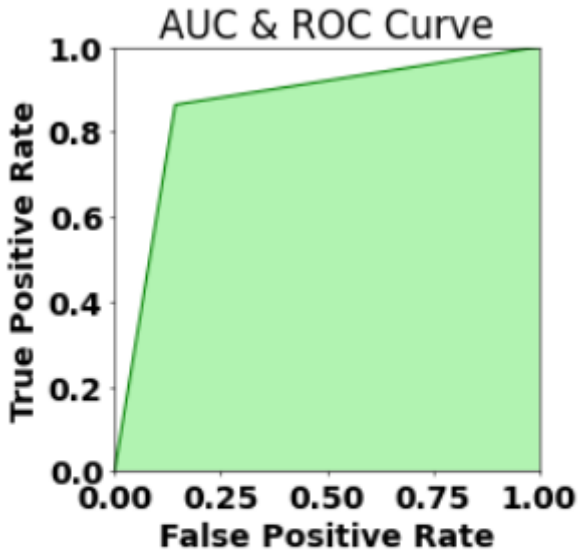
```
# fitting the same model on the original training set and predicting the target on the test set.

##-------TRAIN_TEST sets-------##
#looking at the graph it is clear that the maximum accouracy comes when k = 10.
#Therefore calculating Accuracy, Precision, Recall & F1-Score for k = 10 of our kNN classifier.
knn3 = KNeighborsClassifier(n_neighbors = 10)
knn3.fit(x_train, y_train)
y_pred = knn3.predict(x_test)
# Calculating the accuracy
acc = accuracy_score(y_test, yhat)

# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true=y_test, y_pred=y_pred)

# Print the confusion matrix using Matplotlib
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', size='xx-large')
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



It can be observed from the confusion matrix that our model does a good enough job in classifying our independent variables according to the target variable (0- no heart disease, 1- heart disease)
We find 18 cases of True Negative and 19 cases of True Positive.

3. The AUC, precision, recall and F1 scores are calculated and displayed.

```
#AUC and ROC Curve
auc = metrics.roc_auc_score(y_test, y_pred)
false_positive_rate, true_positive_rate, thresolds = metrics.roc_curve(y_test, y_pred)
plt.figure(figsize=(6, 5), dpi=50)
plt.axis('scaled')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.title("AUC & ROC Curve")
plt.plot(false_positive_rate, true_positive_rate, 'g')
plt.fill_between(false_positive_rate, true_positive_rate, facecolor='lightgreen', alpha=0.7)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.show()

#Accuracy, Precision, Recall & F1-Score
print('Precision: %.3f' % precision_score(y_test, y_pred))
print('F1 Score: %.3f' % f1_score(y_test, y_pred))
print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
print('Recall: %.3f' % recall_score(y_test, y_pred))
print('AUC: %.3f' % auc)
```




```
Precision: 0.864
F1 Score: 0.864
Accuracy: 0.860
Recall: 0.864
AUC: 0.860
```

The final reported accuracy is found to be 86.4 %

We observe that even after training and testing the tuned classifier on the best found value of k, the accuracy is lesser that the one came while tuning the classifier using train-validate test.

We're getting rather odd results, where our validation data is getting better accuracy, than our training data.

According to us the reason might lie in the data splitting. The validation set may be less noisy or less variant from the training set thus easier to predict which might be leading to higher accuracy on validation set than on training set.

Another reason may be because smaller datasets have smaller intrinsic variance thus implying that classifier properly captures patterns inside of data and train error is greater simply because the inner variance of training set is greater than validation set.

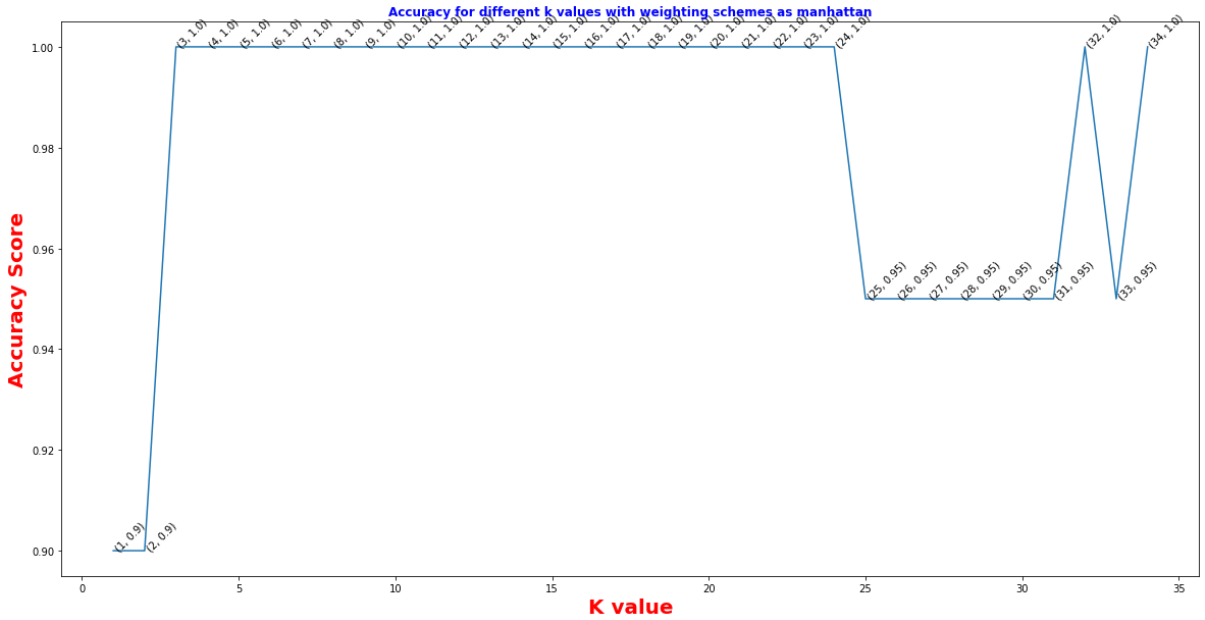
[CM7]: Improving the model by changing and experimenting on other parameters.

Dataset 1- Iris Dataset

1. The purpose now is to improve our model using performance metrics such as weighted knn and other NN algorithms (ball tree, kd tree and brute)

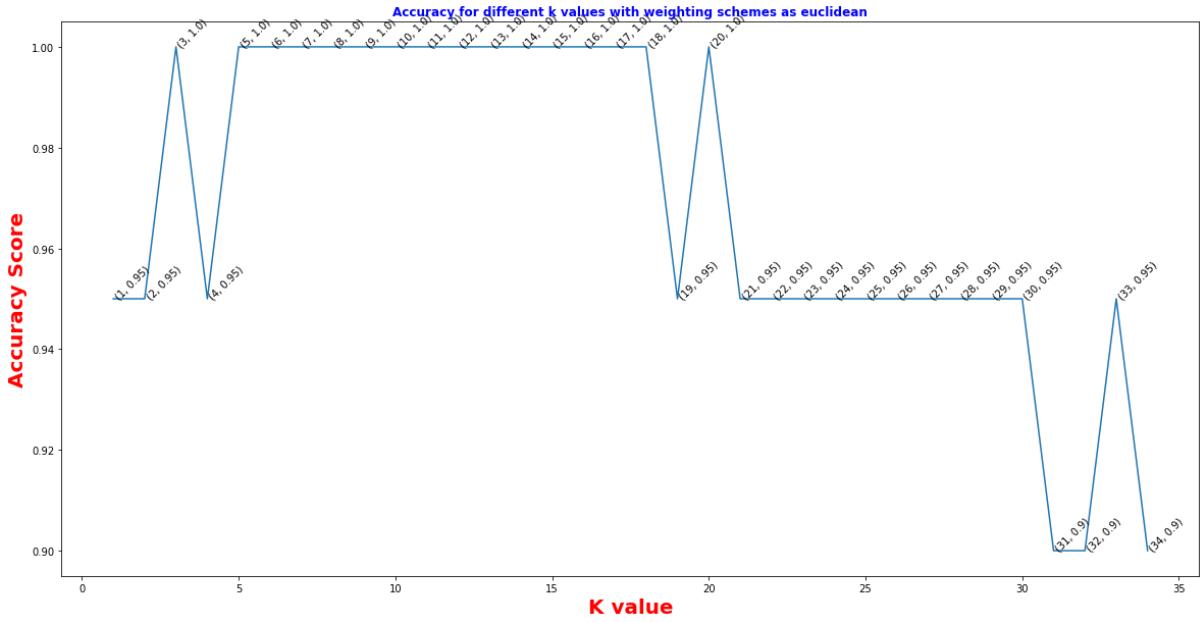
2. We proceed first with the weighted knn method, taking the weighting schemes as "Manhattan" and "Euclidian"

```
# Using Weighted KNN ( weights='distance') with weighting schemes as manhattan (metric='manhattan')

knn_scores = []
for n in list(np.arange(1,35,1)):
   # KNeighborsClassifier with default parameters.
    knn3 = KNeighborsClassifier(n_neighbors=n,weights='distance',metric='manhattan')
    knn3.fit(x_train, y_train)
    yhat = knn3.predict(x_test)
    knn_scores.append(accuracy_score(y_test, yhat))

knn_scores = [ '%.2f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]
f, ax = plt.subplots(1, 1, figsize = (20, 10))
plt.xlabel("K value",color='Red',weight='bold',fontsize='20')
plt.ylabel("Accuracy Score",color='Red',weight='bold',fontsize='20')
title = 'Accuracy for different k values with weighting schemes as manhattan'
plt.title(title,color='Blue',weight='bold')
for i in range(1,35):
    plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=10, rotation=45  )
plt.plot(neighbors, knn_scores)
```



Using the Manhattan distance metric, we observe the highest accuracy for K-values in the range of 3 to 24.

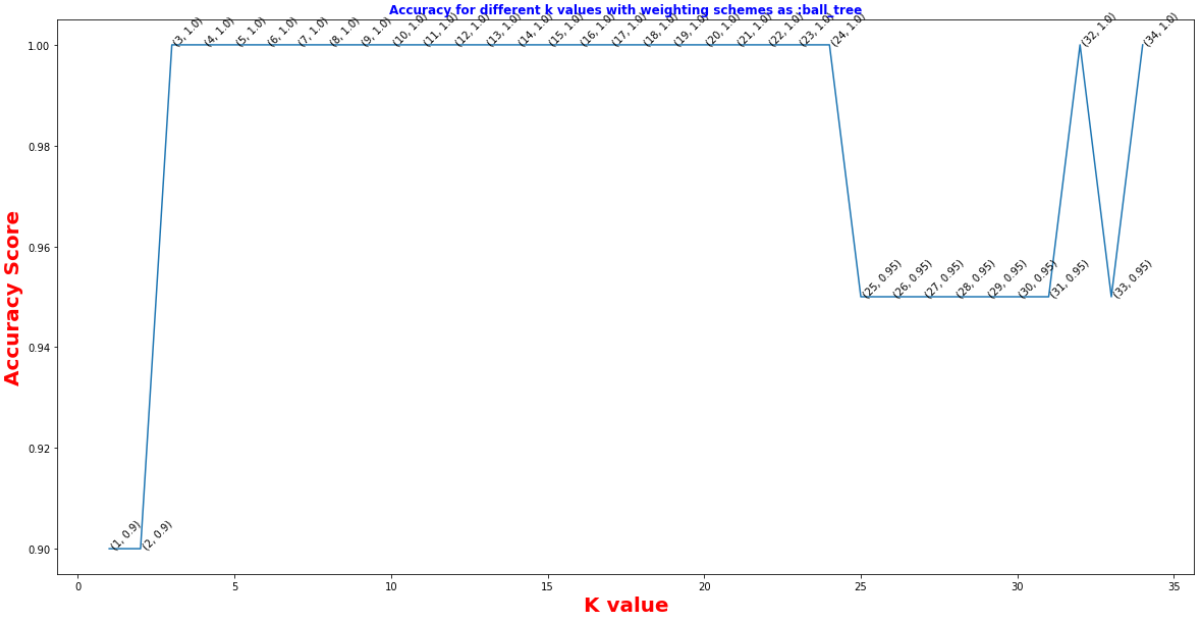
```
# Using Weighted KNN ( weights='distance') with weighting schemes as euclidean (metric='euclidean')

knn_scores = []
for n in list(np.arange(1,35,1)):
   # KNeighborsClassifier with default parameters.
    knn4 = KNeighborsClassifier(n_neighbors=n,weights='distance',metric='euclidean')
    knn4.fit(x_train, y_train)
    yhat = knn4.predict(x_test)
    knn_scores.append(accuracy_score(y_test, yhat))

knn_scores = [ '%.2f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]
f, ax = plt.subplots(1, 1, figsize = (20, 10))
plt.xlabel("K value",color='Red',weight='bold',fontsize='20')
plt.ylabel("Accuracy Score",color='Red',weight='bold',fontsize='20')
title = 'Accuracy for different k values with weighting schemes as euclidean'
plt.title(title,color='Blue',weight='bold')
for i in range(1,35):
    plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=10, rotation=45  )
plt.plot(neighbors, knn_scores)
```

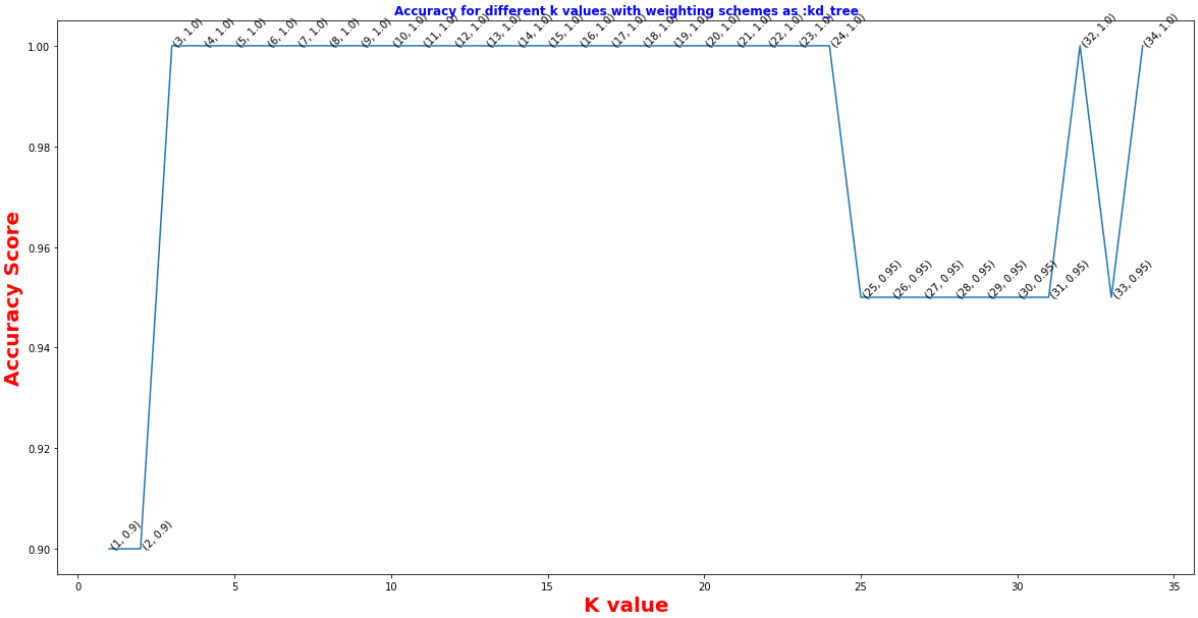


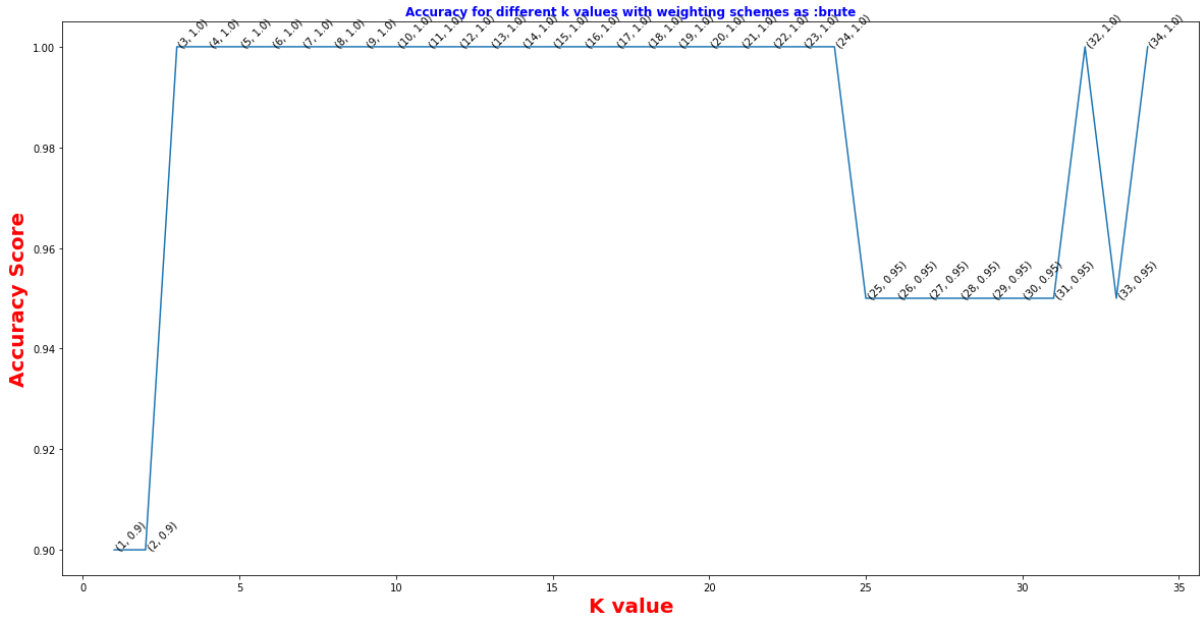
Using the Euclidian distance metric we get the highest accuracy rate for K-values ranged between 5 and 18.

3. We now implement other NN (nearest neighbours) algorithms (ball tree, kd tree and brute)

```
# Different NN Algorithms, i.e., algorithm = ('ball_tree', 'kd_tree', 'brute')
# Taking weighting schemes as manhattan (metric='manhattan').
algo = ['ball_tree', 'kd_tree', 'brute']
for a in range(len(algo)):
    knn_scores = []
    for n in list(np.arange(1,35,1)):
       # KNeighborsClassifier with default parameters.
        knn5 = KNeighborsClassifier(n_neighbors=n, algorithm=algo[a], weights='distance',metric='manhattan')
        knn5.fit(x_train, y_train)
        y_pred = knn5.predict(x_test)
        knn_scores.append(accuracy_score(y_test, y_pred))

    knn_scores = [ '%.2f' % elem for elem in knn_scores ]
    knn_scores = [float(line) for line in knn_scores]
    f, ax = plt.subplots(1, 1, figsize = (20, 10))
    plt.xlabel("K value",color='Red',weight='bold',fontsize='20')
    plt.ylabel("Accuracy Score",color='Red',weight='bold',fontsize='20')
    title = 'Accuracy for different k values with weighting schemes as :' + algo[a]
    plt.title(title,color='Blue',weight='bold')
    for i in range(1,35):
        plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=10, rotation=45  )
    plt.plot(neighbors, knn_scores)
```


Accuracy for different k values with weighting schemes as :ball_tree
Accuracy Score
K value
(1, 0.9)
(2, 0.9)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
(10, 1.0)
(11, 1.0)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 1.0)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 1.0)
(21, 1.0)
(22, 1.0)
(23, 1.0)
(24, 1.0)
(25, 0.95)
(26, 0.95)
(27, 0.95)
(28, 0.95)
(29, 0.95)
(30, 0.95)
(31, 0.95)
(32, 1.0)
(33, 0.95)
(34, 1.0)


Accuracy for different k values with weighting schemes as :kd_tree
Accuracy Score
K value
(1, 0.9)
(2, 0.9)
(3, 1.0)
(4, 1.0)
(5, 1.0)
(6, 1.0)
(7, 1.0)
(8, 1.0)
(9, 1.0)
(10, 1.0)
(11, 1.0)
(12, 1.0)
(13, 1.0)
(14, 1.0)
(15, 1.0)
(16, 1.0)
(17, 1.0)
(18, 1.0)
(19, 1.0)
(20, 1.0)
(21, 1.0)
(22, 1.0)
(23, 1.0)
(24, 1.0)
(25, 0.95)
(26, 0.95)
(27, 0.95)
(28, 0.95)
(29, 0.95)
(30, 0.95)
(31, 0.95)
(32, 1.0)
(33, 0.95)
(34, 1.0)



We observe that all three models, return the same range of K-values with the highest accuracy rate.

4. Based on the above observations, we proceed with the distance metric as “Manhattan” as it had a higher number of K-values with the highest accuracy as compared to that provided by the “Euclidian” distance metric.
   Since all three NN algorithms produced the same result, we can select any one of them for our final improved model. In this case we proceed with the “Brute” NN algorithm.

5. Finally with the aforementioned performance metrics selected, we now train our knn classifier. A confusion matrix is generated followed by the reporting of the AUC and F-scores.

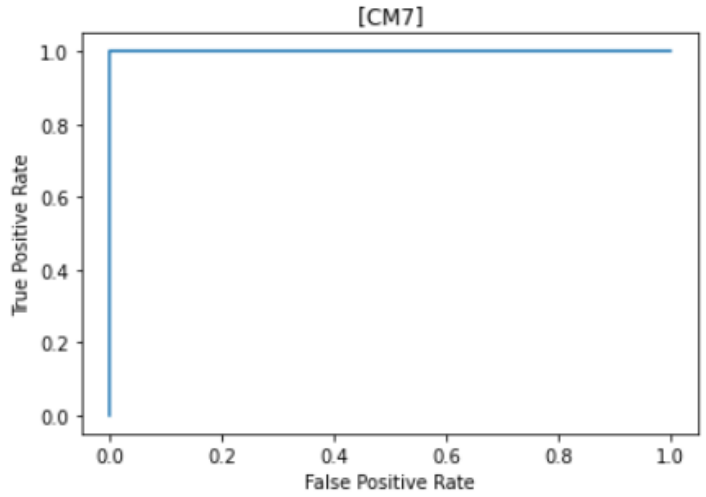
```
# Taking weighting schemes as manhattan (metric='manhattan') and algorithm = ‘brute’
# Calculating ROC and classification scores.

knn6 = KNeighborsClassifier(n_neighbors=3, algorithm = 'brute', weights='distance',metric='manhattan')
knn6.fit(x_train, y_train)
y_pred = knn6.predict(x_test)
knn_scores.append(accuracy_score(y_test, y_pred))

# Creates a confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Transform to dataframe for easier plotting
cm_df = pd.DataFrame(cm, index = ['setosa','versicolor','virginica'], columns = ['setosa','versicolor','virginica'])
# Plotting the matrix
sns.heatmap(cm_df, annot=True)
plt.title('Accuracy using algorithm = brute, weights=distance,metric=manhattan:{0:.2f}%'.format(acc))
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
```

Accuracy using algorithm = brute, weights=distance,metric=manhattan:100.00%


```
# Reporting the AUC, f-score and other classification report scores.
print("classification_report")
print(classification_report(y_test,y_pred))
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred, pos_label=2)
metrics.auc(fpr, tpr)
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('[CM7]')
plt.show()
```

classification_report

	precision	recall	f1-score	support
0	1.00	1.00	1.00	6
1	1.00	1.00	1.00	12
2	1.00	1.00	1.00	3
accuracy			1.00	21
macro avg	1.00	1.00	1.00	21
weighted avg	1.00	1.00	1.00	21



We can see from the confusion matrix that the predicted and the actual labels have the same values. Hence showing a 100% accuracy of the classification model.

The precision, recall and f-scores for all three categories of the target variable (0,1,2) are found to be 100%.

Thus in conclusion, for the Iris dataset, we don't notice any improvement in the accuracy of the model. This being the case cause of the maximum 100% accuracy being reported by our initial knn model itself.

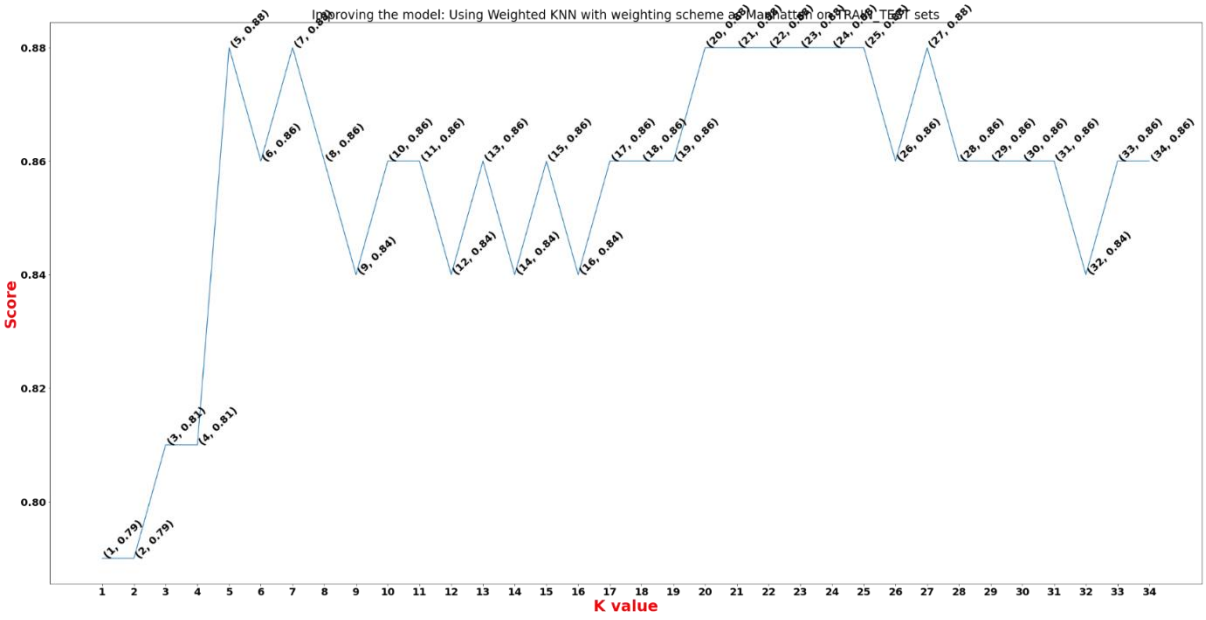
This just further proves the highly clean, normalized and tuned nature of the Iris dataset.

Dataset-2 Heart Disease Dataset

1. The purpose now is to improve our model using performance metrics such as weighted knn and other NN algorithms (ball tree, kd tree and brute)

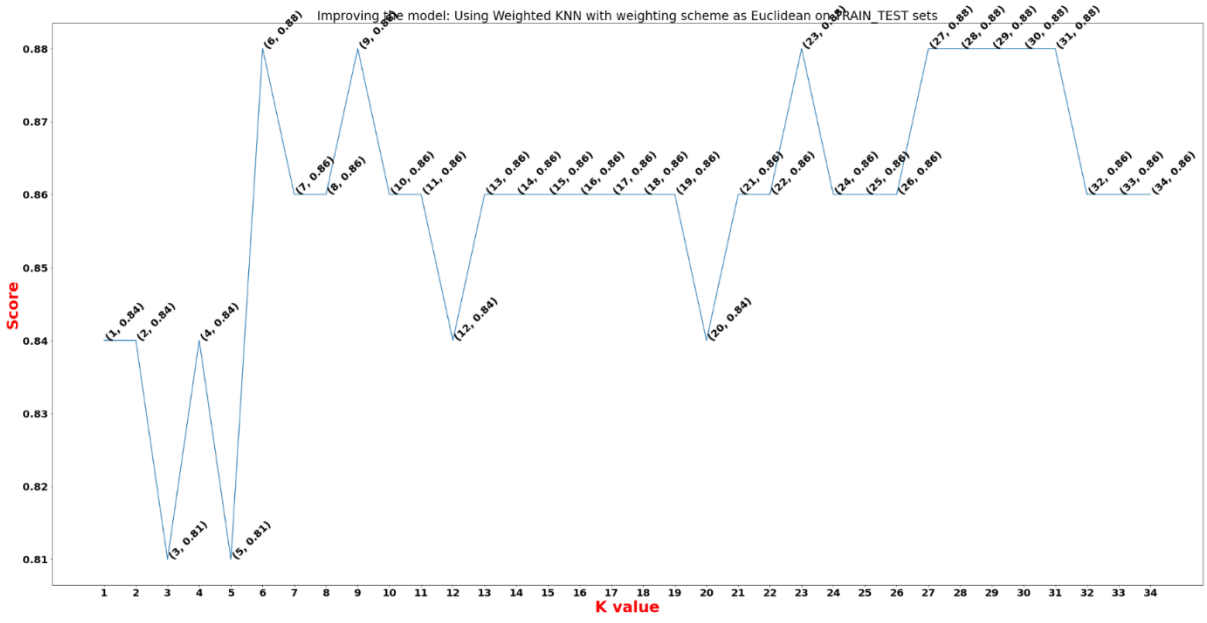
2. We proceed first with the weighted knn method, taking the weighting schemes as "Manhattan" and "Euclidian"

```
##--------Using Weighted KNN with weighting scheme as Manhattan -----------##

# Training the classifier on Training set using knn classifier with weighted knn and manhattan weighting scheme.
knn_scores = []
for k in range(1,35):
    knn4 = KNeighborsClassifier(n_neighbors = k, weights='distance', metric='manhattan')
    knn4.fit(x_train, y_train)
    y_pred =  knn4.predict(x_test)
    knn_scores.append(accuracy_score(y_test, y_pred))

knn_scores = [ '%.2f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]
plt.rc('font', size=20)
f, ax = plt.subplots(1, 1, figsize = (40, 20))
plt.plot(range(1,35), knn_scores)
for i in range(1,35):
    plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=20, rotation=45 )
plt.xticks(np.arange(1,35,1))
plt.title("Improving the model: Using Weighted KNN with weighting scheme as Manhattan on TRAIN_TEST sets")
plt.xlabel("K value",color='Red',weight='bold',fontsize='30')
plt.ylabel("Score",color='Red',weight='bold',fontsize='30')
plt.show()
plt.rcParams["font.weight"]= "bold"
plt.rcParams["axes.labelweight"] = "bold"
print(knn_scores)
```



Using the Manhattan distance metric, we observe the highest accuracy of 88% for 9 K-values.

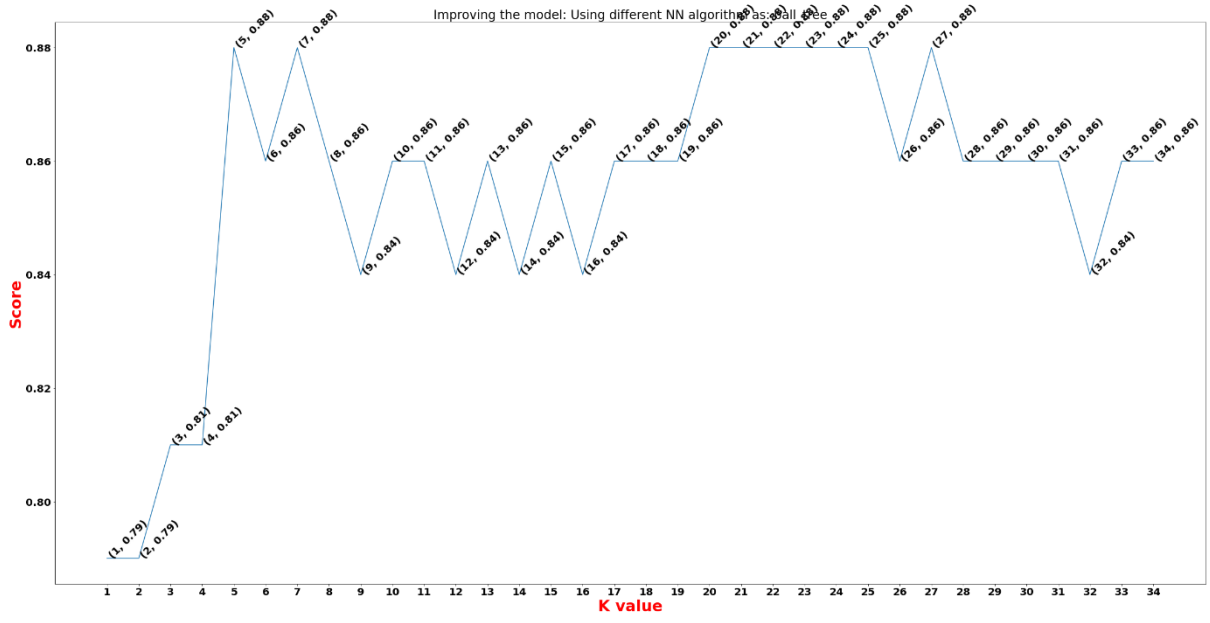
```
##--------Using Weighted KNN with weighting scheme as Euclidean -----------##

# Training the classifier on Training set using knn classifier with weighted knn and Euclidean weighting scheme.
knn_scores = []
for k in range(1,35):
    knn5 = KNeighborsClassifier(n_neighbors = k, weights='distance', metric='euclidean')
    knn5.fit(x_train, y_train)
    y_pred =  knn5.predict(x_test)
    knn_scores.append(accuracy_score(y_test, y_pred))

knn_scores = [ '%.2f' % elem for elem in knn_scores ]
knn_scores = [float(line) for line in knn_scores]
plt.rc('font', size=20)
f, ax = plt.subplots(1, 1, figsize = (40, 20))
plt.plot(range(1,35), knn_scores)
for i in range(1,35):
    plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=20, rotation=45 )
plt.xticks(np.arange(1,35,1))
plt.title("Improving the model: Using Weighted KNN with weighting scheme as Euclidean on TRAIN_TEST sets")
plt.xlabel("K value",color='Red',weight='bold',fontsize='30')
plt.ylabel("Score",color='Red',weight='bold',fontsize='30')
plt.show()
plt.rcParams["font.weight"]= "bold"
plt.rcParams["axes.labelweight"] = "bold"
print(knn_scores)
```

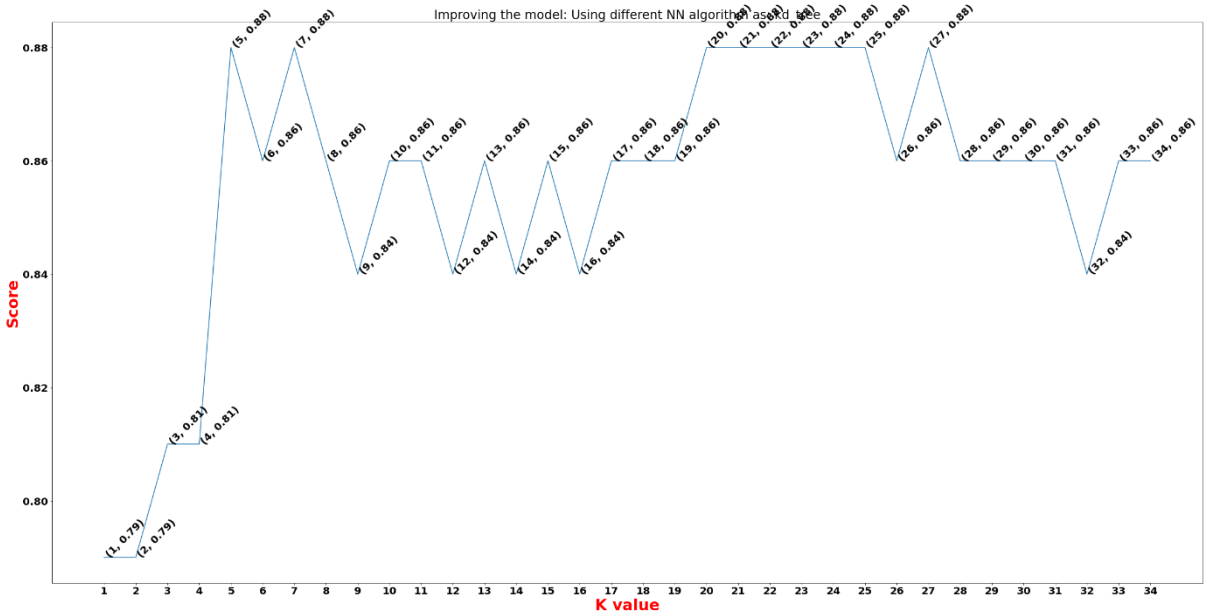


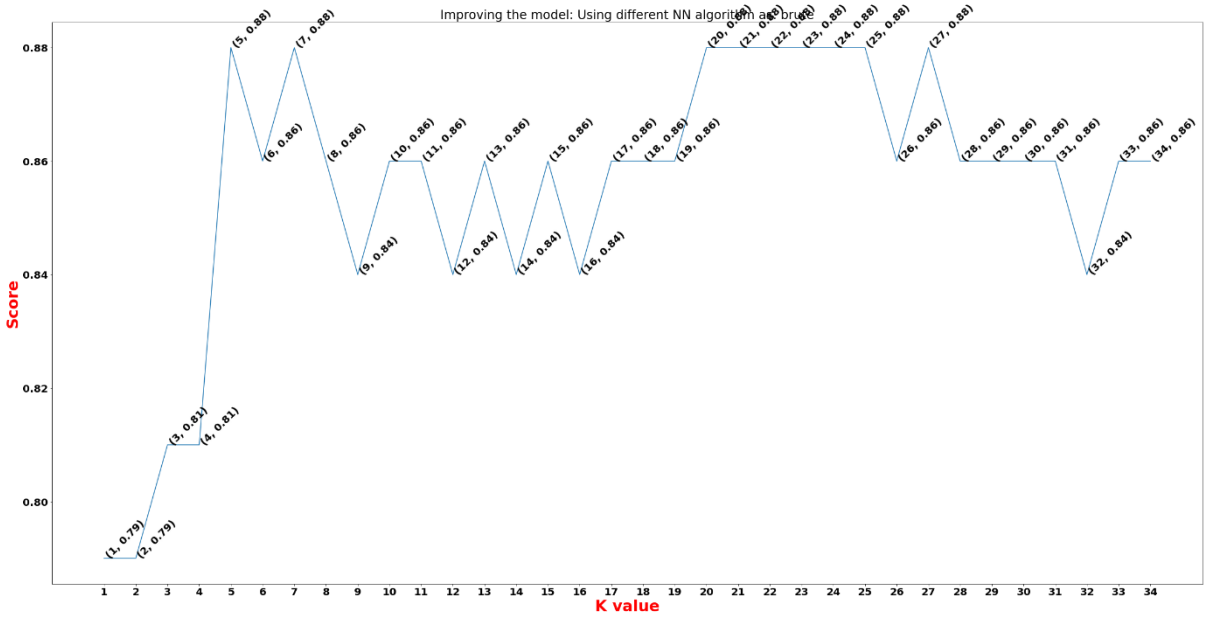
Using the “Euclidian” distance metric, we get the highest accuracy of 88% for 8 K-values.

3. We now implement other NN (nearest neighbours) algorithms (ball tree, kd tree and brute)

```
##--------Using different NN algorithms and weighting scheme as Manhattan-----------##
# storing names of all the algorithms in algo list.
algo = ['ball_tree', 'kd_tree', 'brute']
for a in range(len(algo)):
    knn_scores = []
    for k in range(1,35):
        knn6 = KNeighborsClassifier(n_neighbors = k, algorithm=algo[a], weights='distance', metric='manhattan')
        knn6.fit(x_train, y_train)
        y_pred =  knn6.predict(x_test)
        knn_scores.append(accuracy_score(y_test, y_pred))

    knn_scores = [ '%.2f' % elem for elem in knn_scores ]
    knn_scores = [float(line) for line in knn_scores]
    plt.rc('font', size=20)
    f, ax = plt.subplots(1, 1, figsize = (40, 20))
    plt.plot(range(1,35), knn_scores)
    for i in range(1,35):
        plt.text(i, knn_scores[i-1], (i, knn_scores[i-1]), fontsize=20, rotation=45 )
    plt.xticks(np.arange(1,35,1))
    title = "Improving the model: Using different NN algorithm as: " + algo[a]
    plt.title(title)
    plt.xlabel("K value",color='Red',weight='bold',fontsize='30')
    plt.ylabel("Score",color='Red',weight='bold',fontsize='30')
    plt.show()
    plt.rcParams["font.weight"]= "bold"
    plt.rcParams["axes.labelweight"] = "bold"
    print(knn_scores)
```





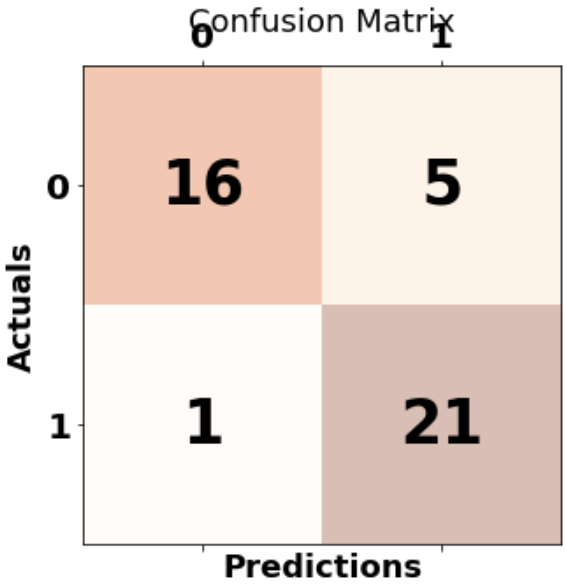


We observe that all three NN algorithms, return the same range of K-values with the highest accuracy rate (88%)

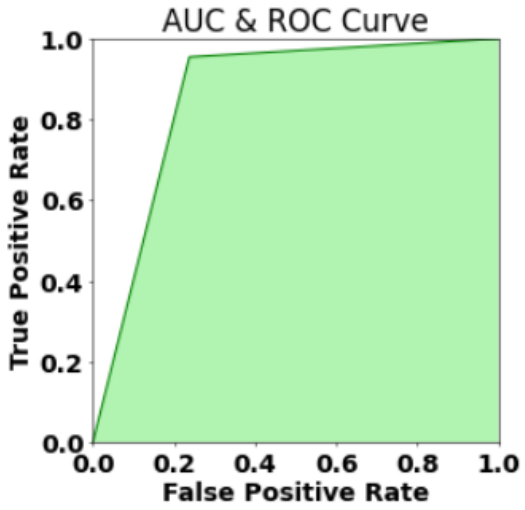
4. Based on the above observations, we proceed with the distance metric as “Manhattan” as it had a higher number of K-values with the highest accuracy as compared to that provided by the “Euclidian” distance metric.
   Since all three NN algorithms produced the same result, we can select any one of them for our final improved model. In this case we proceed with the “Brute” NN algorithm.

5. Finally with the aforementioned performance metrics selected, we now train our knn classifier. A confusion matrix is generated followed by the reporting of the AUC and F-scores.

```
# Choosing algorithm = brute for calculating the AUC, f-score and other.
# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true=y_test, y_pred=y_pred)

# Print the confusion matrix using Matplotlib
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', size='xx-large')
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



```
#AUC and ROC Curve
auc = metrics.roc_auc_score(y_test, y_pred)
false_positive_rate, true_positive_rate, thresolds = metrics.roc_curve(y_test, y_pred)
plt.figure(figsize=(6, 6), dpi=50)
plt.axis('scaled')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.title("AUC & ROC Curve")
plt.plot(false_positive_rate, true_positive_rate, 'g')
plt.fill_between(false_positive_rate, true_positive_rate, facecolor='lightgreen', alpha=0.7)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.show()

#Accuracy, Precision, Recall & F1-Score
print('Precision: %.3f' % precision_score(y_test, y_pred))
print('F1 Score: %.3f' % f1_score(y_test, y_pred))
print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
print('Recall: %.3f' % recall_score(y_test, y_pred))
print('AUC: %.3f' % auc)
```




```
Precision: 0.808
F1 Score: 0.875
Accuracy: 0.860
Recall: 0.955
AUC: 0.858
```

It can be observed from the confusion matrix that our model does a good enough job in classifying our independent variables according to the target variable (0- no heart disease, 1- heart disease)
We find 16 cases of True Negative and 21 cases of True Positive.
And the above observations indicate a slight improvement in our model performance as well.

The precision of our model reduced significantly indicating a drop in the number of relevant results our model produced. However there was a significant improvement in our recall value

implying that even though fewer relevant results were returned by the model (owing to the low precision), most of the relevant results were returned.

The AUC and accuracy values show no significant change.

In conclusion, we observe a marginal improvement in our model performance owing to the high recall value.