EXPERIMENT 1

K-NN SUPERVISED LEARNING

TITLE: *THE K-NEAREST NEIGHBOUR (K-NN) REGRESSION, CLASSIFICATION, AND OUTLIER DETECTION USING PYTHON WITH SCIKIT-LEARN LIBRARY*

INDEX

NOTE: The report form must be filled up based on your understanding after completion of the experiment and have to be submitted after showing the results.

1.1 OBJECTIVE(S):

- To understand and apply K-NN for regression analysis
- To understand and apply K-NN for classification
- To understand and apply K-NN for outlier detection

1.2 EQUIPMENT/SETUP:

- A Computer (PC) with Internet connection
- Operating system
- Any browser to use Google Colab (*https://colab.research.google.com*)

1.3 BACKGROUND:

The K-nearest Neighbors (KNN) algorithm works on previously labeled data and is a type of **supervised machine learning algorithm** used for **classification**, **regression** as well as **outlier detection**. It is called a lazy learning algorithm as it does not have a specialized training phase. It uses all of the data while classifying or regressing a new data point or instance. K-NN is a non-parametric learning algorithm which does not assume anything about the underlying data.

If the dataset contains **continuous** numbers or the estimated values are continuous, K-NN Regression can be used. If the data are **categorized** (discrete), K-NN Classification can be used. It is also possible to estimate or determine a very **different data** which is far away (high distance) from most of the data points and not suitable to fit into any value or category. In these cases, K-NN can be used for outlier detection.

K-NN uses just a single part of the data for deciding the value (regression) or class (classification). As it does not look at all the points of data, it a ***lazy learning approach***.

K-NN does not assume anything about the underlying characteristics of data, such as uniformly distributed, linearly separable, etc. Thus K-NN is named as ***non-parametric learning algorithm***. This is an extremely useful feature since most of the real-world data doesn't really follow any theoretical assumption.

1.4 K-NN STRATEGY:

The algorithm first calculates the *distance* of a new data point to all other training data points. After calculating the distance, K-NN selects a number of nearest data points like 2, 3, 5, 10, etc. This number of points is the **K** in K-NN. The process is presented in Figure 1.1. According to the figure, the **black dot (82, 1.78)** is the new data point (labeled as **'New?'** in Figure 1.1) from which the **dotted circle** is indicating the closest area with 5 nearest neighbors.

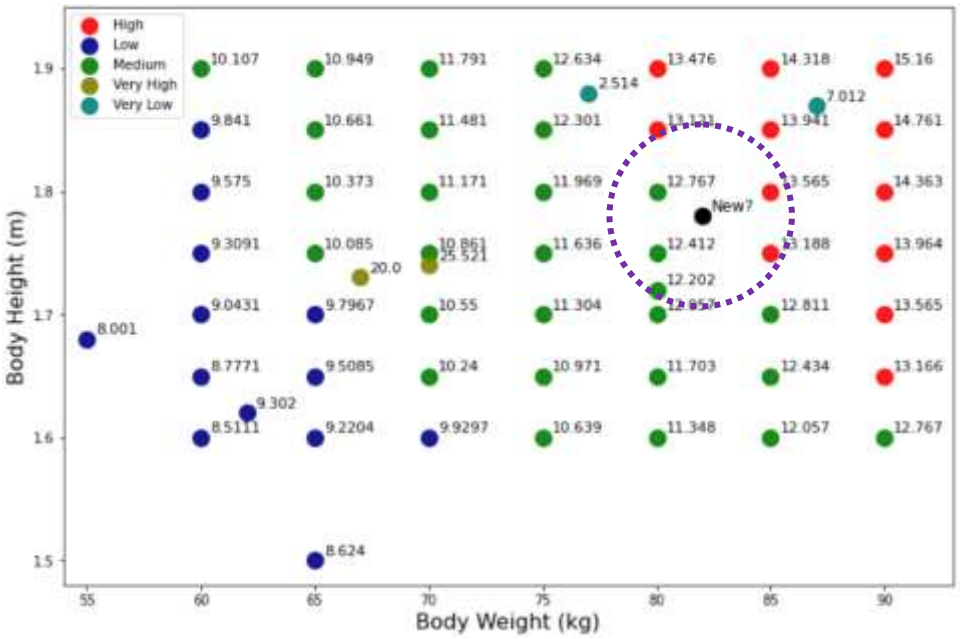

Figure 1.1

Conceptual block diagram of Single-server queue system

Regression: K-NN will calculate the **average weighted sum** of the K-nearest points for the prediction. The process is presented in Equation (1.1) where weight value is considered as, $w = 1$, and K value is considered as, $K = 5$.

$$New\ value = \frac{(11.767\times w)+(11.412\times w)+(11.202\times w)+(13.565\times w)+(13.188\times w)}{K} = 11.827 \qquad (1.1)$$

Classification: The new data point will be assigned to the class to which the **majority of the selected K-nearest points** belong. According to Figure 1.1, there are two neighbor classes, "**Green**" (Medium) and "**Red**" (High) where the majority class is green (green neighbors are 3) and minority class is red (red neighbors are 2). So, the new data point belongs to "**Green**" (Medium) class.

1.5 PROBLEM DESCRIPTION:

Knee joint rehabilitation system is a man-machine cooperative robot that assists in moving human leg and knee joint to mimic various motor function recovery training exercises within the workspace of the limb. Robot assistive system enables a wide range of functional adaptation of various limbs and joints of post-stroke patients. Robotic tools also provide opportunities in observing and measuring functional improvements of particular muscles as well as limbs movements and joints Range of Motion (RoM).

Such kind of assistive system needs to determine the required torque (rotational force) of the targeted joint (knee joint). To predict the required torque, human body parameters (for example, body weight and body height) need to be measured. Figure 1.2 presents a human knee model with RoM while body is in sitting position. A sample data set is shown in Figure 1.3 where body weight and body height with required torque information are provided. Now, it is important to design an Artificial Intelligence (AI) based model to train the system through Machine

Learning (ML) approach so that machine can predict required torque based on the body weight and body height information of a new patient.

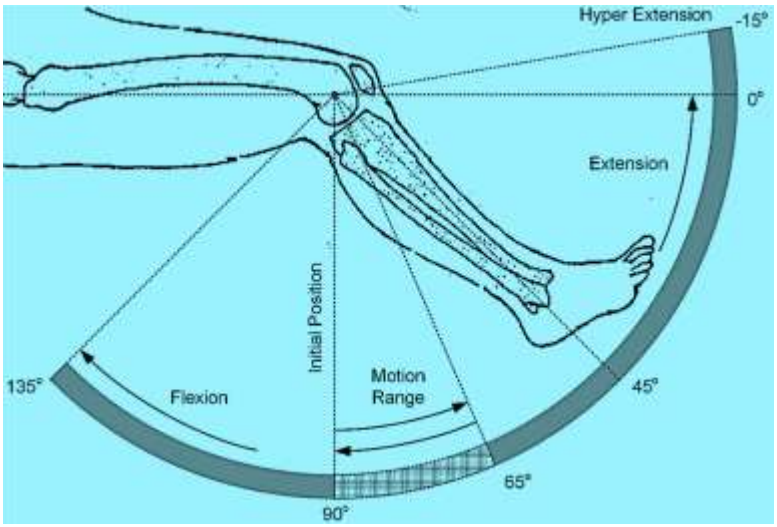

Figure 1.2
Knee joint motion range

1.6 PROCEDURE:

- Get the '**Knee-Torque-ZDataSet.csv**' and upload in your Google Drive
- Open Google Colab (*https://colab.research.google.com/*)
- Create/rename your project as '*KneeTorque_KNN.ipynb*'
- Follow the steps of next section (read and understand) and execute all the codes

1.7 LAB PRACTICE/EXPERIMENT:

1.7.1 Getting the dataset

Download the dataset '**Knee-Torque-ZDataSet.csv**' and upload in your Google Drive. Please open the CSV file to see and understand the data.

1.7.2 Understanding the dataset

The '**Knee-Torque-ZDataSet.csv**' contains six features of left leg of human body. The features are,

Body Weight (kg): It contains the weight of humans' body in kilogram

Body Height (m): It contains the height of corresponding human body in meter

Weight-Height Ratio (R): It contains the body weight-height ratio of corresponding human body

Internal Moment (M): It contains the Internal Moment of the left leg of the corresponding human

Required Torque (N-m): It contains the required torque of the Knee joint (left knee) of the corresponding human subject. Torques are in Newton-meter (N-m) unit.

Torque Category: It contains the category (classifications) of the required torque of the corresponding body. There are **Five** categories, **Very Low**, **Low**, **Medium**, **High**, **Very High**.

The CSV file contains 57 records (small dataset) in total. Table 1.1 shows the data of the CSV file with some missing values.

Table 1.1

*The dataset '**Knee-Torque-ZDataSet.csv**'*

Body Weight (kg)	Body Height (m)	Weight-Height Ratio (R)	Internal Moment (M)	Required Torque (N-m)	Torque Category
60	1.6	37.5	0.86848	8.5111	Low
60	1.65	36.364	0.89562	8.7771	Low
60	1.7	35.294	0.92276	9.0431	Low
60	1.75	34.286	0.9499	9.3091	Low
60	1.8	33.333	0.97704	9.575	Low
60	1.85	31.432	1.0042	9.841	Low
60	1.9	31.579	1.0313	10.107	Medium
65	1.6	40.625	0.94086	9.2204	Low
65	1.65	39.394	0.97026	9.5085	Low
65	1.7	38.235	0.99966	9.7967	Low
65	1.75	37.143	1.0291	10.085	Medium
65	1.8	36.111	1.0585	10.373	Medium
65	1.85	35.135	1.0879	10.661	Medium
65	1.9	34.211	1.1173	10.949	Medium
70	1.6	43.75	1.0132	9.9297	Low
70	1.65	41.424	1.0449	10.24	Medium
70	1.7	41.176	1.0766	10.55	Medium
70	1.75	40	1.1082	10.861	Medium
70	1.8	38.889	1.1399	11.171	Medium
70	1.85	37.838	1.1715	11.481	Medium
70	1.9	36.842	1.2032	11.791	Medium
75	1.6	46.875	1.0856	10.639	Medium
75	1.65	45.455	1.1195	10.971	Medium
75	1.7	44.118	1.1535	11.304	Medium
75	1.75	41.857	1.1874	11.636	Medium
75	1.8	41.667	1.2213	11.969	Medium
75	1.85	40.541	1.2552	11.301	Medium
75	1.9	39.474	1.2892	11.634	Medium
80	1.6	50	1.158	11.348	Medium
80	1.65	48.485	1.1942	11.703	Medium
80	1.7	47.059	1.2304	11.057	Medium
80	1.75	45.714	1.2665	11.412	Medium
80	1.8	44.444	1.3027	11.767	Medium
80	1.85	43.243	1.3389	13.121	High
80	1.9	41.105	1.3751	13.476	High
85	1.6	53.125	1.2304	11.057	Medium
85	1.65	51.515	1.2688	11.434	Medium
85	1.7	50	1.3072	11.811	Medium
85	1.75	48.571	1.3457	13.188	High
85	1.8	47.222	1.3841	13.565	High
85	1.85	45.946	1.4226	13.941	High
85	1.9	44.737	1.461	14.318	High
90	1.6	56.25	1.3027	11.767	Medium
90	1.65	54.545	1.3434	13.166	High
90	1.7	51.941	1.3841	13.565	High
90	1.75	51.429	1.4249	13.964	High
90	1.8	50	1.4656	14.363	High
90	1.85	48.649	1.5063	14.761	High
90	1.9	47.368	1.547	15.16	High
55	1.68			8.001	Low
80	1.72			11.202	Medium
70	1.74			25.521	Very High
62	1.62			9.302	Low
65	1.5			8.624	Low
87	1.87			7.012	Very Low
77	1.88			1.514	Very Low
67	1.73			20	Very High

For this experiment only two features, 'Body Weight (kg)' and 'Body Height (m)' are considered for training and predict. The two features, 'Required Torque (N-m)' and 'Torque Category' are the **targets** where 'Required Torque (N-m)' is for K-NN **regression** and 'Torque Category' is for **classification**.

1.7.3 Import Libraries in Google Colab

Write the codes shown in Figure 1.3 to import preliminary libraries and execute. More libraries are required to conduct the experiment which will be imported latter.

```
# Importing the library files...
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Figure 1.3

Importing some preliminary libraries into colab

1.7.4 Fetching data from google drive

Write the following code shown in Figure 1.4 to fetch the dataset from google drive and execute. Figure 1.5 shows some data from last containing NaN values.

```
# Fetching the Dataset for training...
path = "/content/drive/MyDrive/ZDataSet/Knee-Torque-ZDataSet.csv"
df = pd.read_csv(path)
df.tail()
```

Figure 1.4

Fetching dataset for training and test

	Body Weight (kg)	Body Height (m)	Weight-Height Ratio (R)	Internal Moment (M)	Required Torque (N-m)	Torque Category
52	62	1.62	NaN	NaN	9.302	Low
53	65	1.50	NaN	NaN	8.624	Low
54	87	1.87	NaN	NaN	7.012	Very Low
55	77	1.88	NaN	NaN	2.514	Very Low
56	67	1.73	NaN	NaN	20.000	Very High

Figure 1.5

Some data from the last of Data Frame (df) containing NaN values

1.7.5 Plotting the data for understanding

Write the following code shown in Figure 1.6 to plot (scatter plot) data where 'Body Weight (kg)' in **x-axis** and 'Body Height (m)' in **y-axis**. This will help to understand the distribution of the data. Each point of the scatter plot shows the corresponding or required torque of left knee joint of human subjects of corresponding body weight and height. The scatter plot is presented in Figure 1.7.

It is possible and sometimes imperative to plot data in different fashion to understand the various distributions and patterns of data. For example, Body weight vs. Torque, Body height vs. Torque, etc.

```
# Plotting the Body Weight-Height graph...
# Help link: https://matplotlib.org/stable/tutorials/introductory/pyplot.html
#            https://www.w3schools.com/python/matplotlib_plotting.asp
# https://www.python-graph-gallery.com/custom-legend-with-matplotlib
plt.figure(figsize=(12, 8))
plt.scatter(df["Body Weight (kg)"], df["Body Height (m)"], s=150, c='c')
plt.axis([54, 93, 1.48, 1.92]) # plt.axis([x-min, x-max, y-min, y-max])

plt.xlabel("Body Weight (kg)")
plt.ylabel("Body Height (m)")

w = df["Body Weight (kg)"].to_numpy()
h = df["Body Height (m)"].to_numpy()
i=0
# df.to_numpy() to convert the DataFrame to a NumPy array...
for v in df["Required Torque (N-m)"].to_numpy(): #df.iloc[:, 4].values:
  #plt.text(df["Body Weight (kg)"], df["Body Height (m)"], "1")
  plt.text(w[i]+.4, h[i]+.004, v)
  i=i+1
```

Figure 1.6

Plotting 'Body Weight (kg)' and 'Body Height (m)' scatter graph with corresponding torque values

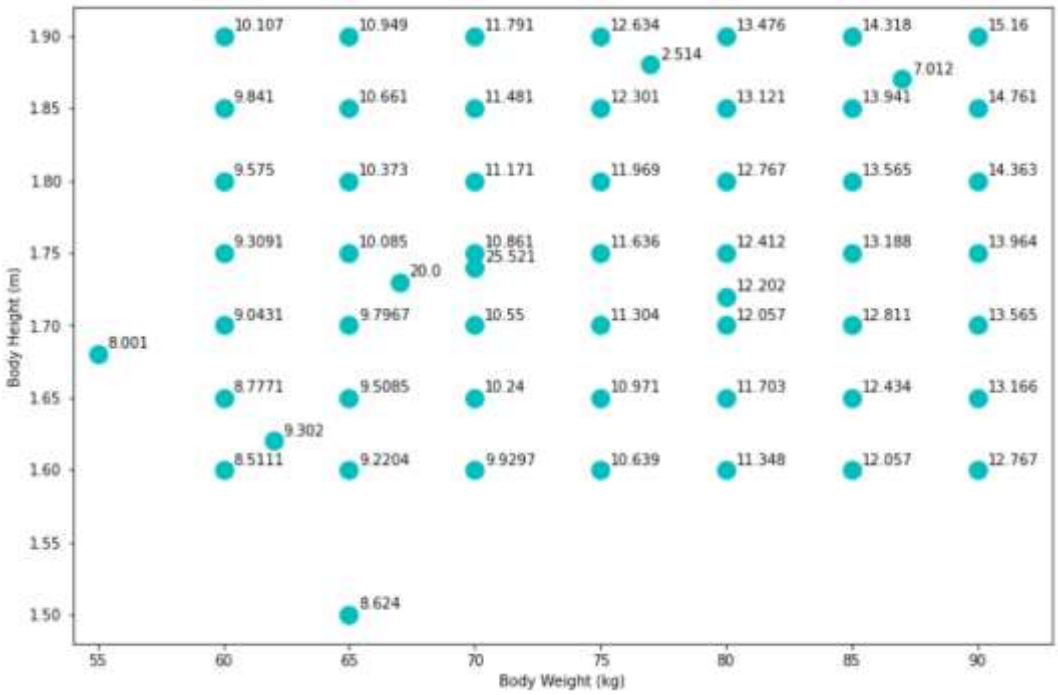

Figure 1.7

Scatter plot of 'Body Weight (kg)' and 'Body Height (m)' with torque values

From Figure 1.7 it can be observed that the torques (rotational forces) has some linear behavior. Which mean, if Body Weight increases, the torque increases; again if Body Height increases, torque values also increase. This behavior is reflected because of the computer generated data (the data set is generated based on a general equation). For the real data, the behavior would be different compared to the shown graph. From the graph (Figure 1.7), some abnormal values also can be observed, i.e. the data points (70, 1.74), (87, 1.87), (77, 1.88), (67, 1.73), etc. From the graph, the classes (categories) can not be observed.

1.7.6 Plotting the data in different fashion

Write the code shown in Figure 1.8 to plot the data in different fashion.

```
T_CATEGORY = df['Torque Category'].values
T_CATEGORY_ = np.unique(T_CATEGORY)
COLORS = ["#FF0000", "#00008b", "#008000", "#808000", "#008080"]

fig, ax = plt.subplots(figsize=(12,8))
plt.axis([54, 93, 1.48, 1.95]) # plt.axis([x-min, x-max, y-min, y-max])
for catagory, color in zip(T_CATEGORY_, COLORS):
    idxs = np.where(T_CATEGORY == catagory)
    # No legend will be generated if we do not pass label=catagory...
    ax.scatter(
        w[idxs], h[idxs], label=catagory, s=150, color=color, alpha=0.9
    )
ax.legend()

font = {'family': 'sans serif', 'color':  'black',
        'weight': 'normal', 'style' : 'normal', 'size': 11,
        }
i=0
for v in df["Required Torque (N-m)"].to_numpy(): #df.iloc[:, 4].values:
  plt.text(w[i]+.4, h[i]+.004, v, fontdict=font)
  i=i+1

# https://matplotlib.org/stable/tutorials/text/text_props.html
font = {'family': 'sans serif', 'color':  'black',
        'weight': 'normal', 'style' : 'normal', 'size': 16,
        }
ax.set_xlabel("Body Weight (kg)", fontdict=font, fontsize=16)
ax.set_ylabel("Body Height (m)", fontdict=font, fontsize=16)

plt.scatter(82, 1.78, s=150, c='#000000')
plt.text(82+.4, 1.78+.004, "New?", fontname='sans serif', fontsize=12)
```

Figure 1.8

Python code to plot the data with legend and categories with different colors

Figure 1.9 presents the same plot with various classes (categories) indicated by five different colors. High torque ($13\ N-m \leq \tau < 15N-m$, Red in color), Medium torque ($10N-m \leq \tau < 13\ N-m$, Green in color), Low torque ($8\ N-m \leq \tau < 10N-m$, Blue in color), Very High torque ($15\ N-m \leq \tau$, Brown in color), and Very Low torque ($\tau < 8\ N-m$, Light blue in color).

A new data point (82, 1.78; Black in color) is also plotted in the graph which torque needs to be predicted based on K-NN Regression and needs to be classified based on K-NN Classification methods. In this experiment, the test data set will be used to compare the predicted and test data set values to find and compare the errors. There are very few records for Very High and Very Low classes (categories) in the used data set. Even those records are not appropriate. So, these records can be discarded while applying the K-NN methods.

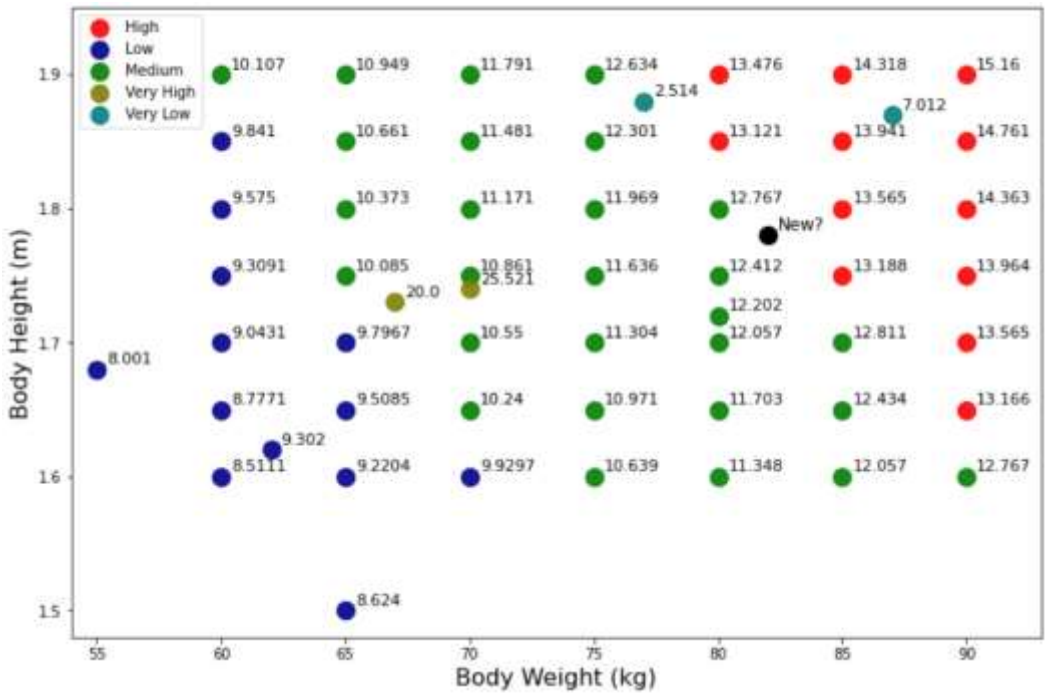

Figure 1.9

Scatter plot with five different classes and a new test point

1.7.7 Data Pre-processing

1.7.7.1 Checking the NULL values

Write the code shown in Figure 1.10 to check which feature has NULL values and how many.

```
# df # df will sohow all the records with values...
# df.isnull() # This will soow all the records with Ture or False (bool) values...
df.isnull().sum() # [df.isna().sum()] Shows the features with NULL values count...
```

```
Body Weight (kg)          0
Body Height (m)           0
Weight-Height Ratio (R)   8
Internal Moment (M)       8
Required Torque (N-m)     0
Torque Category           0
dtype: int64
```

Figure 1.10

Code to check NULL values in a data-frame with corresponding outputs

1.7.7.2 Dropping the NULL value records

Write the code shown in Figure 1.11 to drop the NULL value records. The Null values can be replaced by the average values, or mean values, or any values base on the existing values of the corresponding feature columns. In this experiment, the Null value records are discarded which also eliminates the Very High and Very Low classes (categories) presenting some anomalies or abnormal data. After dropping the Null records, the total rows become 49, as presented in Figure 1.11. At the same time, the ***df*** (Data Frame) is assigned to ***df1*** without destroying the original data.

```
# ==================================
# Dropping the NULL records...
# ==================================
# df1 = df
# avg1 = df['Weight-Height Ratio (R)'].sum()/df['Weight-Height Ratio (R)'].count()
# avg2 = df['Internal Moment (M)'].sum()/df['Internal Moment (M)'].count()

#### df.dropna() or df.fillna(100) or df.replace(np.nan, 0) or df.interpolate() ...
# df1['Weight-Height Ratio (R)'] = df['Weight-Height Ratio (R)'].replace(np.nan, avg1)
# df1['Internal Moment (M)'] = df['Internal Moment (M)'].replace(np.nan, avg2)

df1 = df.dropna()

df1.tail(10)
```

Figure 1.11

Dropping the Null value records

	Body Weight (kg)	Body Height (m)	Weight-Height Ratio (R)	Internal Moment (M)	Required Torque (N-m)	Torque Category
39	85	1.80	47.222	1.3841	13.565	High
40	85	1.85	45.946	1.4226	13.941	High
41	85	1.90	44.737	1.4610	14.318	High
42	90	1.60	56.250	1.3027	12.767	Medium
43	90	1.65	54.545	1.3434	13.166	High
44	90	1.70	52.941	1.3841	13.565	High
45	90	1.75	51.429	1.4249	13.964	High
46	90	1.80	50.000	1.4656	14.363	High
47	90	1.85	48.649	1.5063	14.761	High
48	90	1.90	47.368	1.5470	15.160	High

Figure 1.12

Last few records after eliminating the Null value records

1.7.7.3 Determining dependent and independent features

Write the code shown in Figure 1.13 to separate the dependent and independent features. The dependent feature is the target which needs to be predicted. The independent features contain the values based on which the target value have to be predicted. The variable '*y*' is holding the target feature column (dependent feature). The variable '*X*' is holding the independent features. The '*X.describe()*' method shows the data descriptions of the independent features.

```
y = df1['Required Torque (N-m)']
X = df1.drop(['Weight-Height Ratio (R)', 'Internal Moment (M)',
              'Required Torque (N-m)', 'Torque Category'], axis=1)
X.describe()
```

Figure 1.13

Eliminating the unnecessary features

Figure 1.14 shows the descriptions of the independent features of the data frame '*X*'. Here, the mean value of 'Body Weight (kg)' is approximately 75.00 and the mean value of 'Body Height (m)' is about 1.75, making it ≈ 41.8 times larger than 'Body Height (m)'. Other features also have differences in mean and standard deviation.

For 'Body Weight (kg)' std is approximately 10.103, for 'Body Height (m)', std is 0.101.

Distance-based algorithms (K-NN) suffer greatly from data that is not on the same scale. The scale of the points may distort the real distance between values. To perform Feature Scaling, ***Scikit-Learn's StandardScaler class*** will be applied later. If the scaling is applied before train-test split, the calculation would include test data, effectively leaking test data information into the rest of the pipeline. This sort of data leakage is unfortunately commonly skipped, resulting in irreproducible or illusory findings.

	Body Weight (kg)	Body Height (m)
count	49.00000	49.000000
mean	75.00000	1.750000
std	10.10363	0.101036
min	60.00000	1.600000
25%	65.00000	1.650000
50%	75.00000	1.750000
75%	85.00000	1.850000
max	90.00000	1.900000

Figure 1.14

Description of independent features after dropping the unnecessary features

1.7.7.4 Splitting Train and Test data

Write the code shown in Figure 1.15 to split train and test data. This code samples 75% of the data for training and 25% of the data for testing. By changing the *test_size* parameter value, it is possible to split into different percentages.

By using 75% of the data for training and 25% for testing, out of 57 records, the training set contains 42 and the test set contains 15 records, respectively. The *len*() function shows the number of records for individual dataset.

```
# Splitting Train and Test data (75% & 25%, respectively) ...
from sklearn.model_selection import train_test_split
# SEED = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

len(X)       # 57
len(X_train) # 42
len(X_test)  # 15

X_test
```

Figure 1.15

Splitting Train and Test data, 75% and 25%, respectively

Figure 1.16 show the test dataset (X_test) after splitting into 25% of total dataset in the variable X.

	Body Weight (kg)	Body Height (m)
30	80	1.70
5	60	1.85
43	90	1.65
21	75	1.60
6	60	1.90
35	85	1.60
16	70	1.70
29	80	1.65
18	70	1.80
12	65	1.85
44	90	1.70
36	85	1.65
28	80	1.60

Figure 1.16
Splitting Train and Test data, 75% and 25%, respectively

1.7.7.5 Creating a test dataset with single record

Write the code shown in Figure 1.17 to create a test dataset with single record of Body Weight as 80 kg and Body Height as 1.7 m. The result of the new dataset X_test2 is shown in the same figure.

```
X_test2 = X_test[:1]
X_test2
```

	Body Weight (kg)	Body Height (m)
30	80	1.7

Figure 1.17
A new test dataset with single record

1.7.7.6 Feature Scaling for KNN Regression

Write the code shown in Figure 1.18 to do the feature scaling for K-NN regression. **Now check and observe each of the datasets (X_tran, X_test, and X_test2) by using your own code.**

First ***StandardScaler*** is imported, then instantiating and fitting is conducted according to the train data (preventing leakage). After that transforming both train and test datasets, feature scaling is performed.

```
# Feature Scaling for KNN Regression...
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Fit only on X_train...
scaler.fit(X_train)

X_t = X_test
X_t2 = X_test2

# Scale both X_train and X_test...
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
X_test2 = scaler.transform(X_test2)
```

Figure 1.18

Scaling of datasets (train and test separately) for K-NN regression

When applied on a *DataFrame*, the scaler maintains only the data points, not the column names. So it is necessary to organize the data into a *DataFrame* again with column names and use *describe*() to observe the changes in mean and std. Figure 1.19 shows the necessary codes and corresponding results.

Observe that all the standard deviations are now became close to 1 and the means have become smaller. This is what makes the data more uniform. Now data are ready to train and evaluate using the K-NN based *regressor*.

```
col_names=['Body Weight (kg)', 'Body Height (m)'] # , 'Weight-Height Ratio (R)', 'Internal Moment (M)']
scaled_df = pd.DataFrame(X_train, columns=col_names)
scaled_df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Body Weight (kg)	36.0	4.502571e-16	1.014185	-1.464704	-0.96252	0.041849	1.046217	1.548402
Body Height (m)	36.0	2.800229e-15	1.014185	-1.703893	-0.66423	-0.144398	0.895266	1.415098

Figure 1.19

Organizing the scaled data into DataFrame again

1.7.8 K-NN Regression

1.7.8.1 Training and Predicting using K-NN Regression

Scikit − Learn makes training ***regressors*** and classifiers very straightforward. The API is intuitive and stable. Now import the ***KNeighborsRegressor*** class from the ***sklearn.neighbors*** module, instantiate it, and fit it to the train data.

In the code, shown in Figure 1.20, the ***n_neighbors*** is the value for *K*, or the number of neighbors the algorithm will take into consideration for choosing a new torque value. The number 5 is the default value for ***KNeighborsRegressor***().

There is no ideal value for *K*. It is selected after testing and evaluation, however, to start out, 5 is a commonly used value for K-NN, thus set as the default value in this

example code. The final step is to make predictions on the test data. To do so, execute the following script and observe the outputs.

```
# Training and Predicting K-NN Regression...
from sklearn.neighbors import KNeighborsRegressor
K = 5
regressor = KNeighborsRegressor(n_neighbors=K)
regressor.fit(X_train, y_train)

y_pred = regressor.predict(X_test)
print(X_test), print(y_test), print(y_pred)
```

Figure 1.20

Training and predicting by using K-NN regressor

1.7.8.2 Predicting for a single record as test

In the code, shown in Figure 1.21, process of predicting required torque for a single input record is presented. The ***y_pred2*** is holding the required torque which is further organized into *DataFrame* in the variable ***pred_df***.

Now, modify the *pred_df* again by adding a new column for existing "*Required Torque* $(N-m)$" in the test dataset.

```
y_pred2 = regressor.predict(X_test2)
y_pred2
```

```
array([12.2702])
```

```
#col_names=['Body Weight (kg)', 'Body Height (m)', 'Weight-Height Ratio (R)', 'Internal Moment (M)']
pred_df = X_t2 # pd.DataFrame(X_test2, columns=col_names)
pred_df
```

	Body Weight (kg)	Body Height (m)
30	80	1.7

```
pred_df['Required Torque (N-m)'] = y_pred2
pred_df
```

	Body Weight (kg)	Body Height (m)	Required Torque (N-m)
30	80	1.7	12.2702

Figure 1.21

Predicting required torque for single input record using K-NN regressor

1.7.8.3 Evaluating the results for K-NN Regression

The most commonly used regression metrics for evaluating the algorithm are mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), and coefficient of determination (R^2).

The **MAE** metric gives a notion of the overall error for each prediction of the model, the smaller (closer to 0) the better. Equation (1.2) shows the general equation for calculating MAE.

$$MAE = \left(\frac{1}{n}\right)\Sigma_{i=1}^{n}|Actual - Predict| \qquad (1.2)$$

The **MSE** metric is similar to the MAE metric, but it squares the absolute values of the errors. Also, as with MAE, the smaller, or closer to 0, the better. The MSE value is squared so as to make large errors even larger.

$$MSE = \left(\frac{1}{n}\right)\Sigma_{i=1}^{D}(Actual - Predict)^2 \tag{1.3}$$

The **RMSE** metric is to solve the interpretation problem raised in the MSE by getting the square root of its final value, to scale it back to the same units of data. It is easier to interpret and good when display of the actual value of the data with the error is needed. It shows how much the data may vary. The closer to 0, the better as well.

$$RMSE = \sqrt{\left(\frac{1}{n}\right)\Sigma_{i=1}^{D}(Actual - Predict)^2} \tag{1.4}$$

The error calculations scripts and corresponding results are shown in Figure 1.21. The R^2 can be calculated directly with the ***score***() method. Results show that the K-NN algorithm overall error and mean error are around 0.27, and 0.10. The RMSE shows that the value can go above or below the actual value by adding or subtracting 0.65. Now describe the predicted output and observe.

```
# Evaluating the Algorithm for KNN Regression...
from sklearn.metrics import mean_absolute_error, mean_squared_error

mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)

print(f'mae: {mae}')
print(f'mse: {mse}')
print(f'rmse: {rmse}')
```

```
mae: 0.2714353846153841
mse: 0.10830680212307671
rmse: 0.32909998803262924
```

```
# R2 can be calculated...
regressor.score(X_test, y_test)
```

```
0.9105616219325816
```

```
y.describe()
```

```
count    49.000000
mean     11.636300
std       1.707826
min       8.511100
25%      10.240000
50%      11.636000
75%      12.811000
max      15.160000
Name: Required Torque (N-m), dtype: float64
```

Figure 1.22

Predicting required torque for single input record using K-NN regressor

The R^2 value, the closest to 1 (or 100) is the better. It tells how much of the changes in data, or data variance are being understood or explained by K-NN. The calculation procedure is presented in Equation (1.5).

$$R^2 = 1 - \left(\frac{\Sigma(Actual - Predict)^2}{\Sigma(Actual - Actual\ Mean)^2}\right) \quad (1.5)$$

With a value of 0.91, it can be observed that the model explains 91% of data variance. It is more than 90%, which is good.

1.7.8.4 Finding the Best K for K-NN Regression

To choose the best K by using only the mean absolute error is possible. It is possible to change it to any other metric and compare the results. The script in Figure 1.23 creates a for loop and run models that have from 1 to N neighbors. At each interaction, the MAE is calculated and plot the number of Ks along with MAE result. Figure 1.24 shows the plot where the lowest MAE value is observed when K is 1. This is obviously not a good result. The output can be varied in your simulation and also could be improved if real dataset can be used.

```
# ========================================
# Finding the Best K for KNN Regression...
# ========================================
error = []

# Calculating MAE error for K values between 1 and 29
for i in range(1, 30):
    knn = KNeighborsRegressor(n_neighbors=i)
    knn.fit(X_train, y_train)
    pred_i = knn.predict(X_test)
    mae = mean_absolute_error(y_test, pred_i)
    error.append(mae)
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(range(1, 30), error, color='red',
         linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=10)

plt.title('K Value MAE')
plt.xlabel('K Value')
plt.ylabel('Mean Absolute Error (MAE)')
```

Figure 1.23

Script to calculate and plot MAE for all possible K values

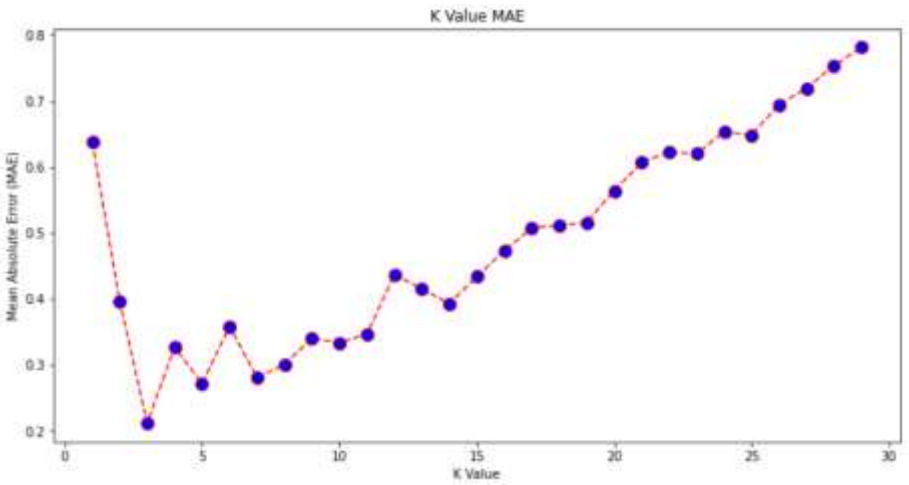

Figure 1.24

Graph of MAE for all possible K values

1.7.8.5 Predicting with minimum error position as K=2

It is possible to obtain the lowest error and the index of that point using the built-in $min()$ function (works on lists) or convert the list into a $NumPy$ array and get the $argmin()$ (index of the element with the lowest value). Figure 1.25 presents the script to predict based on test dataset with minimum error position as $K = 1$.

Observe the result and describe your comments in detail.

```
# ====================================
# Predicting with minimum error position as K...
# ====================================
import numpy as np

# print(min(error))
# print(np.array(error).argmin())

print(min(error[2:]))          # 0.14875230769230813
print(np.array(error[2:]).argmin()+2) # 4
K_neigh = np.array(error[2:]).argmin()+2

knn_reg = KNeighborsRegressor(n_neighbors=4)
knn_reg.fit(X_train, y_train)
y_pred_reg = knn_reg.predict(X_test)
r2_reg = knn_reg.score(X_test, y_test)

mae_reg = mean_absolute_error(y_test, y_pred_reg)
mse_reg = mean_squared_error(y_test, y_pred_reg)
rmse_reg = mean_squared_error(y_test, y_pred_reg, squared=False)
print(f'r2: {r2_reg}, \nmae: {mae_reg} \nmse: {mse_reg} \nrmse: {rmse_reg}')
```

```
0.21149487179487172
2
r2: 0.8485152716705489,
mae: 0.3261134615384617
mse: 0.18344279995192306
rmse: 0.42830222968357645
```

Figure 1.25

Predicting with minimum error position as K=2

1.7.9 K-NN Classification with Scikit-Learn

Instead of predicting a continuous value, it is possible to predict the class to which the block groups belong. To do that, a particular feature can be selected to determine the groups. In this experiment, the given dataset already has the '$Torque\ Category$' column which is used to determine the groups.

1.7.9.1 Preprocessing Data for Classification

Create the data bins to transform continuous values into categories. In this case, it is not necessary as category is given. Figure 1.26 shows the scripts for preprocessing of data for K-NN Classification.

```
# ================================================
# Classification using K-Nearest Neighbors with Scikit-Learn...
# ================================================

dfc = df.dropna()

# Creating 4 categories and assigning them to a NewColumn...
# dfc["NewColumn"] = pd.qcut(dfc["ClassificatinColumn"], 4, retbins=False,
#              labels=[1, 2, 3, 4])
y = dfc['Torque Category']
X = dfc.drop(['Weight-Height Ratio (R)', 'Internal Moment (M)',
              'Required Torque (N-m)', 'Torque Category'], axis = 1)
```

Figure 1.26

Data preprocessing and determining target and independent features

1.7.9.2 Splitting dataset and feature scaling

Figure 1.27 shows the scripts to split the dataset into train data and test data. Feature scaling script is also presented in the same figure. Write the script and observe.

```
# Splitting Data into Train and Test Sets...
from sklearn.model_selection import train_test_split

# SEED = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25) #, random_state=SEED)
```

```
# Feature Scaling for Classification...
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 1.27

Splitting data and feature scaling

1.7.9.3 Training and Predicting for Classification

For the prediction of class, 5 neighbors is considered as baseline. It is possible to instantiate the ***KNeighbors***_ class without any arguments which will automatically set 5 neighbors. Here, instead of importing the *KNeighborsRegressor* , ***KNeighborsClassifier*** is imported.

```
# Training and Predicting for Classification...
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier() # n_neighbors = 5
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
y_pred
```

```
array(['High', 'High', 'High', 'Medium', 'Medium', 'Medium', 'Medium',
       'Medium', 'High', 'High', 'High', 'Medium', 'Medium'], dtype=object)
```

Figure 1.28

Training and prediction of classes

1.7.9.4 Evaluating K-NN for Classification

For evaluating the K-NN classifier, the score method can be used, but it executes a different metric as scoring a classifier is different than scoring a regression. The basic metric for classification is accuracy. It describes how many predictions the classifier got right. The lowest accuracy value is 0 and the highest is 1. Usually that value is multiplied by 100 to obtain a percentage. The equation for accuracy is presented in Equation (1.6). Figure 1.29 shows the script to determine accuracy (R^2 value). By looking at the resulting score (Figure 1.29), it is identified that the classifier got ≈ 84% of the classes as correct.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ Predections} \tag{1.6}$$

```
# Evaluating KNN for Classification...
acc =  classifier.score(X_test, y_test)
print(acc) # 0.8461538461538461
```

```
0.8461538461538461
```

Figure 1.29

Evaluating the prediction of classes

Other metrics are able to evaluate the results, such as Confusion Matrix, Precision, Recall, and F1-score.

Confusion Matrix: It is used to know the count (how many) of right or wrong for each class. The values that are correct and correctly predicted are called true positives, the ones that are predicted as positives but are not positives are called false positives. The same nomenclature of true negatives and false negatives is used for negative values.

Precision: It is used to understand what correct prediction values are considered correct by the classifier. Precision will divide those true positives values by anything that was predicted as a positive. The relevant equation is shown in Equation (1.7).

$$Precision = \frac{True\ Positive}{Total\ PREDICTED\ positive} = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{1.7}$$

Recall: It helps to understand how many of the true positives are identified by the classifier. The recall is calculated by dividing the true positives by anything that should have been predicted as positive. The equation is presented in Equation (1.8).

$$Recall = \frac{True\ Positive}{Total\ ACTUAL\ positive} = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{1.8}$$

F1-score (Harmonic mean): It is the balanced mean of precision and recall (**Harmonic Precision-recall mean**). The lowest value is 0 and the highest is 1. The $F1\ score = 1$ means all classes are correctly predicted. This is a very hard score to obtain with real data (exceptions always exist). Necessary equation is presented in Equation (1.9).

$$F1\ score = \frac{1}{\frac{\frac{1}{Precession} + \frac{1}{Recall}}{2}} = 2 \times \frac{Precession \times Recall}{Precession + Recall} \tag{1.9}$$

1.7.9.5 Visualizing through Confusion Matrix

The ***confusion_matrix***() and ***classification_report***() methods of the ***sklearn.metrics*** module can be used to calculate and display all these metrics. The ***confusion_matrix*** is better visualized using a heat-map graph. The classification report gives the accuracy, precision, recall, and F1-score. To obtain metrics, execute the following scripts shown in Figure 1.30. Figure 1.31 presents the corresponding outputs. The results show that K-NN is able to classify all the records in the test set with **85**% accuracy.

```
# Visualize using a heatmap....
from sklearn.metrics import classification_report, confusion_matrix
#importing Seaborn's to use the heatmap
import seaborn as sns

# Adding classes names for better interpretation
classes_names = ['High','Medium']
cm = pd.DataFrame(confusion_matrix(y_test, y_pred),
                  columns=classes_names, index = classes_names)

# Seaborn's heatmap to better visualize the confusion matrix
sns.heatmap(cm, annot=True, fmt='d');

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

Figure 1.30

Script to visualize the Confusion matrix and classification report

```
              precision    recall  f1-score   support

        High       0.67      1.00      0.80         4
      Medium       1.00      0.78      0.88         9

    accuracy                           0.85        13
   macro avg       0.83      0.89      0.84        13
weighted avg       0.90      0.85      0.85        13
```

Figure 1.31

Visualizing the Confusion matrix and classification report

1.7.9.6 Comparing the test classes with the predicted classes

Figure 1.32 presents the scripts to print the test and predicted datasets for visual comparison.

```
print(f'Test: {np.array(y_test)} \n')
print(f'Pred: {y_pred}')
```

```
Test: ['High' 'Medium' 'Medium' 'Medium' 'Medium' 'Medium' 'Medium' 'Medium'
 'High' 'High' 'High' 'Medium' 'Medium']

Pred: ['High' 'High' 'High' 'Medium' 'Medium' 'Medium' 'Medium' 'Medium' 'High'
 'High' 'High' 'Medium' 'Medium']
```

Figure 1.32

Printing the test and predicted datasets for visual comparison

1.7.9.7 Finding the Best K for K-NN Classification

Figure 1.33 presents the scripts to plot K-values vs. F1-score to find the best K-value for K-NN classification. Figure 1.34 presents the corresponding graph.

```
# ============================================
# Finding the Best K for KNN Classification...
# ============================================
from sklearn.metrics import f1_score

f1s = []

# Calculating f1 score for K values between 1 and 30
for i in range(1, 30):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    pred_i = knn.predict(X_test)
    # average='weighted' to calculate a weighted average for the classes
    f1s.append(f1_score(y_test, pred_i, average='weighted'))

plt.figure(figsize=(12, 6))
plt.plot(range(1, 30), f1s, color='red', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=10)
plt.title('F1 Score K Value')
plt.xlabel('K Value')
plt.ylabel('F1 Score')
```

Figure 1.33

Script to generate K-values vs. F1-score graph

From the output graph (Figure 1.34), it can be seen that the F1-score is the highest when the value of the K is 7 or 11. Now retrain the classifier with 7, 11, and **any other K-values as neighbors and observe what it does to the classification report results.** Figure 1.35 presents the necessary scripts for generating classification reports for three different K-values.

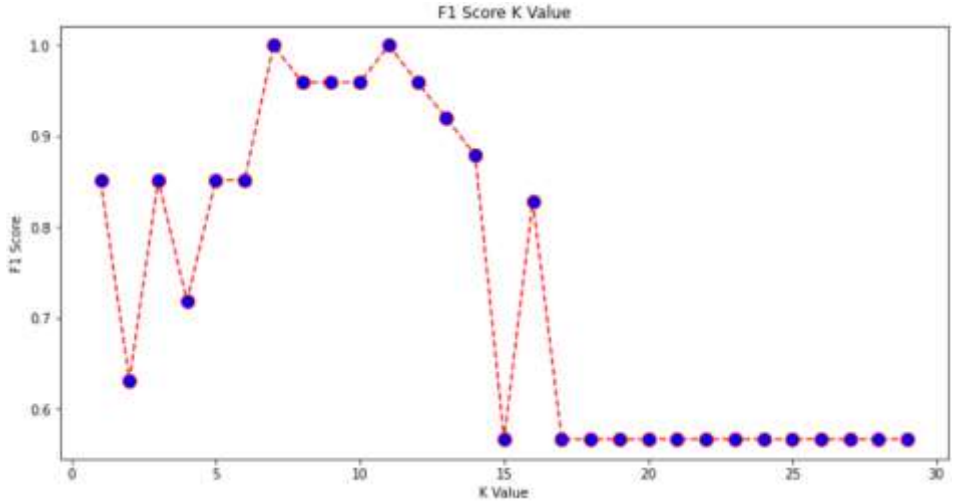

Figure 1.34

K-values vs. F1-score graph

```
np.array(f1s)
```

```
array([0.85192308, 0.63116371, 0.85192308, 0.71828172, 0.85192308,
       0.85192308, 1.        , 0.95927602, 0.95927602, 0.95927602,
       1.        , 0.95927602, 0.91960671, 0.87912088, 0.56643357,
       0.82820513, 0.56643357, 0.56643357, 0.56643357, 0.56643357,
       0.56643357, 0.56643357, 0.56643357, 0.56643357, 0.56643357,
       0.56643357, 0.56643357, 0.56643357, 0.56643357, 0.56643357])
```

```
# The f1-score is the highest when the value of the K is 7 or 11...
# Retrain the classifier with 7 neighbors...
classifier7 = KNeighborsClassifier(n_neighbors=7)
classifier7.fit(X_train, y_train)
y_pred7 = classifier7.predict(X_test)
print(classification_report(y_test, y_pred7))
```

```
              precision    recall  f1-score   support

        High       1.00      1.00      1.00         4
      Medium       1.00      1.00      1.00         9

    accuracy                           1.00        13
   macro avg       1.00      1.00      1.00        13
weighted avg       1.00      1.00      1.00        13
```

```
# Retrain the classifier with 11 neighbors...
classifier11 = KNeighborsClassifier(n_neighbors=11)
classifier11.fit(X_train, y_train)
y_pred11 = classifier11.predict(X_test)
print(classification_report(y_test, y_pred11))
```

```
              precision    recall  f1-score   support

        High       1.00      1.00      1.00         4
      Medium       1.00      1.00      1.00         9

    accuracy                           1.00        13
   macro avg       1.00      1.00      1.00        13
weighted avg       1.00      1.00      1.00        13
```

Figure 1.35

Classification reports for various K-values

1.7.10 Implementing K-NN for Outlier Detection with Scikit-Learn

1.7.10.1 Calculating distances of each data points

Outlier detection uses different method than what was conducted previously for regression and classification. Here, it will be observed that how far each of the neighbors is from a data point. First consider the default 5 neighbors. For a data point, the distance to each of the K-nearest neighbors are calculated. To do that, it is necessary to import another K-NN algorithm (***NearestNeighbors***) from Scikit-learn which is not specific for either regression or classification. Figure 1.36 illustrates the script to calculate distances for each data point.

```
# ============================================================
# Implementing KNN for Outlier Detection with Scikit-Learn...
# ============================================================
from sklearn.neighbors import NearestNeighbors

nbrs = NearestNeighbors(n_neighbors = 5)
nbrs.fit(X_train)
# Distances and indexes of the 5 neighbors...
distances, indexes = nbrs.kneighbors(X_train)

# 5 distances for each data point (distance between itself and 5 neighbors)...
distances[:3], distances.shape
```

```
(array([[0.        , 0.48989795, 0.48989795, 0.51034318, 0.70742502],
        [0.        , 0.48989795, 0.51034318, 0.70742502, 0.9797959 ],
        [0.        , 0.48989795, 0.51034318, 0.70742502, 1.02068636]]),
 (36, 5))
```

```
# Look at the neighbors' indexes for 3 rows...
indexes[:3], indexes[:3].shape
```

```
(array([[ 0, 23, 27,  4, 25],
        [ 1, 23, 10, 25,  0],
        [ 2, 24, 11,  6, 30]]), (3, 5))
```

Figure 1.36

Script to calculate distances for each data point (between itself and 4 neighbors)

1.7.10.2 Calculating mean of the 5 distances and plot graph

Now, continue to calculate the mean of the 5 distances and plot a bar graph that counts each row on the X-axis and displays each mean distance on the Y-axis. Figure 1.37 shows the scripts to calculate mean of the 5 distances and plot a bar graph with threshold line.

In the plotted graph, there is a part where the mean distances have uniform values. That Y-axis point in which the means are not too high or too low is exactly the point that needs to be identified to cut off the outlier values. In this case, the cut off distance is considered as 0.56, as presented in the graph of Figure 1.37, the horizontal dotted red line.

```
# calculate the mean of the 5 distances and plot a graph that counts
# each row on the X-axis and displays each mean distance on the Y-axis...
dist_means = distances.mean(axis=1)
plt.bar(np.array(range(0, 36)), dist_means)
plt.title('Mean of the 5 neighbors distances for each data point')
plt.xlabel('Count')
plt.ylabel('Mean Distances')

plt.axhline(y = 0.56, color = 'r', linestyle = '--')
```

```
<matplotlib.lines.Line2D at 0x7f2e82d31710>
```

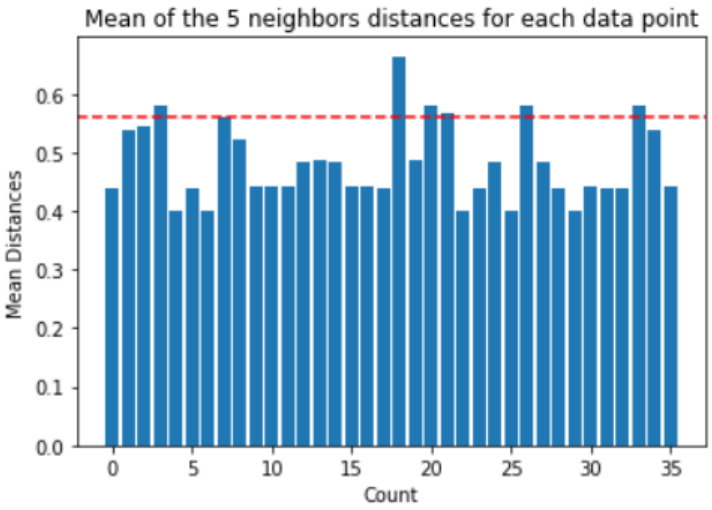

Figure 1.37

Calculate mean of the 5 distances and plot a bar graph with threshold line

1.7.10.3 Determine outlier point indexes and locate in the Data frame

Figure 1.38 shows the scripts to determine the outlier point indexes, and locate them in the data frame.

```
import numpy as np

# Visually determine cutoff values > 0.56
outlier_index = np.where(dist_means > 0.56)
outlier_index
```

```
(array([ 3,  7, 18, 20, 21, 26, 33]),)
```

```
# Filter outlier values (locate them in the dataframe)...
outlier_values = df1.iloc[outlier_index]
outlier_values
```

	Body Weight (kg)	Body Height (m)	Weight-Height Ratio (R)	Internal Moment (N)	Required Torque (N-m)	Torque Category
3	60	1.75	34.286	0.94990	9.3091	Low
7	65	1.60	40.625	0.94086	9.2204	Low
18	70	1.80	38.889	1.13990	11.1710	Medium
20	70	1.90	36.842	1.20320	11.7910	Medium
21	75	1.60	46.875	1.08560	10.6390	Medium
26	75	1.85	40.541	1.25520	12.3010	Medium
33	80	1.85	43.243	1.33890	13.1210	High

Figure 1.38

Outlier point indexes and locate in the Data frame

1.8 RESULT/COMMENTS:

Run all the codes sequentially, observe all the outputs, and write comments in the report form.

Notes: The dataset is small in size and it does not contain real data. Moreover, the Training and Test data selection are random. So, the simulated results could be different from the results shown in this document.

1.9 INSTRUCTIONS:

1. Run all the codes/models and show to invigilator/instructor.
1. Observe each of the results and provide your comments in the form.
3. Fill up the report form properly, take signature from instructor, and submit.

REFERENCES:

Akhtaruzzaman, M., Shafie, A. A., Khan, M. R., & Rahman, M. M. (2019, December 20-22). Knee Joint Kinesiology: A Study on Human Knee Joint Mechanics. 4th International Conference on Electrical Information and Communication Technology (EICT). Khulna, Bangladesh.

Akhtaruzzaman, M., Shafie, A. A., Khan, M. R., & Rahman, M. M. (2020). Modeling and Control Simulation of a Robotic Chair-Arm: Protection against COVID-19 in Rehabilitation Exercise. MIST INTERNATIONAL JOURNAL OF SCIENCE AND TECHNOLOGY, 8(2), 31-40. Doi: https://doi.org/10.47981/j.mijst.08(02)2020.214(31-40)

Sampaio, C. (2013-2022). Guide to the K-Nearest Neighbors Algorithm in Python and Scikit-Learn. Stack Abuse. Extracted from: *https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn*; Extracted on: 10th Oct. 2021.

REPORT ON
MACHINE LEARNING

EXPERIMENT NO.: 02
TITLE: K-NN REGRESSION, CLASSIFICATION, AND OUTLIER DETECTION

DATE:
Batch & Section:

Student ID: _____ Student Name: _____

Objectives of the experiment:

Some Outputs/Graphs with appropriate title:

Comments (overall understanding):

To fill by the instructor:

Marks from the instructor: ☐ X ☐ Y ☐ Z

Instructor's Name and Signature: _____