Data Science

Algorithms

TASK I: Implement and evaluate k-Nearest Neighbor (k-NN) classifiers for a given dataset.

This question is divided into two sub-questions. Please follow the instructions and answer both sub-questions accordingly.

1. Implement the k-NN classifiers.

Given a dataset with numerical or categorical features and a target variable, implement k-Nearest Neighbor (k-NN) classifiers from scratch. Your implementation should include the following functions:

1.1. Function to calculate the distance between two instances, considering both numerical and categorical features. You may choose an appropriate distance metric for this purpose, such as Euclidean distance or Manhattan distance for numerical features and Hamming distance for categorical features.

1.2. Function to predict the class of a new instance based on the calculated distances and finding the nearest neighbor(s) in the training dataset. Your function should be able to handle both k-NN cases, depending on the value of k.

You may use Python programming language. Please provide clear and concise code, along with comments explaining your implementation.

A summary of different distance measuring algorithms

1. Role of Distance Measures
2. Hamming Distance
3. Euclidean Distance
4. Manhattan Distance (Taxicab or City Block)

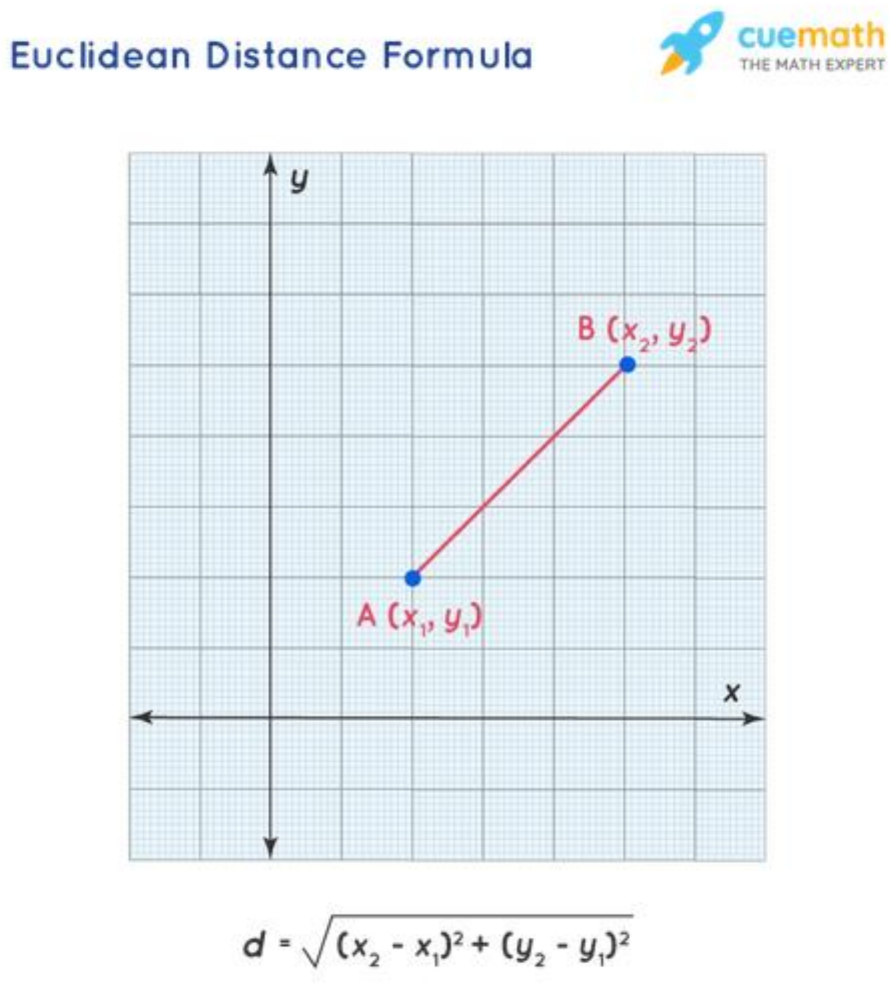
5. Minkowski Distance

1. Euclidean distance

The Euclidean distance is the separation of two real-valued vectors or points. The Euclidean distance formula can be obtained by using Pythagoras' theorem. When utilizing the Euclidean distance method, the distance between two rows of data should have numerical values, such as floating points or integer values. If any of the columns contain values with different scales, then normalizing or standardizing the numerical values across all columns is required before calculating the Euclidean distance. If not, columns with high values will be used to drive the distance measure.

To find the Euclidean distance, take the square root of the total squared differences between the two vectors.

The formular of the Euclidean distance:


Euclidean Distance Formula
cuemath
THE MATH EXPERT
y
x
B (x_2, y_2)
A (x_1, y_1)
$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Source: cuemath.com

d = √ [(x2– x1)^2 + (y2 – y1)^2]

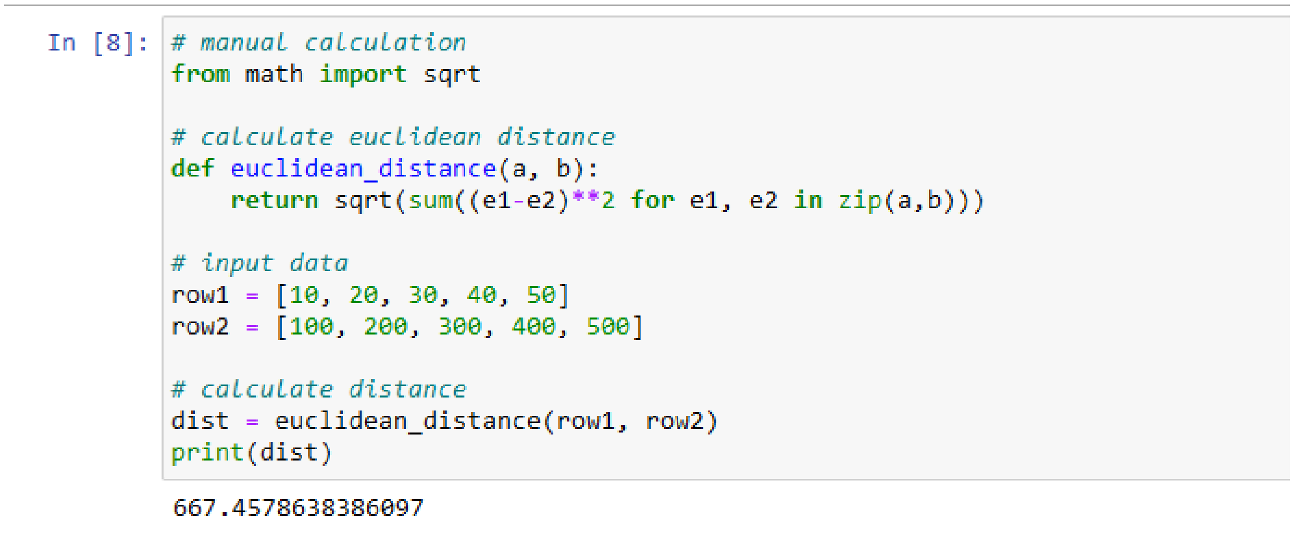
- (x1, y1) are the coordinates of one point.
- (x2, y2) are the coordinates of the other point.
- d is the distance between (x1, y1) and (x2, y2).

This equation can be generalized to n-dimensional space as:

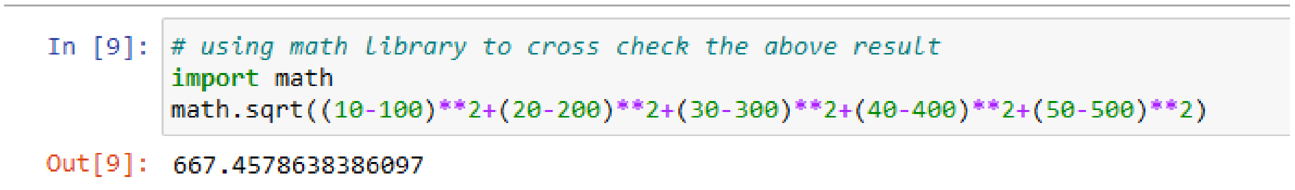
$$d = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2 + \ldots + (x_n - x_1)^2)}$$

It is standard procedure to exclude the square root approach if the distance computation will be performed dozens or millions of times in an attempt to expedite the calculation.

Unit Test: Calculating Euclidean distance between 2 numerical vectors.

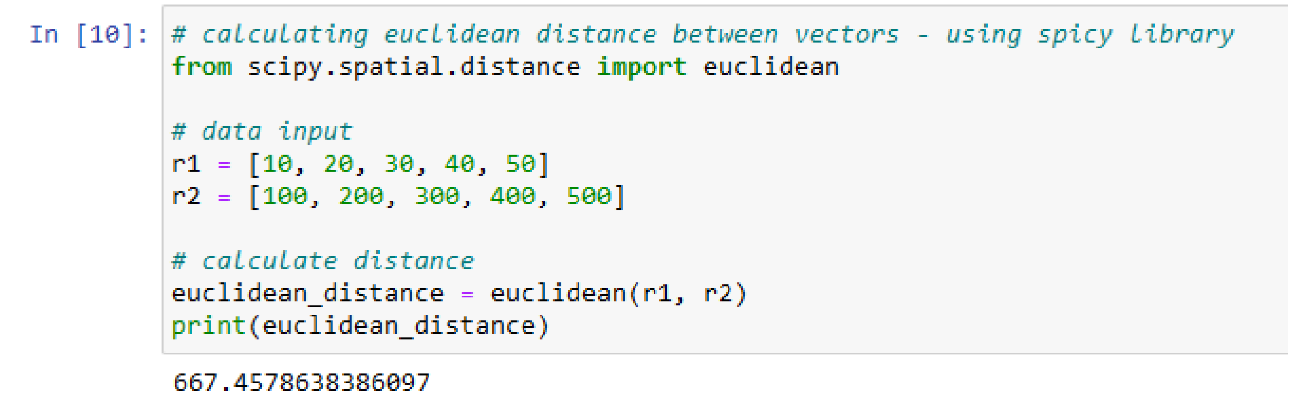


```
In [8]: # manual calculation
        from math import sqrt

        # calculate euclidean distance
        def euclidean_distance(a, b):
            return sqrt(sum((e1-e2)**2 for e1, e2 in zip(a,b)))

        # input data
        row1 = [10, 20, 30, 40, 50]
        row2 = [100, 200, 300, 400, 500]

        # calculate distance
        dist = euclidean_distance(row1, row2)
        print(dist)

        667.4578638386097
```



```
In [9]: # using math library to cross check the above result
        import math
        math.sqrt((10-100)**2+(20-200)**2+(30-300)**2+(40-400)**2+(50-500)**2)

Out[9]: 667.4578638386097
```

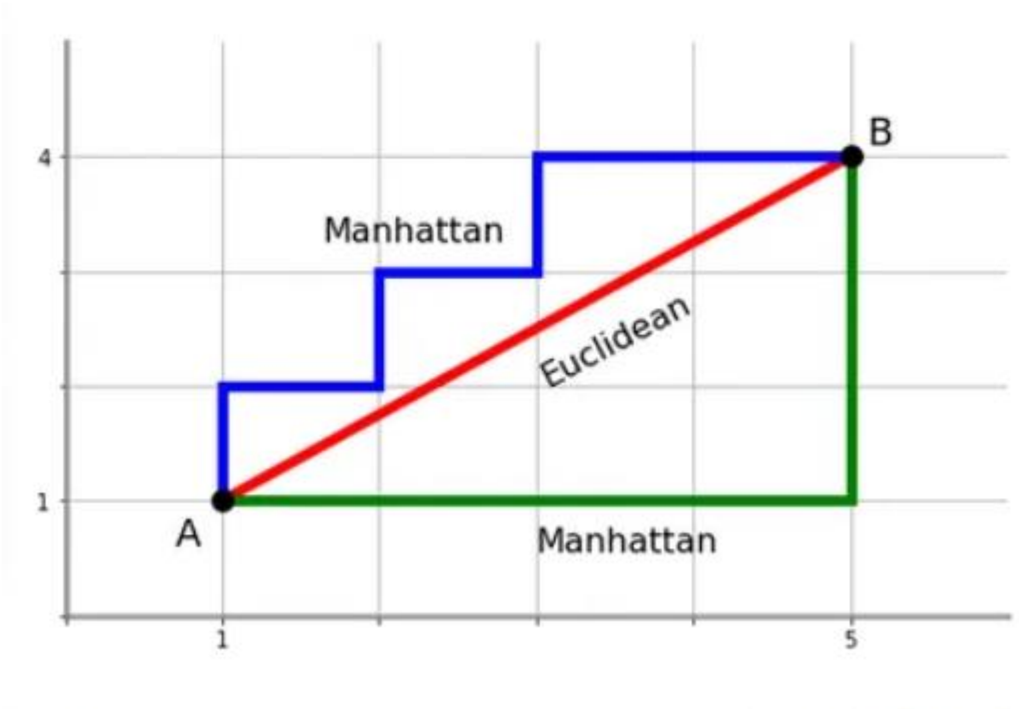
Unit Test: Comparing above results by using the Euclidean python library.



```
In [10]: # calculating euclidean distance between vectors - using spicy library
         from scipy.spatial.distance import euclidean

         # data input
         r1 = [10, 20, 30, 40, 50]
         r2 = [100, 200, 300, 400, 500]

         # calculate distance
         euclidean_distance = euclidean(r1, r2)
         print(euclidean_distance)

         667.4578638386097
```

2. Manhattan distance

Manhattan distance is a metric for estimating the distance between two points. It is the total of the absolute differences in coordinates between these locations.

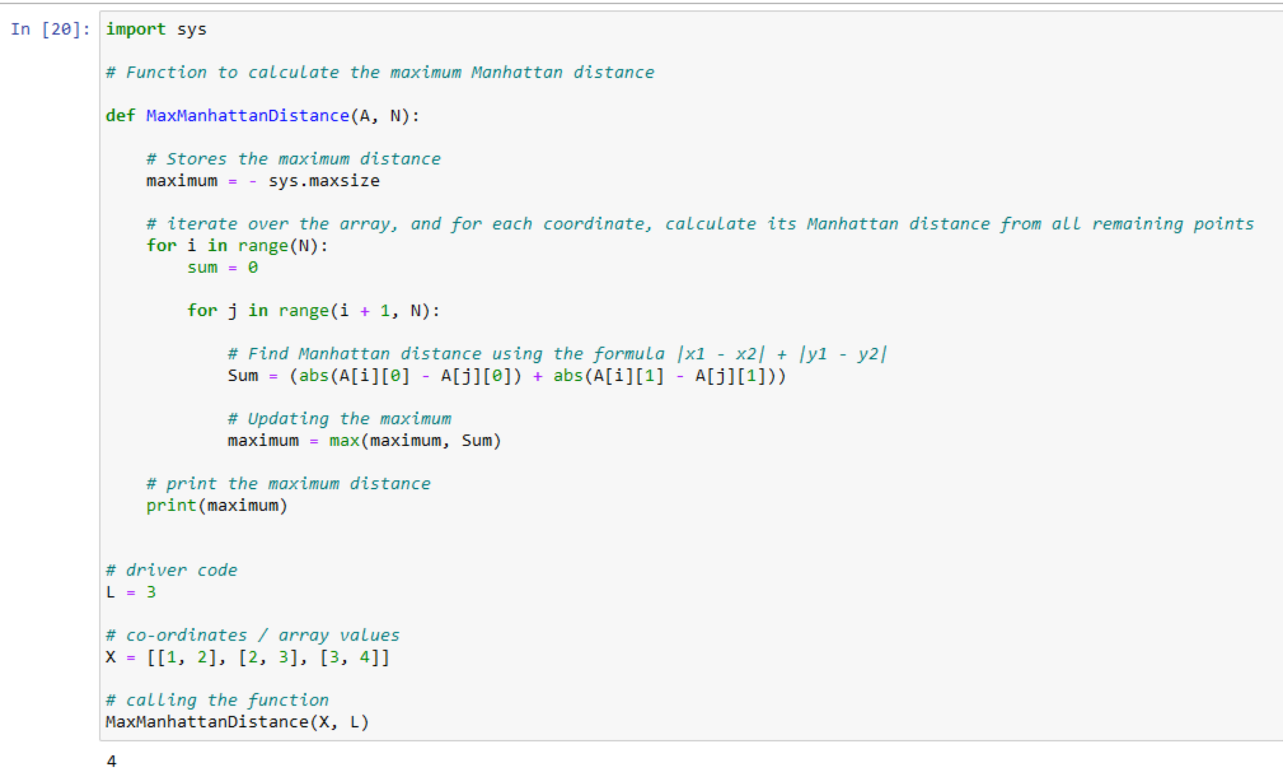


The Manhattan distance and the Euclidean distance between points $A(1,1)$ *and* $B(5,4)$.
The Manhattan distance is longer, and you can find it with more than one path.

Source: omnicalculator.com

Manhattan Distance between two points (A1, B1) and (A2, B2) = |A1 – A2| + |B1 – B2|**.**

Unit Test: Calculating the maximum Manhattan Distance.



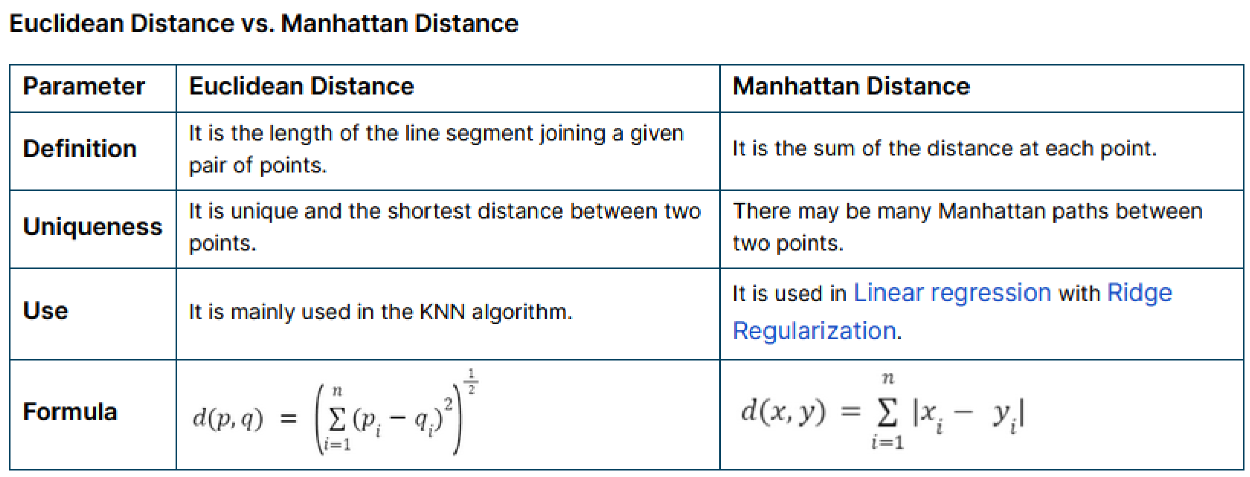
```
In [20]: import sys

         # Function to calculate the maximum Manhattan distance

         def MaxManhattanDistance(A, N):

             # Stores the maximum distance
             maximum = - sys.maxsize

             # iterate over the array, and for each coordinate, calculate its Manhattan distance from all remaining points
             for i in range(N):
                 sum = 0

                 for j in range(i + 1, N):

                     # Find Manhattan distance using the formula |x1 - x2| + |y1 - y2|
                     Sum = (abs(A[i][0] - A[j][0]) + abs(A[i][1] - A[j][1]))

                     # Updating the maximum
                     maximum = max(maximum, Sum)

             # print the maximum distance
             print(maximum)


         # driver code
         L = 3

         # co-ordinates / array values
         X = [[1, 2], [2, 3], [3, 4]]

         # calling the function
         MaxManhattanDistance(X, L)

         4
```

Explanation:

Input array = [[1, 2], [2, 3], [3, 4]]

Output = |1-2| + |2-3| + |2-3| + |3-4| = 4

Comparison between the Euclidean Distance and Manhattan Distance.



Euclidean Distance vs. Manhattan Distance

Parameter	Euclidean Distance	Manhattan Distance
Definition	It is the length of the line segment joining a given pair of points.	It is the sum of the distance at each point.
Uniqueness	It is unique and the shortest distance between two points.	There may be many Manhattan paths between two points.
Use	It is mainly used in the KNN algorithm.	It is used in Linear regression with Ridge Regularization.
Formula	$d(p,q) = \left(\sum_{i=1}^{n}(p_i - q_i)^2\right)^{\frac{1}{2}}$	$d(x,y) = \sum_{i=1}^{n} \lvert x_i - y_i \rvert$

Source: shiksha.com

3. Hamming distance

The Hamming distance is used to calculate the distances between category variables, also known as nominal variables. The amount of difference between two binary strings is represented by the Hamming distance. Regarding categorical variables, there is no such thing as order. Because of this particular property of a categorical variable, it is necessary to compute the change in categorical values relative to binary values.

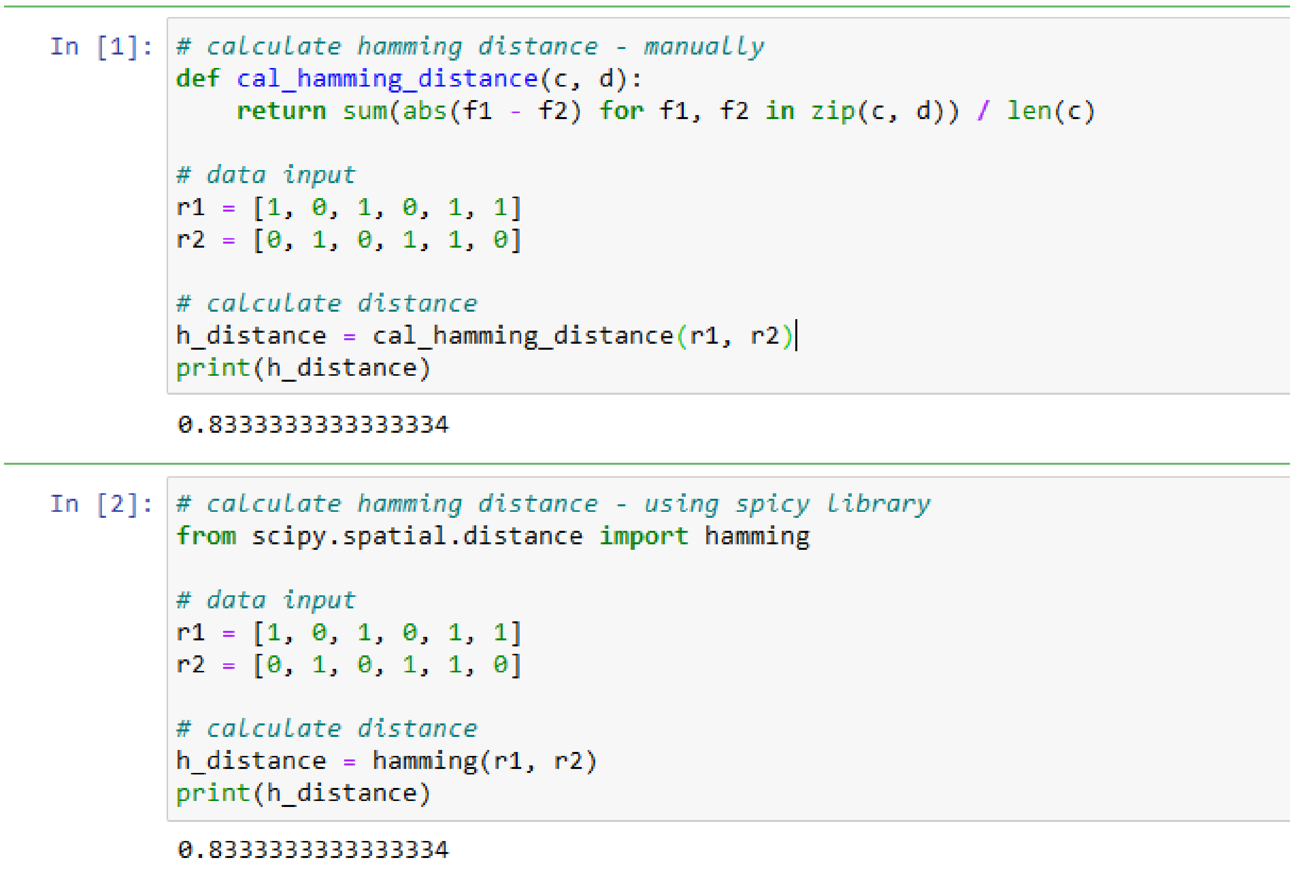
Formular for Hamming Distance:

Hamming Distance = sum for i to N abs(v1[i] – v2[i])

or

HammingDistance = (sum for i to N abs(v1[i] – v2[i])) / N

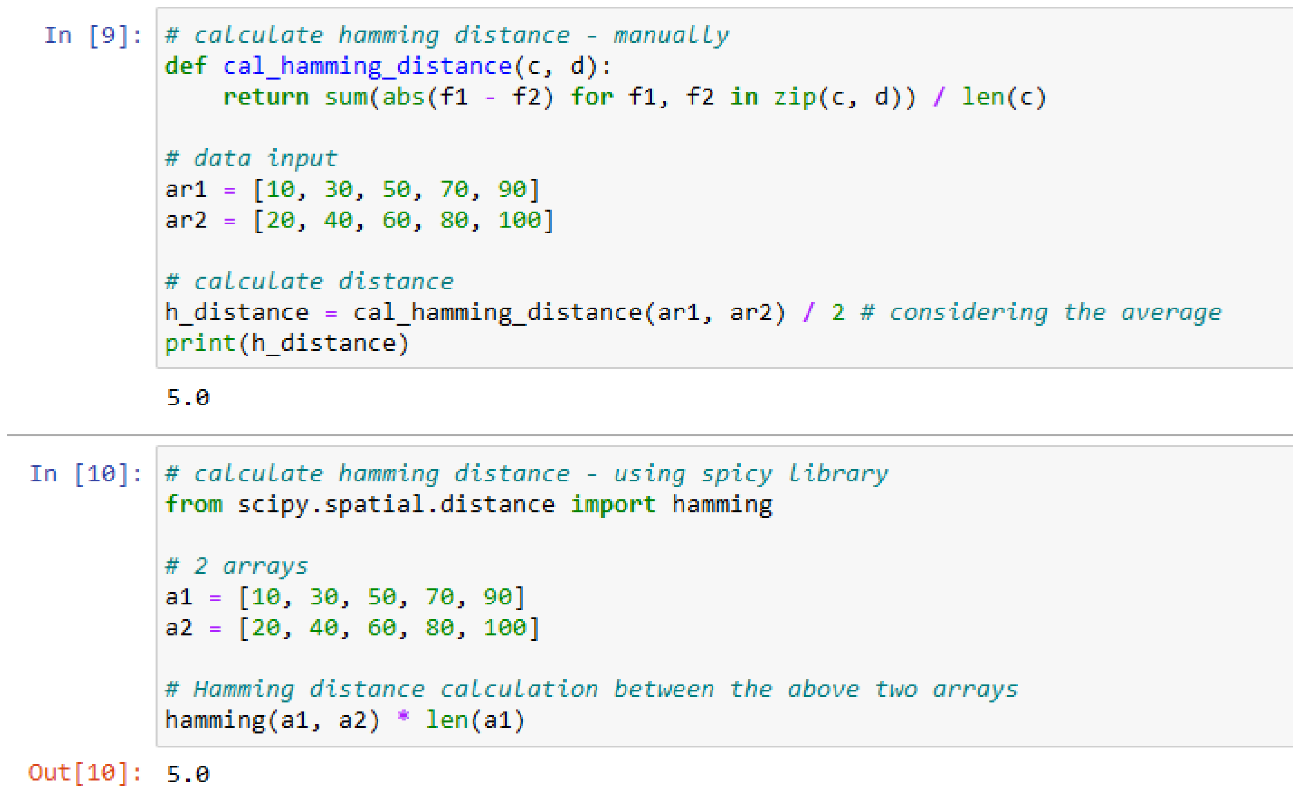
Unit Test: Calculating Hamming Distance between binary arrays.



```
In [1]: # calculate hamming distance - manually
        def cal_hamming_distance(c, d):
            return sum(abs(f1 - f2) for f1, f2 in zip(c, d)) / len(c)

        # data input
        r1 = [1, 0, 1, 0, 1, 1]
        r2 = [0, 1, 0, 1, 1, 0]

        # calculate distance
        h_distance = cal_hamming_distance(r1, r2)
        print(h_distance)

        0.8333333333333334
```

```
In [2]: # calculate hamming distance - using spicy library
        from scipy.spatial.distance import hamming

        # data input
        r1 = [1, 0, 1, 0, 1, 1]
        r2 = [0, 1, 0, 1, 1, 0]

        # calculate distance
        h_distance = hamming(r1, r2)
        print(h_distance)

        0.8333333333333334
```

Unit Test: Calculating Hamming Distance between numerical arrays.

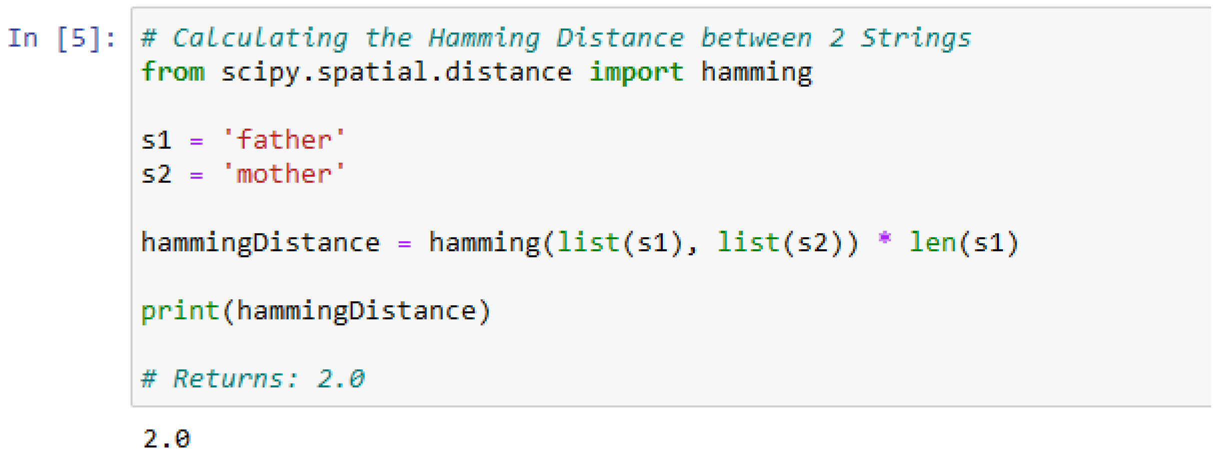


```
In [9]:  # calculate hamming distance - manually
         def cal_hamming_distance(c, d):
             return sum(abs(f1 - f2) for f1, f2 in zip(c, d)) / len(c)

         # data input
         ar1 = [10, 30, 50, 70, 90]
         ar2 = [20, 40, 60, 80, 100]

         # calculate distance
         h_distance = cal_hamming_distance(ar1, ar2) / 2 # considering the average
         print(h_distance)

         5.0
```

```
In [10]: # calculate hamming distance - using spicy library
         from scipy.spatial.distance import hamming

         # 2 arrays
         a1 = [10, 30, 50, 70, 90]
         a2 = [20, 40, 60, 80, 100]

         # Hamming distance calculation between the above two arrays
         hamming(a1, a2) * len(a1)

Out[10]: 5.0
```

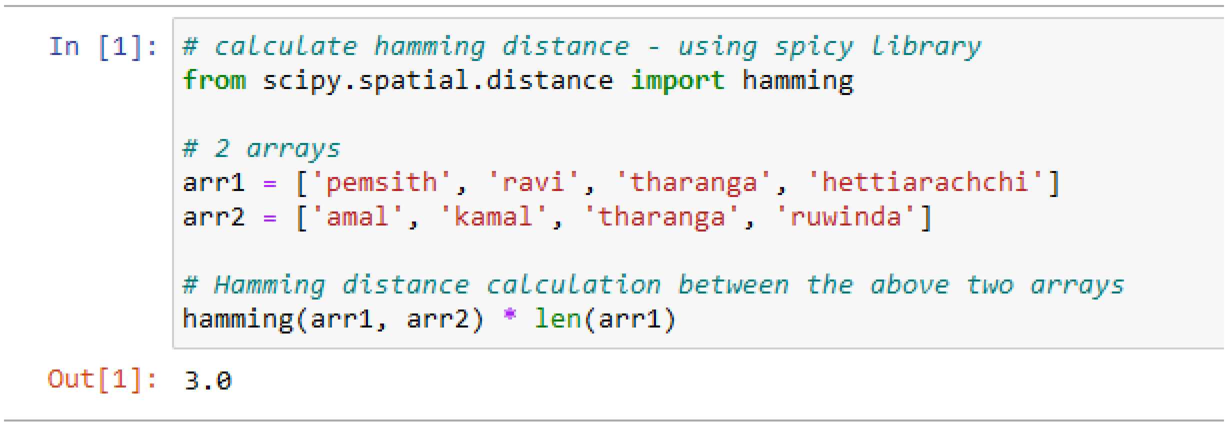
Unit Test: Calculating Hamming Distance between strings.

Condition: length of both strings should be the same.



```
In [5]:  # Calculating the Hamming Distance between 2 Strings
         from scipy.spatial.distance import hamming

         s1 = 'father'
         s2 = 'mother'

         hammingDistance = hamming(list(s1), list(s2)) * len(s1)

         print(hammingDistance)

         # Returns: 2.0

         2.0
```

Unit Test: Calculating Hamming Distance between string arrays.



```
In [1]: # calculate hamming distance - using spicy library
        from scipy.spatial.distance import hamming

        # 2 arrays
        arr1 = ['pemsith', 'ravi', 'tharanga', 'hettiarachchi']
        arr2 = ['amal', 'kamal', 'tharanga', 'ruwinda']

        # Hamming distance calculation between the above two arrays
        hamming(arr1, arr2) * len(arr1)

Out[1]: 3.0
```

The k-Nearest Neighbors (KNN) algorithm:

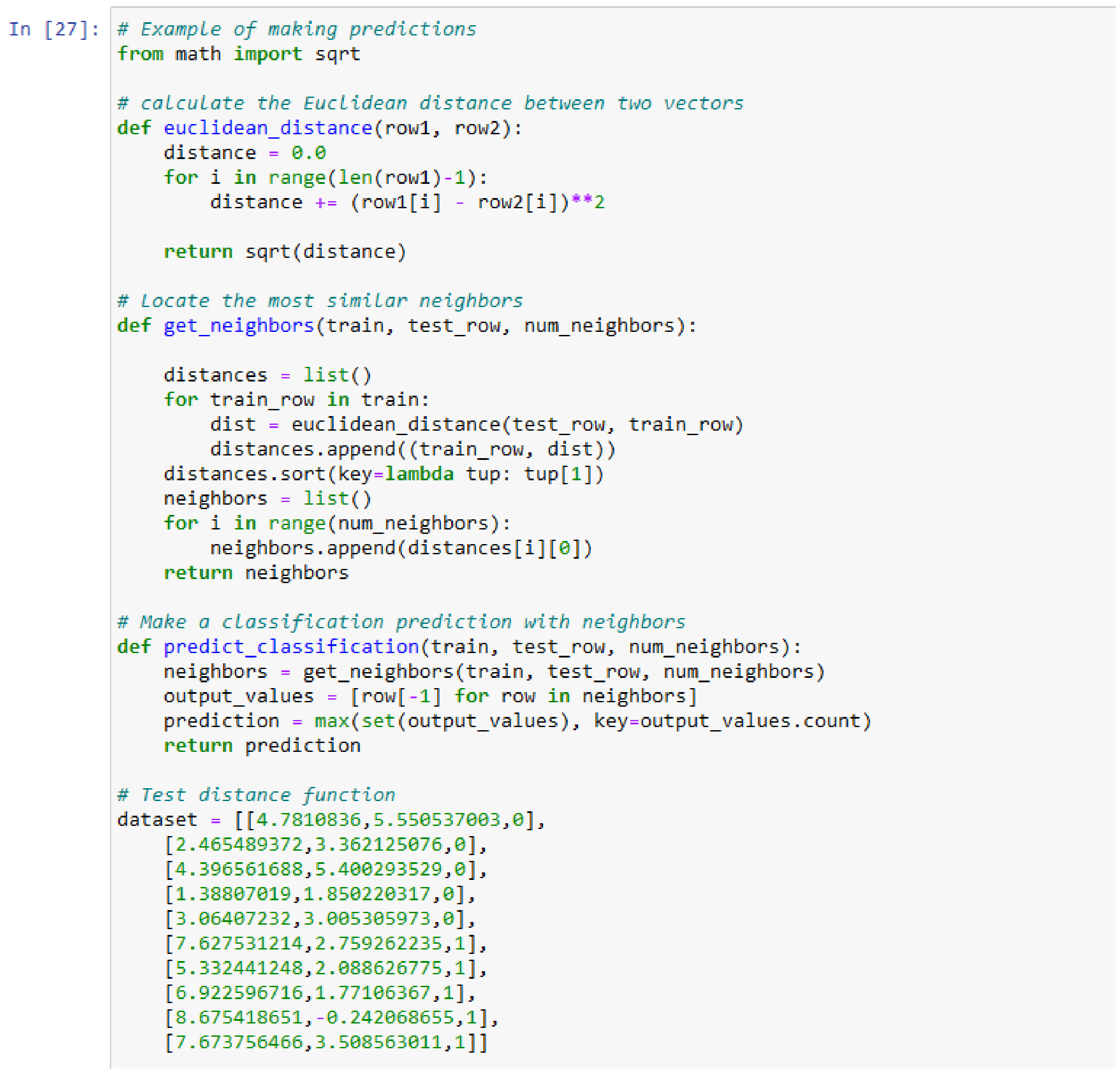
Finding the k-Nearest Neighbors and selecting the ideal k value is one of the key phases in utilizing the KNN.

Steps to finding k-Nearest Neighbours.

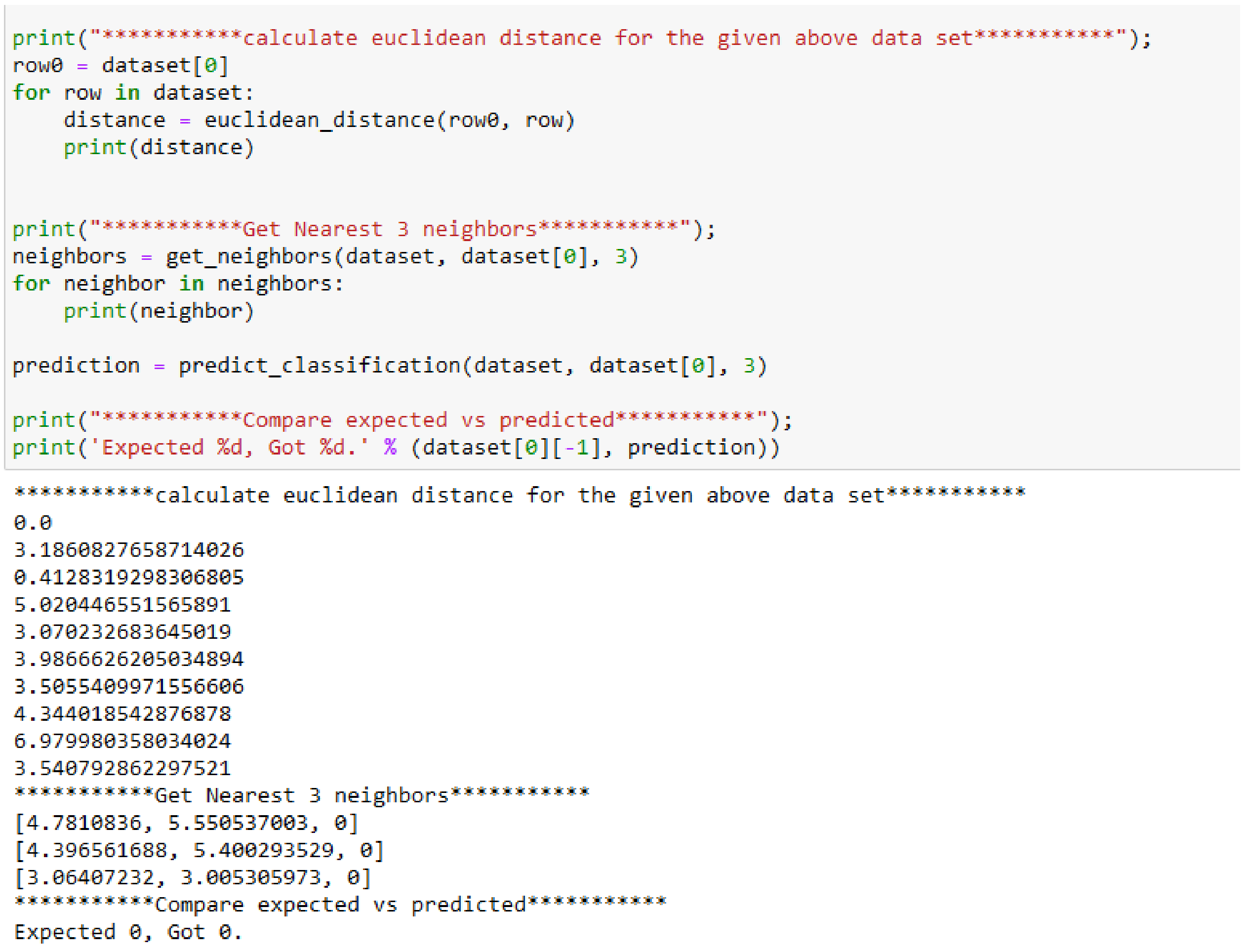
1. Decide on a value for k, which indicates how many neighbors KNN will take into account when making future predictions. Smoother predictions are produced by algorithms with larger k values, whereas more local variances are introduced by algorithms with lower k values.
2. Calculate Euclidean distances: Ascertain the distance in between each dataset point and the points that the model need to forecast or classify. This means calculating the Euclidean distance in the high-dimensional feature space.
3. Choose the neighbor that is k-nearest: After sorting the computed distances in ascending order, the top k data points with the shortest Euclidean distances are identified as the nearest neighbors.

4. Classification using majority voting: Assign a class label to a newly discovered data point in a classification work, taking into account the majority class of its k-nearest neighbors. This is often achieved by voting.
5. Average of regression: To estimate the goal value for the new data point in a regression task, average the target values of the k-nearest neighbors.

Unit Test: Test with a predefined dataset.



```
In [27]: # Example of making predictions
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2

    return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):

    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors

# Make a classification prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction

# Test distance function
dataset = [[4.7810836,5.550537003,0],
    [2.465489372,3.362125076,0],
    [4.396561688,5.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]
```



```
print("***********calculate euclidean distance for the given above data set***********");
row0 = dataset[0]
for row in dataset:
    distance = euclidean_distance(row0, row)
    print(distance)


print("***********Get Nearest 3 neighbors***********");
neighbors = get_neighbors(dataset, dataset[0], 3)
for neighbor in neighbors:
    print(neighbor)

prediction = predict_classification(dataset, dataset[0], 3)

print("***********Compare expected vs predicted***********");
print('Expected %d, Got %d.' % (dataset[0][-1], prediction))
```

```
***********calculate euclidean distance for the given above data set***********
0.0
3.1860827658714026
0.4128319298306805
5.020446551565891
3.070232683645019
3.9866626205034894
3.5055409971556606
4.344018542876878
6.979980358034024
3.540792862297521
***********Get Nearest 3 neighbors***********
[4.7810836, 5.550537003, 0]
[4.396561688, 5.400293529, 0]
[3.06407232, 3.005305973, 0]
***********Compare expected vs predicted***********
Expected 0, Got 0.
```

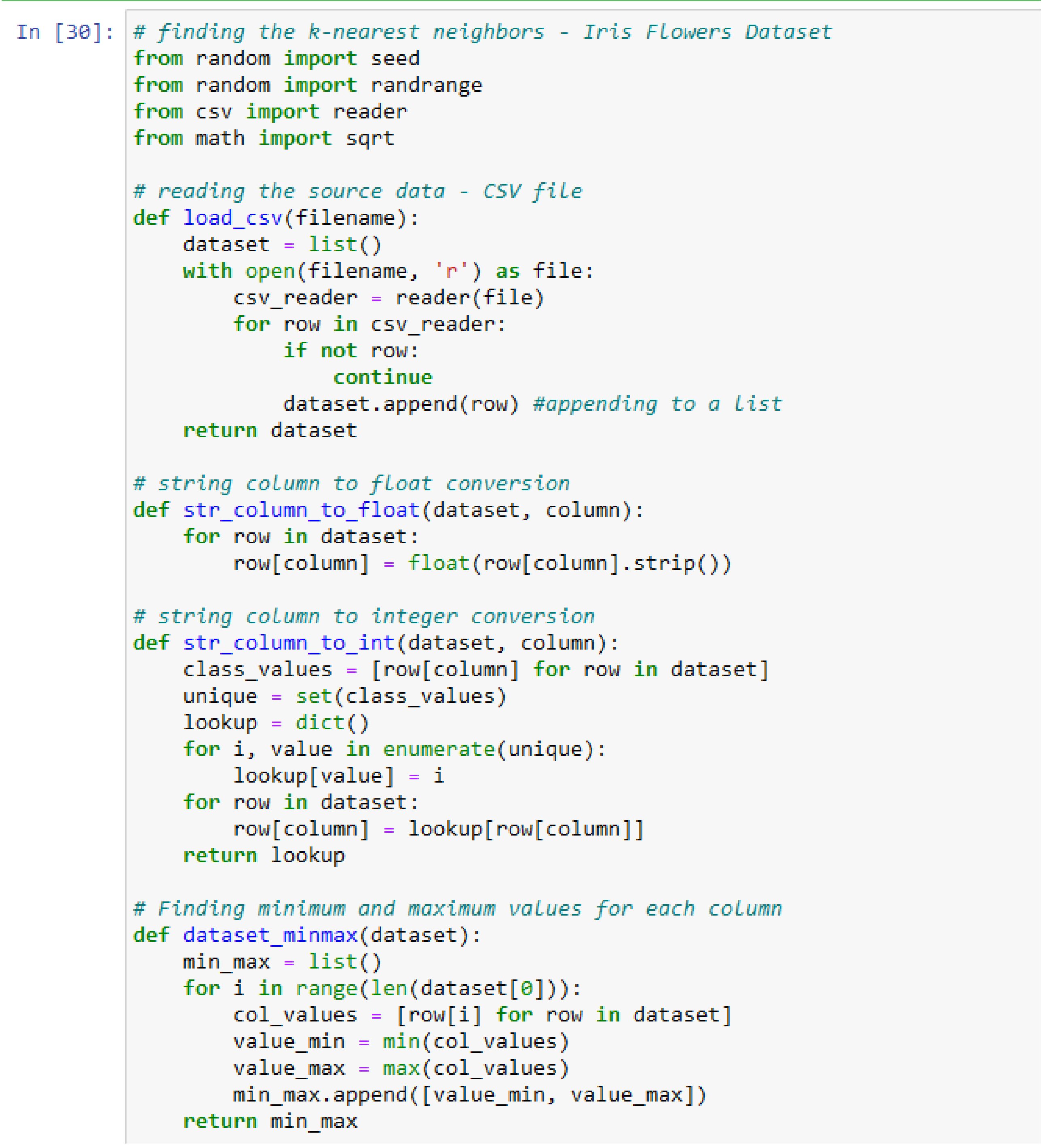
Case study 1: Applying KNN algorithm to Iris flower species dataset.

Data source: https://www.kaggle.com/code/skalskip/iris-data-visualization-and-knn-classification

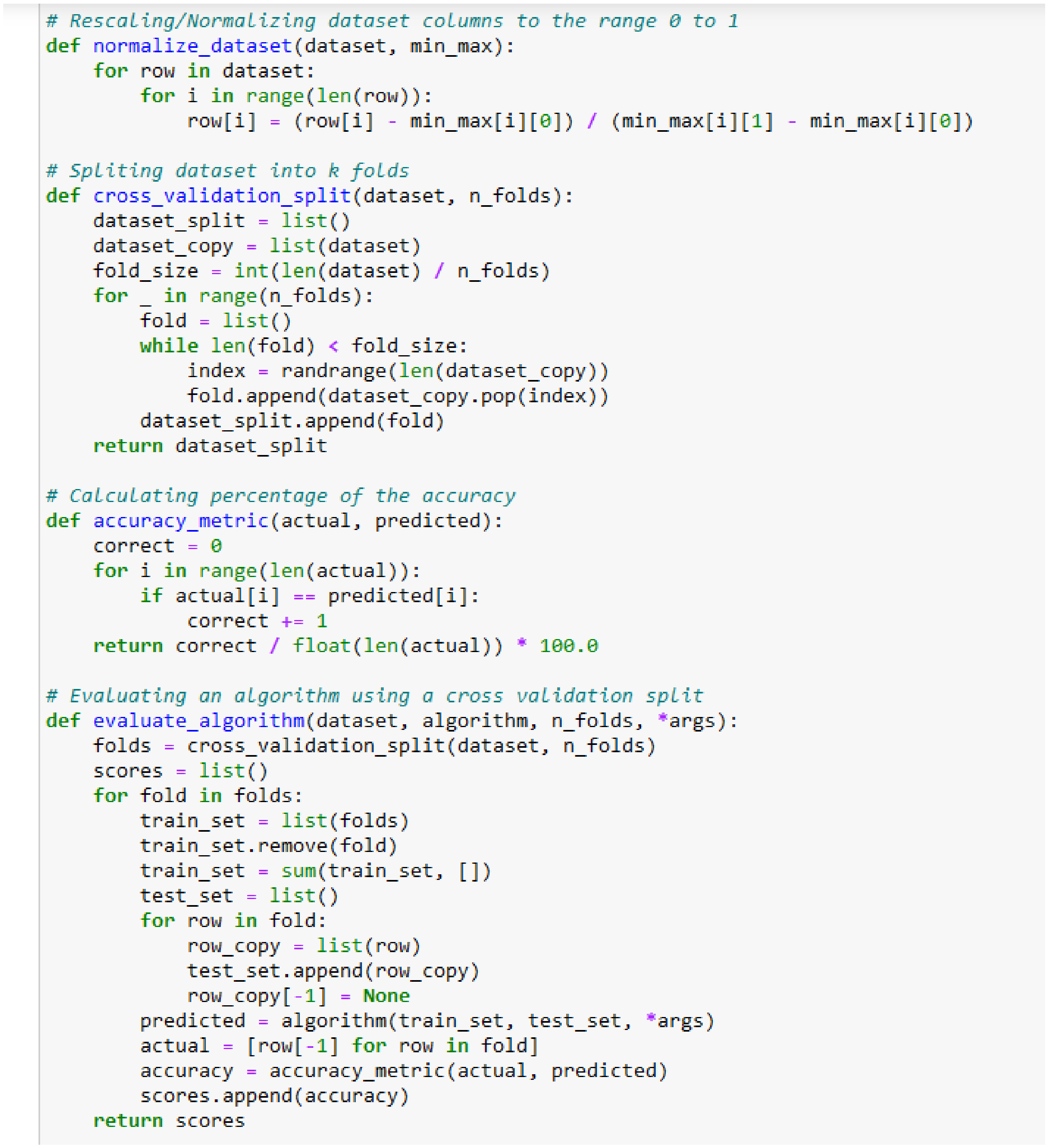
Steps:

- Load the dataset and transform the data into numbers so that we can compute the standard deviation and mean.
- Use five folds of k-fold cross-validation to assess the method.
- The helper functions accuracy_metric() and evaluate_algorithm() compute the prediction accuracy and algorithm evaluation using cross-validation, respectively.

- K_nearest_neighbors() was created to control how the KNN algorithm is applied.
- Acquiring knowledge of the statistics from a training dataset and applying it to forecast data from a test dataset.

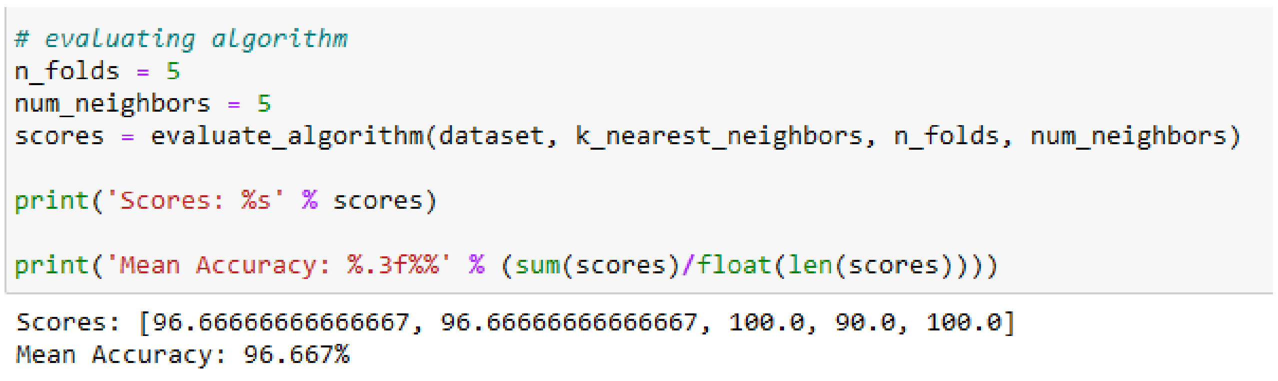


```
In [30]: # finding the k-nearest neighbors - Iris Flowers Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# reading the source data - CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row) #appending to a list
    return dataset

# string column to float conversion
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# string column to integer conversion
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup

# Finding minimum and maximum values for each column
def dataset_minmax(dataset):
    min_max = list()
    for i in range(len(dataset[0])):
        col_values = [row[i] for row in dataset]
        value_min = min(col_values)
        value_max = max(col_values)
        min_max.append([value_min, value_max])
    return min_max
```



```python
# Rescaling/Normalizing dataset columns to the range 0 to 1
def normalize_dataset(dataset, min_max):
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - min_max[i][0]) / (min_max[i][1] - min_max[i][0])

# Spliting dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for _ in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

# Calculating percentage of the accuracy
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluating an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores
```

```python
# Calculating the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

# Finding most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors

# Making a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction

# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
    predictions = list()
    for row in test:
        output = predict_classification(train, row, num_neighbors)
        predictions.append(output)
    return(predictions)

# Testing the kNN method on the Iris Flowers dataset
seed(1)
filename = 'D:\iris.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)

# converting class column to integer values
str_column_to_int(dataset, len(dataset[0])-1)
```



```
# evaluating algorithm
n_folds = 5
num_neighbors = 5
scores = evaluate_algorithm(dataset, k_nearest_neighbors, n_folds, num_neighbors)

print('Scores: %s' % scores)

print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
Scores: [96.66666666666667, 96.66666666666667, 100.0, 90.0, 100.0]
Mean Accuracy: 96.667%
```

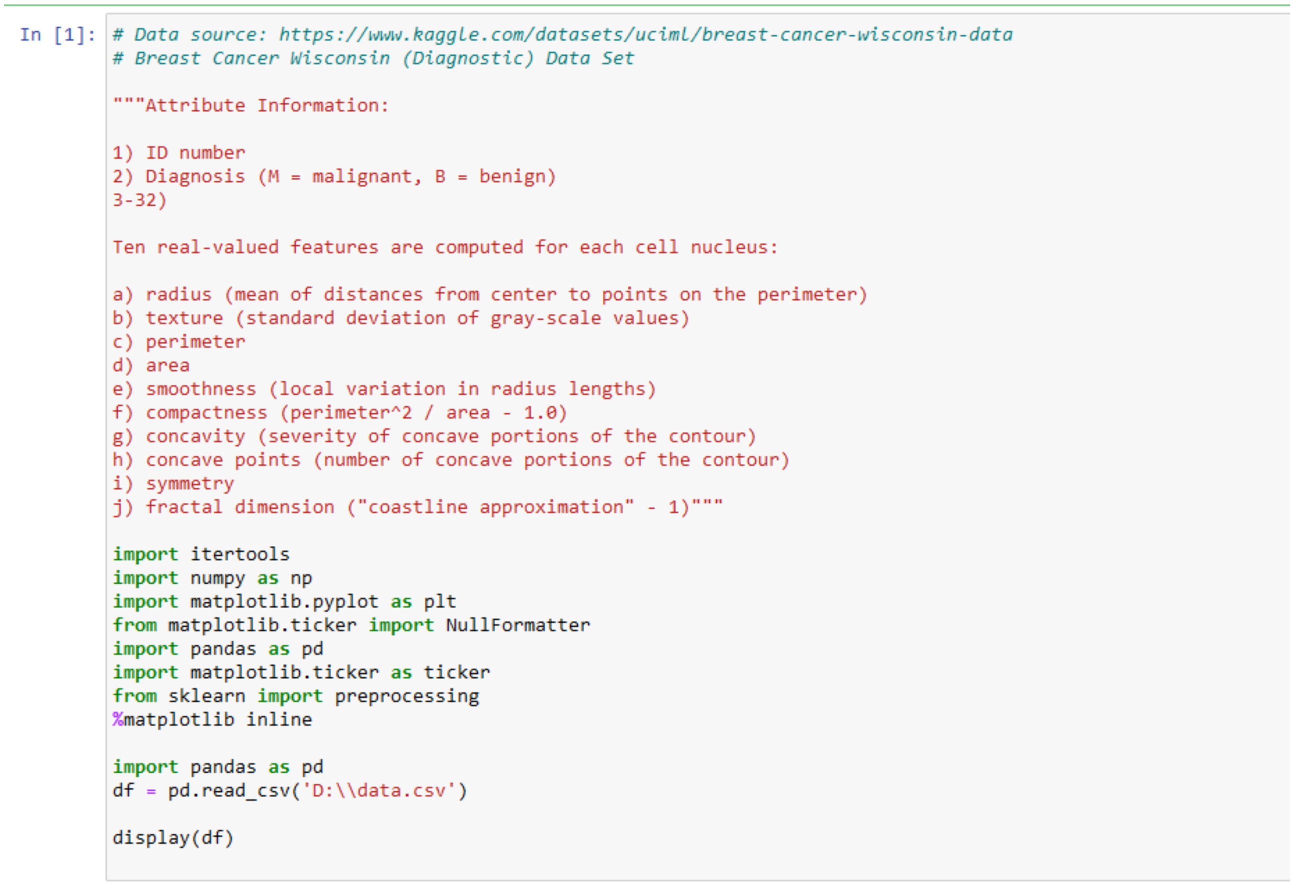
It is observed that the mean accuracy of this model is 96.667%

The optimal K value: it is the square root of N, where N is the total number of samples. It is possible to determine the most optimal K value by using an accuracy or error plot.

Case Study 2: Applying KNN algorithm to Breast Cancer Wisconsin (Diagnostic) Data Set.

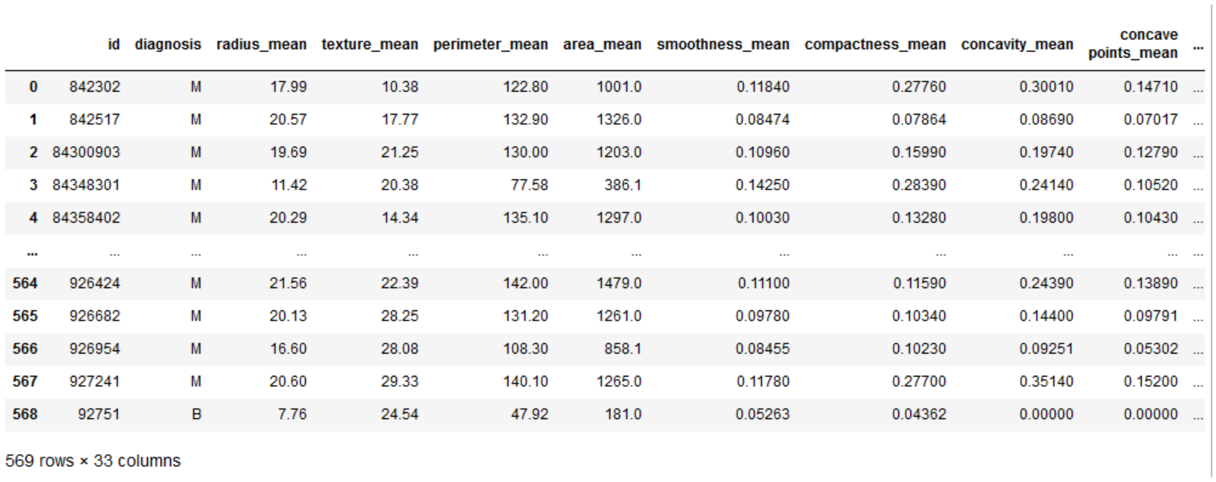
Data source: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data/data

A digital image of a fine needle aspirate (FNA) of a breast mass is used to compute features.



```
In [1]: # Data source: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data
        # Breast Cancer Wisconsin (Diagnostic) Data Set

        """Attribute Information:

        1) ID number
        2) Diagnosis (M = malignant, B = benign)
        3-32)

        Ten real-valued features are computed for each cell nucleus:

        a) radius (mean of distances from center to points on the perimeter)
        b) texture (standard deviation of gray-scale values)
        c) perimeter
        d) area
        e) smoothness (local variation in radius lengths)
        f) compactness (perimeter^2 / area - 1.0)
        g) concavity (severity of concave portions of the contour)
        h) concave points (number of concave portions of the contour)
        i) symmetry
        j) fractal dimension ("coastline approximation" - 1)"""

        import itertools
        import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.ticker import NullFormatter
        import pandas as pd
        import matplotlib.ticker as ticker
        from sklearn import preprocessing
        %matplotlib inline

        import pandas as pd
        df = pd.read_csv('D:\\data.csv')

        display(df)
```

Displaying the dataset.



	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	...
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	...
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	...
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	...
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	...
...	...	...	...	...	...	...	...	...	...	...	...
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	...
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	...
566	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	...
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	...
568	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	...

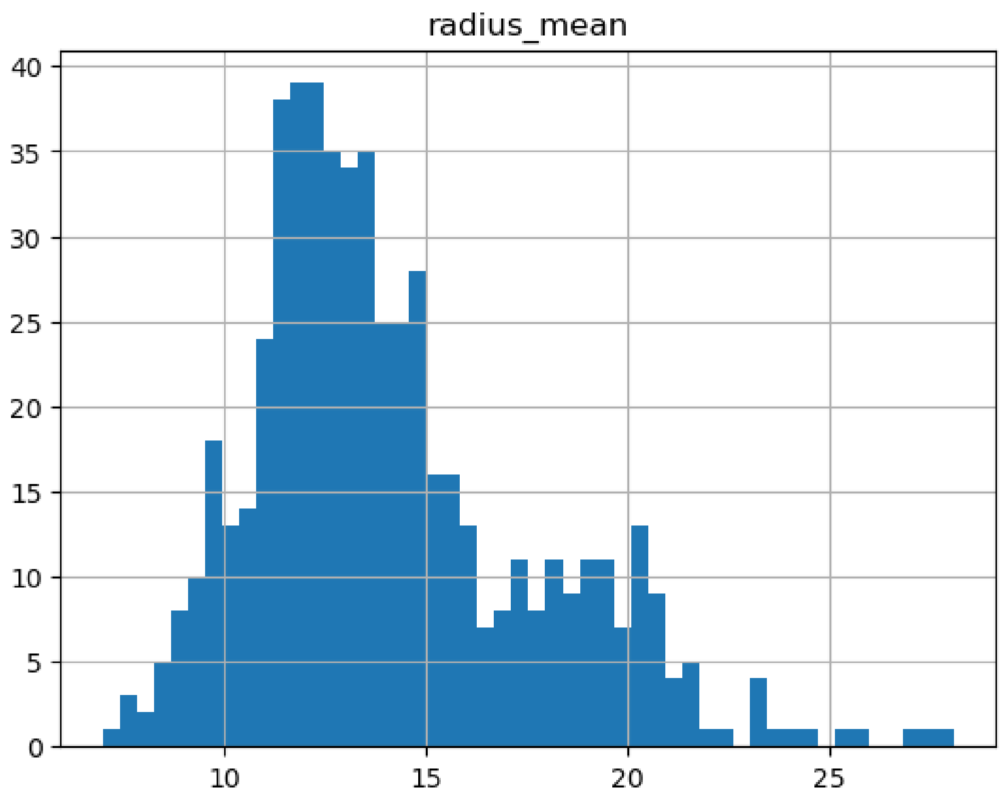
569 rows × 33 columns

Finding how many diagnosis records under each class.

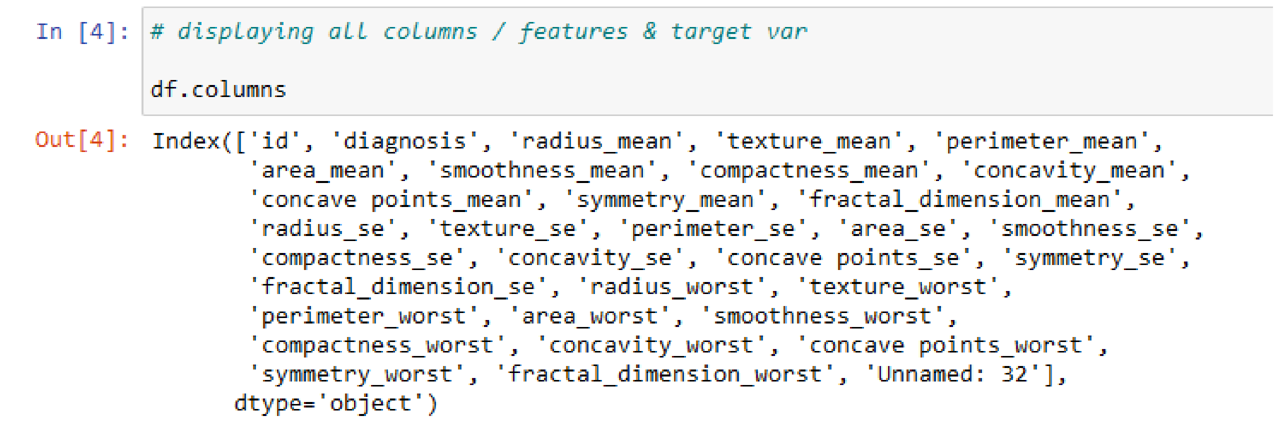
```
In [2]: # each class (M = malignant, B = benign) how many records (Diagnosis)

        df['diagnosis'].value_counts()

Out[2]: B    357
        M    212
        Name: diagnosis, dtype: int64
```

Visualizing data using the radius_mean column just to check data.

```
In [3]: # visualising data
        # use the feature radius_mean and generating a histogram chart

        df.hist(column='radius_mean', bins = 50)

Out[3]: array([[<Axes: title={'center': 'radius_mean'}>]], dtype=object)
```

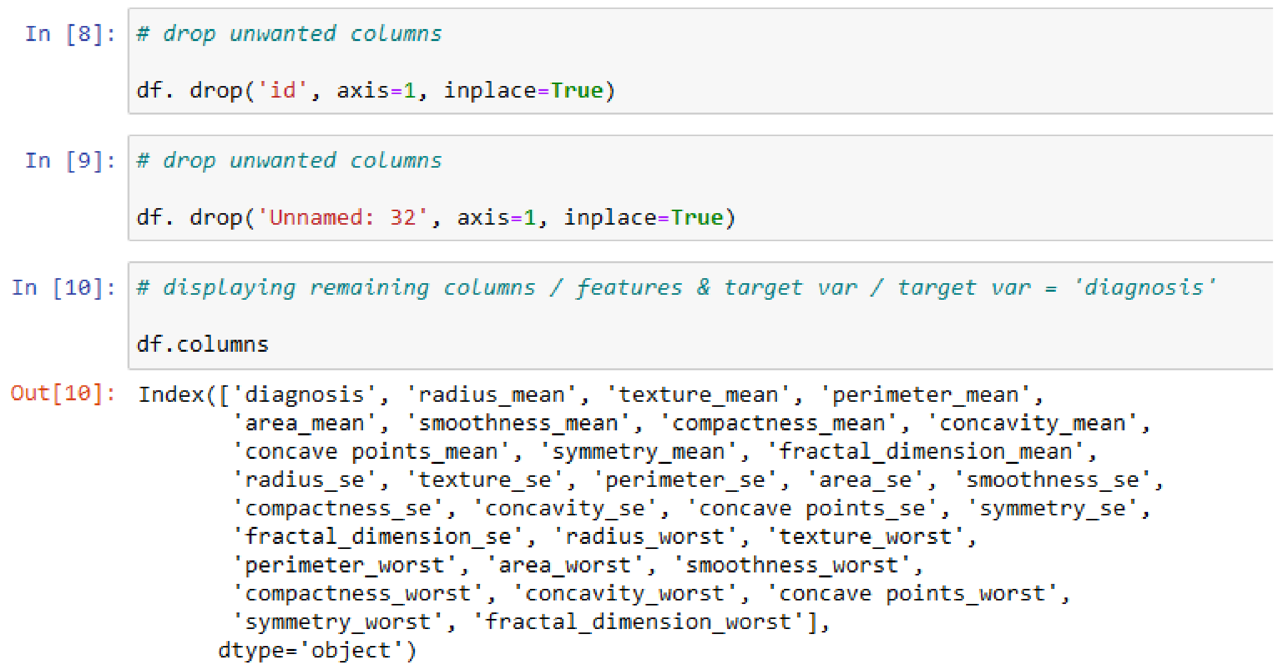


Viewing all columns in the dataset.



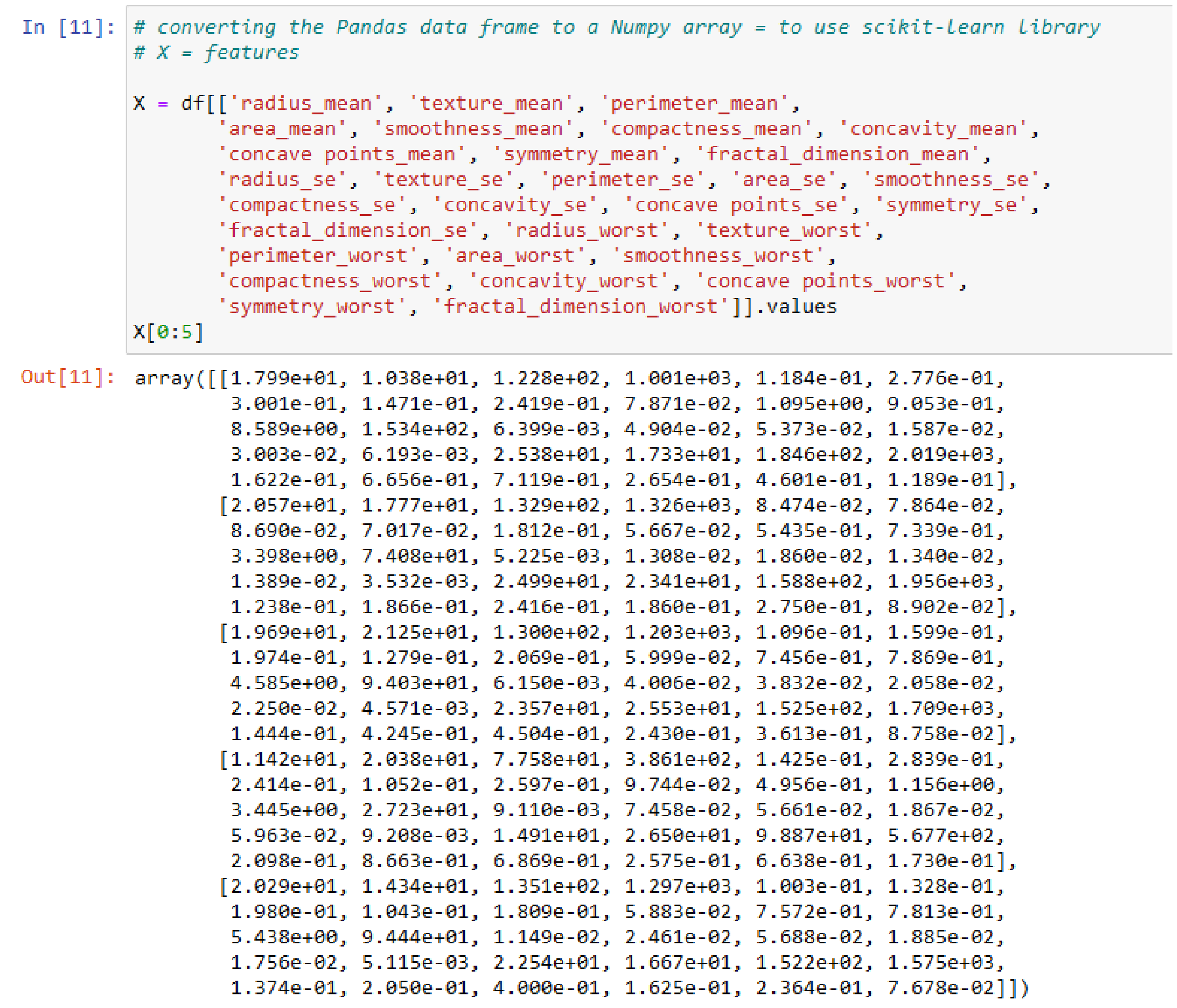
```
In [4]: # displaying all columns / features & target var

        df.columns

Out[4]: Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
               'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
               'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
               'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
               'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
               'fractal_dimension_se', 'radius_worst', 'texture_worst',
               'perimeter_worst', 'area_worst', 'smoothness_worst',
               'compactness_worst', 'concavity_worst', 'concave points_worst',
               'symmetry_worst', 'fractal_dimension_worst', 'Unnamed: 32'],
              dtype='object')
```

Dropping unwanted columns.



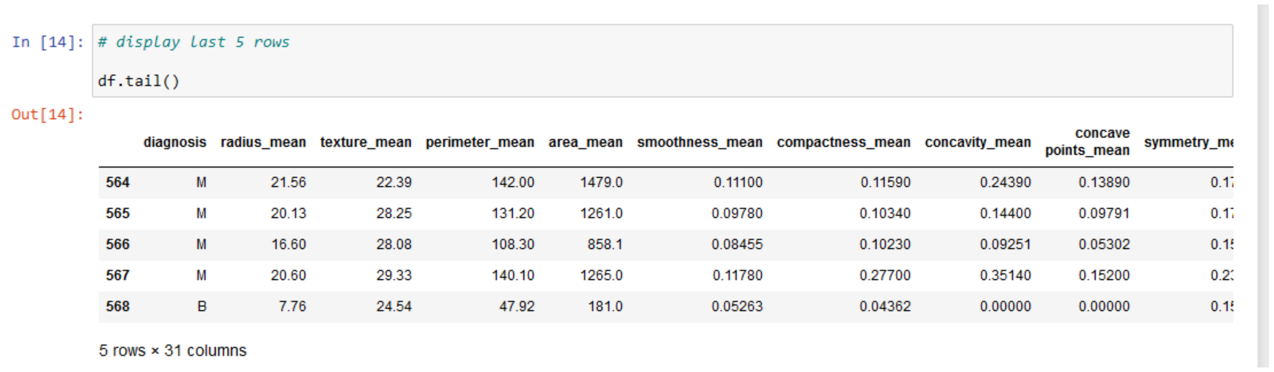
```
In [8]: # drop unwanted columns

        df. drop('id', axis=1, inplace=True)

In [9]: # drop unwanted columns

        df. drop('Unnamed: 32', axis=1, inplace=True)

In [10]: # displaying remaining columns / features & target var / target var = 'diagnosis'

         df.columns

Out[10]: Index(['diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
                'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
                'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
                'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
                'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
                'fractal_dimension_se', 'radius_worst', 'texture_worst',
                'perimeter_worst', 'area_worst', 'smoothness_worst',
                'compactness_worst', 'concavity_worst', 'concave points_worst',
                'symmetry_worst', 'fractal_dimension_worst'],
               dtype='object')
```

Converting the dataset to Numpy array.



```
In [11]: # converting the Pandas data frame to a Numpy array = to use scikit-learn library
         # X = features

         X = df[['radius_mean', 'texture_mean', 'perimeter_mean',
                 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
                 'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
                 'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
                 'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
                 'fractal_dimension_se', 'radius_worst', 'texture_worst',
                 'perimeter_worst', 'area_worst', 'smoothness_worst',
                 'compactness_worst', 'concavity_worst', 'concave points_worst',
                 'symmetry_worst', 'fractal_dimension_worst']].values
         X[0:5]

Out[11]: array([[1.799e+01, 1.038e+01, 1.228e+02, 1.001e+03, 1.184e-01, 2.776e-01,
                 3.001e-01, 1.471e-01, 2.419e-01, 7.871e-02, 1.095e+00, 9.053e-01,
                 8.589e+00, 1.534e+02, 6.399e-03, 4.904e-02, 5.373e-02, 1.587e-02,
                 3.003e-02, 6.193e-03, 2.538e+01, 1.733e+01, 1.846e+02, 2.019e+03,
                 1.622e-01, 6.656e-01, 7.119e-01, 2.654e-01, 4.601e-01, 1.189e-01],
                [2.057e+01, 1.777e+01, 1.329e+02, 1.326e+03, 8.474e-02, 7.864e-02,
                 8.690e-02, 7.017e-02, 1.812e-01, 5.667e-02, 5.435e-01, 7.339e-01,
                 3.398e+00, 7.408e+01, 5.225e-03, 1.308e-02, 1.860e-02, 1.340e-02,
                 1.389e-02, 3.532e-03, 2.499e+01, 2.341e+01, 1.588e+02, 1.956e+03,
                 1.238e-01, 1.866e-01, 2.416e-01, 1.860e-01, 2.750e-01, 8.902e-02],
                [1.969e+01, 2.125e+01, 1.300e+02, 1.203e+03, 1.096e-01, 1.599e-01,
                 1.974e-01, 1.279e-01, 2.069e-01, 5.999e-02, 7.456e-01, 7.869e-01,
                 4.585e+00, 9.403e+01, 6.150e-03, 4.006e-02, 3.832e-02, 2.058e-02,
                 2.250e-02, 4.571e-03, 2.357e+01, 2.553e+01, 1.525e+02, 1.709e+03,
                 1.444e-01, 4.245e-01, 4.504e-01, 2.430e-01, 3.613e-01, 8.758e-02],
                [1.142e+01, 2.038e+01, 7.758e+01, 3.861e+02, 1.425e-01, 2.839e-01,
                 2.414e-01, 1.052e-01, 2.597e-01, 9.744e-02, 4.956e-01, 1.156e+00,
                 3.445e+00, 2.723e+01, 9.110e-03, 7.458e-02, 5.661e-02, 1.867e-02,
                 5.963e-02, 9.208e-03, 1.491e+01, 2.650e+01, 9.887e+01, 5.677e+02,
                 2.098e-01, 8.663e-01, 6.869e-01, 2.575e-01, 6.638e-01, 1.730e-01],
                [2.029e+01, 1.434e+01, 1.351e+02, 1.297e+03, 1.003e-01, 1.328e-01,
                 1.980e-01, 1.043e-01, 1.809e-01, 5.883e-02, 7.572e-01, 7.813e-01,
                 5.438e+00, 9.444e+01, 1.149e-02, 2.461e-02, 5.688e-02, 1.885e-02,
                 1.756e-02, 5.115e-03, 2.254e+01, 1.667e+01, 1.522e+02, 1.575e+03,
                 1.374e-01, 2.050e-01, 4.000e-01, 1.625e-01, 2.364e-01, 7.678e-02]])
```

Displaying the last 5 rows.

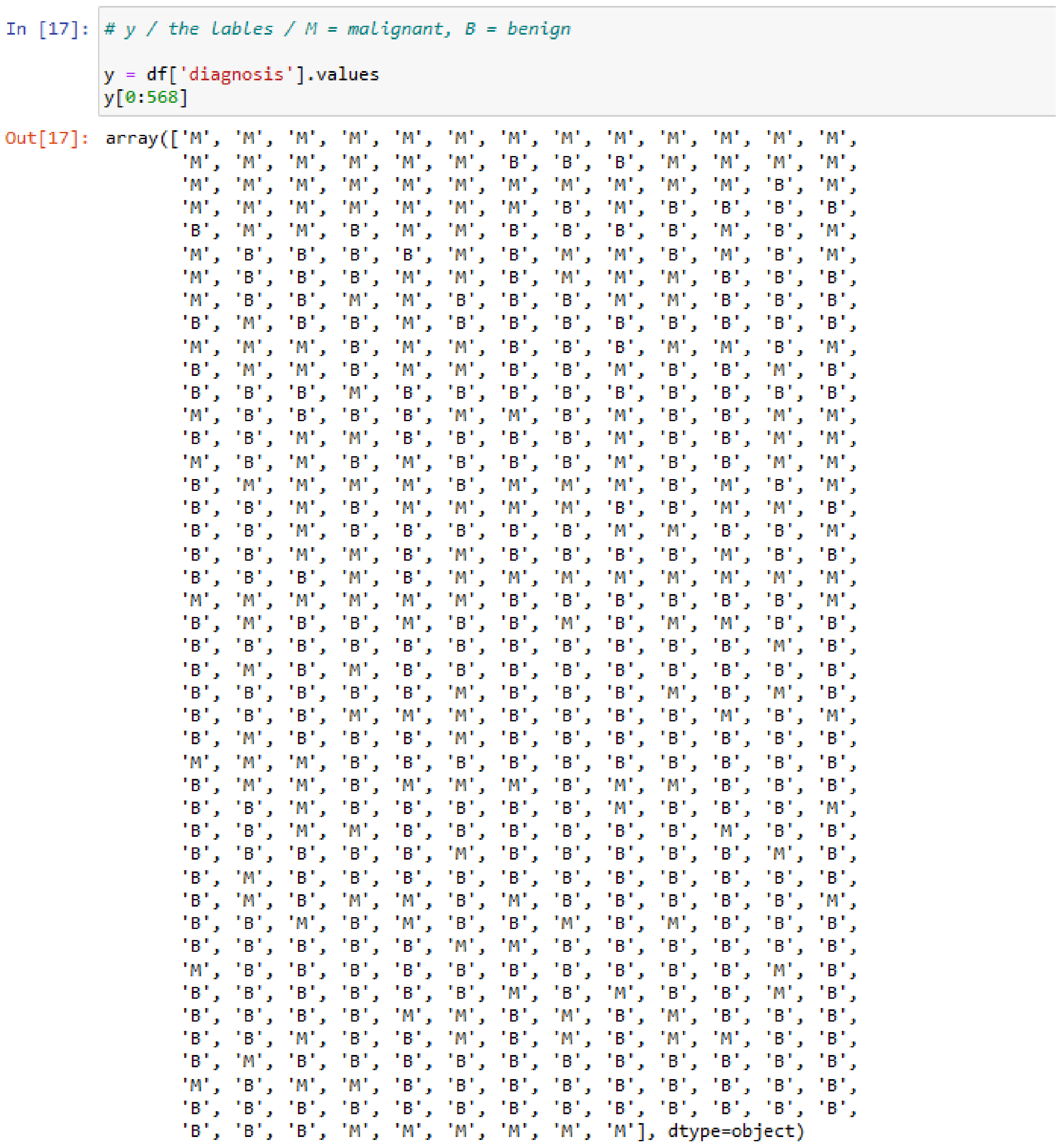


```
In [14]: # display last 5 rows

         df.tail()

Out[14]:
```

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_me
564	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1
565	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1
566	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1
567	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2
568	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1

5 rows × 31 columns

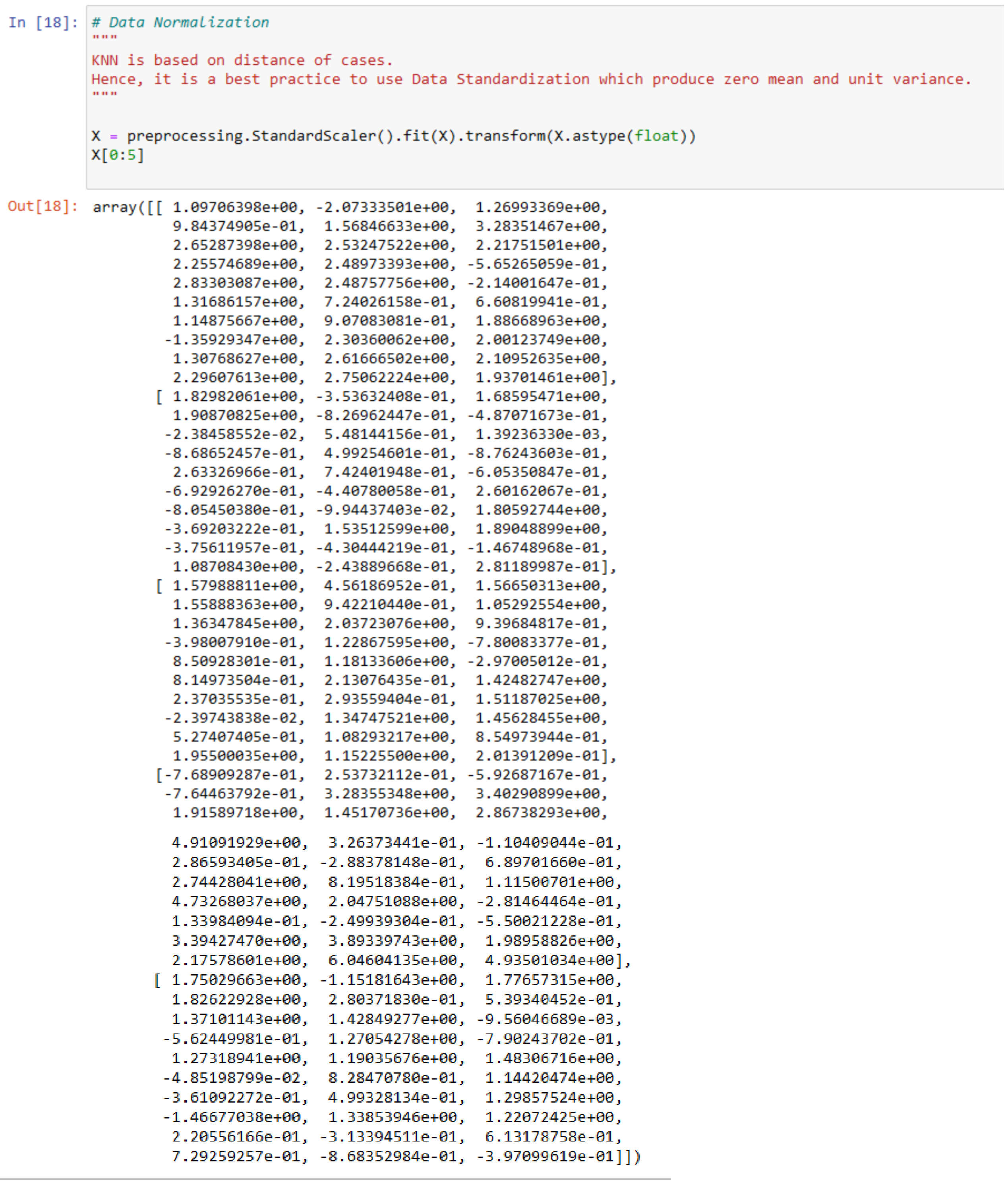
Displaying only the labels under the diagnosis attribute.



```
In [17]: # y / the lables / M = malignant, B = benign

         y = df['diagnosis'].values
         y[0:568]
```

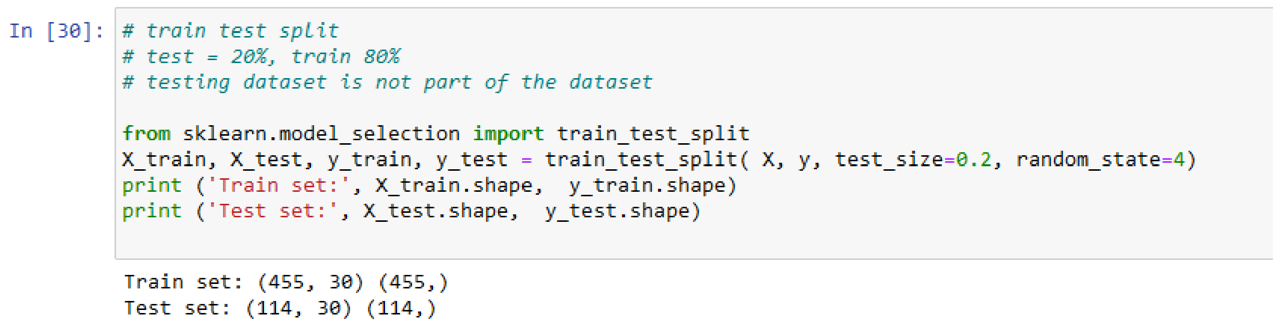
```
Out[17]: array(['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
                'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'M',
                'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M',
                'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B',
                'B', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M',
                'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'M',
                'M', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'B', 'B',
                'M', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B',
                'B', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'M', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'M',
                'B', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'M', 'B',
                'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'M', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'M', 'M',
                'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'M',
                'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'M',
                'B', 'M', 'M', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'M', 'B', 'M',
                'B', 'B', 'M', 'B', 'M', 'M', 'M', 'M', 'B', 'B', 'M', 'M', 'B',
                'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'M',
                'B', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
                'B', 'B', 'B', 'M', 'B', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
                'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M',
                'B', 'M', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
                'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
                'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B',
                'B', 'B', 'B', 'M', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M',
                'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'M', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'B', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B',
                'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M',
                'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
                'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
                'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M',
                'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'B',
                'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B',
                'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
                'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B',
                'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B',
                'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
                'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
                'B', 'B', 'B', 'M', 'M', 'M', 'M', 'M', 'M'], dtype=object)
```

Normalizing the data.



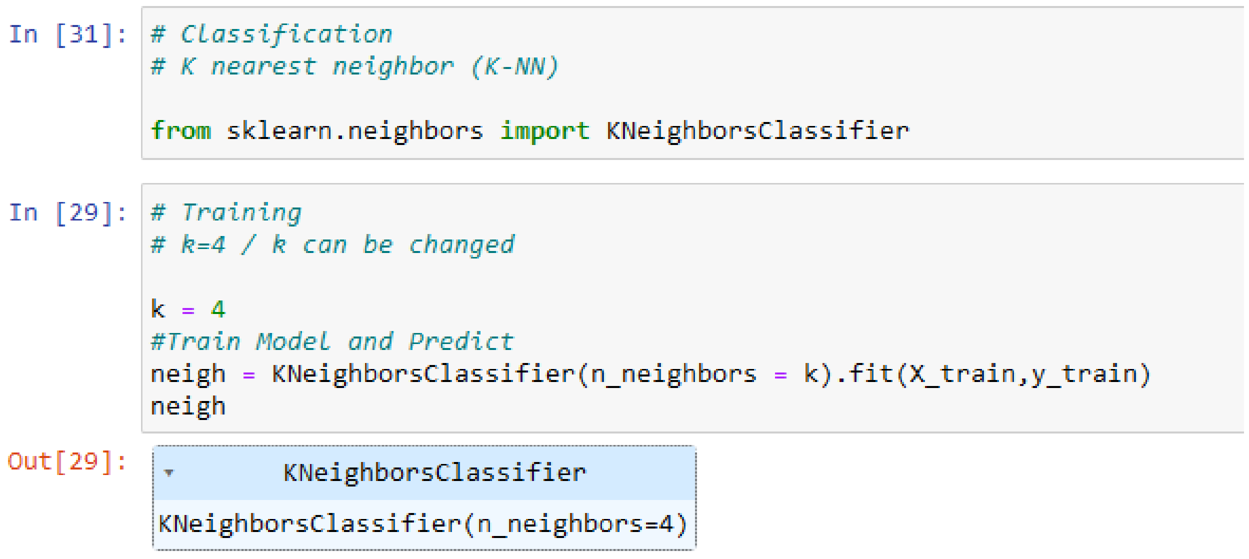
```
In [18]: # Data Normalization
         """
         KNN is based on distance of cases.
         Hence, it is a best practice to use Data Standardization which produce zero mean and unit variance.
         """

         X = preprocessing.StandardScaler().fit(X).transform(X.astype(float))
         X[0:5]

Out[18]: array([[ 1.09706398e+00, -2.07333501e+00,  1.26993369e+00,
                  9.84374905e-01,  1.56846633e+00,  3.28351467e+00,
                  2.65287398e+00,  2.53247522e+00,  2.21751501e+00,
                  2.25574689e+00,  2.48973393e+00, -5.65265059e-01,
                  2.83303087e+00,  2.48757756e+00, -2.14001647e-01,
                  1.31686157e+00,  7.24026158e-01,  6.60819941e-01,
                  1.14875667e+00,  9.07083081e-01,  1.88668963e+00,
                 -1.35929347e+00,  2.30360062e+00,  2.00123749e+00,
                  1.30768627e+00,  2.61666502e+00,  2.10952635e+00,
                  2.29607613e+00,  2.75062224e+00,  1.93701461e+00],
                [ 1.82982061e+00, -3.53632408e-01,  1.68595471e+00,
                  1.90870825e+00, -8.26962447e-01, -4.87071673e-01,
                 -2.38458552e-02,  5.48144156e-01,  1.39236330e-03,
                 -8.68652457e-01,  4.99254601e-01, -8.76243603e-01,
                  2.63326966e-01,  7.42401948e-01, -6.05350847e-01,
                 -6.92926270e-01, -4.40780058e-01,  2.60162067e-01,
                 -8.05450380e-01, -9.94437403e-02,  1.80592744e+00,
                 -3.69203222e-01,  1.53512599e+00,  1.89048899e+00,
                 -3.75611957e-01, -4.30444219e-01, -1.46748968e-01,
                  1.08708430e+00, -2.43889668e-01,  2.81189987e-01],
                [ 1.57988811e+00,  4.56186952e-01,  1.56650313e+00,
                  1.55888363e+00,  9.42210440e-01,  1.05292554e+00,
                  1.36347845e+00,  2.03723076e+00,  9.39684817e-01,
                 -3.98007910e-01,  1.22867595e+00, -7.80083377e-01,
                  8.50928301e-01,  1.18133606e+00, -2.97005012e-01,
                  8.14973504e-01,  2.13076435e-01,  1.42482747e+00,
                  2.37035535e-01,  2.93559404e-01,  1.51187025e+00,
                 -2.39743838e-02,  1.34747521e+00,  1.45628455e+00,
                  5.27407405e-01,  1.08293217e+00,  8.54973944e-01,
                  1.95500035e+00,  1.15225500e+00,  2.01391209e-01],
                [-7.68909287e-01,  2.53732112e-01, -5.92687167e-01,
                 -7.64463792e-01,  3.28355348e+00,  3.40290899e+00,
                  1.91589718e+00,  1.45170736e+00,  2.86738293e+00,

                  4.91091929e+00,  3.26373441e-01, -1.10409044e-01,
                  2.86593405e-01, -2.88378148e-01,  6.89701660e-01,
                  2.74428041e+00,  8.19518384e-01,  1.11500701e+00,
                  4.73268037e+00,  2.04751088e+00, -2.81464464e-01,
                  1.33984094e-01, -2.49939304e-01, -5.50021228e-01,
                  3.39427470e+00,  3.89339743e+00,  1.98958826e+00,
                  2.17578601e+00,  6.04604135e+00,  4.93501034e+00],
                [ 1.75029663e+00, -1.15181643e+00,  1.77657315e+00,
                  1.82622928e+00,  2.80371830e-01,  5.39340452e-01,
                  1.37101143e+00,  1.42849277e+00, -9.56046689e-03,
                 -5.62449981e-01,  1.27054278e+00, -7.90243702e-01,
                  1.27318941e+00,  1.19035676e+00,  1.48306716e+00,
                 -4.85198799e-02,  8.28470780e-01,  1.14420474e+00,
                 -3.61092272e-01,  4.99328134e-01,  1.29857524e+00,
                 -1.46677038e+00,  1.33853946e+00,  1.22072425e+00,
                  2.20556166e-01, -3.13394511e-01,  6.13178758e-01,
                  7.29259257e-01, -8.68352984e-01, -3.97099619e-01]])
```

Separating the data as train and test sets.



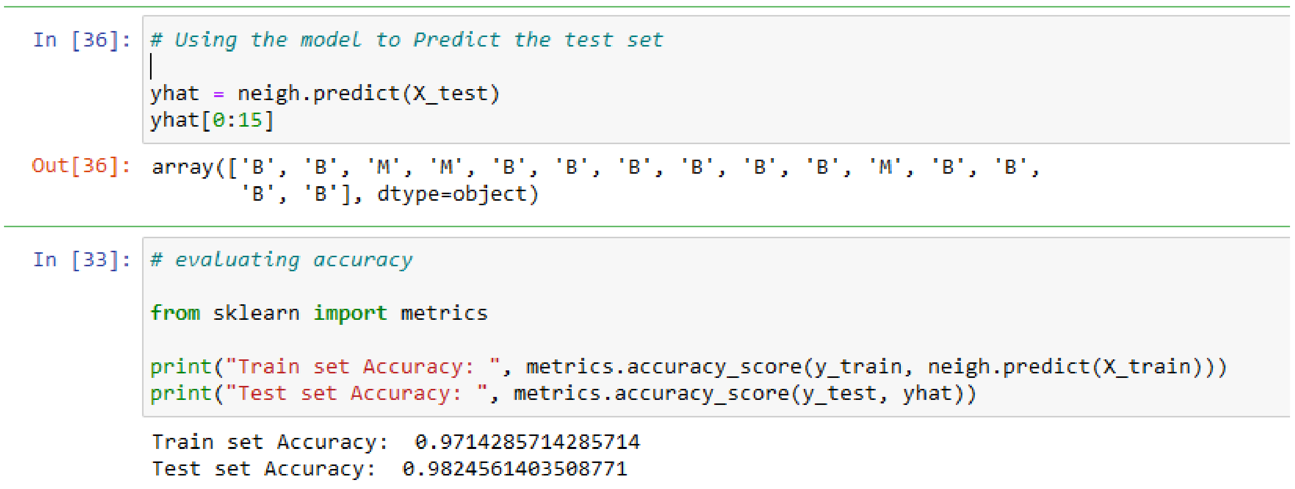
```
In [30]: # train test split
         # test = 20%, train 80%
         # testing dataset is not part of the dataset

         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
         print ('Train set:', X_train.shape,  y_train.shape)
         print ('Test set:', X_test.shape,  y_test.shape)

         Train set: (455, 30) (455,)
         Test set: (114, 30) (114,)
```

Using k-NN classifier.



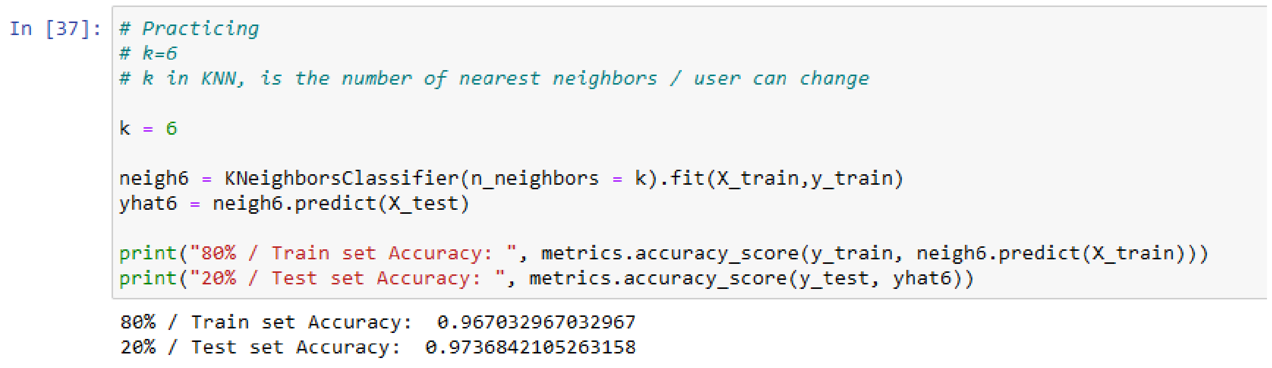
```
In [31]: # Classification
         # K nearest neighbor (K-NN)

         from sklearn.neighbors import KNeighborsClassifier

In [29]: # Training
         # k=4 / k can be changed

         k = 4
         #Train Model and Predict
         neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
         neigh

Out[29]: ▾          KNeighborsClassifier
         KNeighborsClassifier(n_neighbors=4)
```

Using the model to predict the test set.



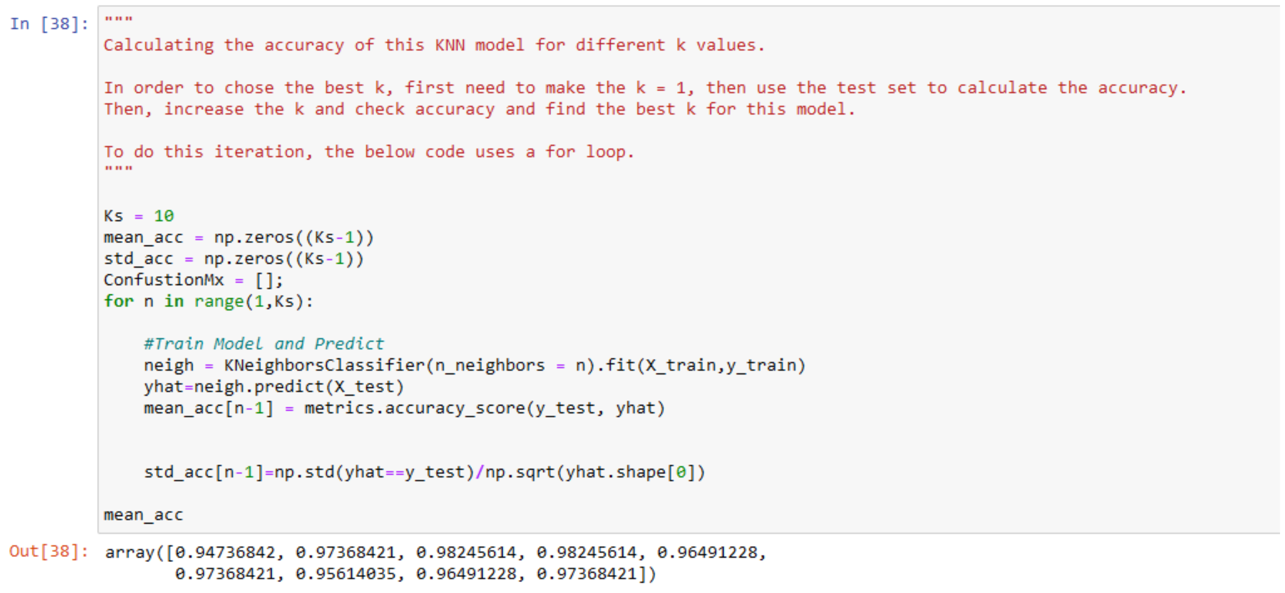
```
In [36]: # Using the model to Predict the test set
         |
         yhat = neigh.predict(X_test)
         yhat[0:15]

Out[36]: array(['B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
                'B', 'B'], dtype=object)

In [33]: # evaluating accuracy

         from sklearn import metrics

         print("Train set Accuracy: ", metrics.accuracy_score(y_train, neigh.predict(X_train)))
         print("Test set Accuracy: ", metrics.accuracy_score(y_test, yhat))

         Train set Accuracy:  0.9714285714285714
         Test set Accuracy:  0.9824561403508771
```

Practicing using k = 6 (the number of neighbors is 6)



```
In [37]: # Practicing
         # k=6
         # k in KNN, is the number of nearest neighbors / user can change

         k = 6

         neigh6 = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
         yhat6 = neigh6.predict(X_test)

         print("80% / Train set Accuracy: ", metrics.accuracy_score(y_train, neigh6.predict(X_train)))
         print("20% / Test set Accuracy: ", metrics.accuracy_score(y_test, yhat6))

         80% / Train set Accuracy:  0.967032967032967
         20% / Test set Accuracy:  0.9736842105263158
```

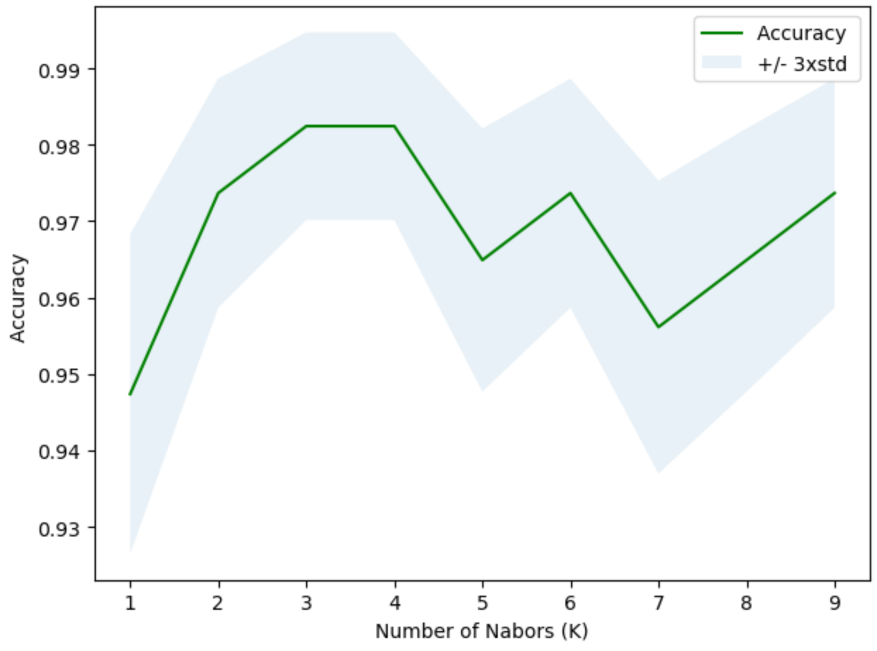
We must determine the ideal accuracy value by examining the effects of various k values on the classifier's performance, or accuracy.



```
In [38]: """
         Calculating the accuracy of this KNN model for different k values.

         In order to chose the best k, first need to make the k = 1, then use the test set to calculate the accuracy.
         Then, increase the k and check accuracy and find the best k for this model.

         To do this iteration, the below code uses a for loop.
         """

         Ks = 10
         mean_acc = np.zeros((Ks-1))
         std_acc = np.zeros((Ks-1))
         ConfustionMx = [];
         for n in range(1,Ks):

             #Train Model and Predict
             neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train,y_train)
             yhat=neigh.predict(X_test)
             mean_acc[n-1] = metrics.accuracy_score(y_test, yhat)


             std_acc[n-1]=np.std(yhat==y_test)/np.sqrt(yhat.shape[0])

         mean_acc

Out[38]: array([0.94736842, 0.97368421, 0.98245614, 0.98245614, 0.96491228,
                0.97368421, 0.95614035, 0.96491228, 0.97368421])
```

Finding the best k using a diagram.

```
In [26]: # generate a diagram to visualize the best k value

         plt.plot(range(1,Ks),mean_acc,'g')
         plt.fill_between(range(1,Ks),mean_acc - 1 * std_acc,mean_acc + 1 * std_acc, alpha=0.10)
         plt.legend(('Accuracy ', '+/- 3xstd'))
         plt.ylabel('Accuracy ')
         plt.xlabel('Number of Nabors (K)')
         plt.tight_layout()
         plt.show()
```



Viewing the best k and the accuracy of the model.

```
In [39]: # the value of the best accuracy + it's k value

         print( "The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1)

         The best accuracy was with 0.9824561403508771 with k= 3
```

This model’s accuracy is 98.24% and its k value is 3.

2. Apply the k-NN classifiers to a dataset.

Choose a suitable dataset from the “KNN Datasets” folder with numerical or categorical features and a target variable. Apply your k-NN classifier implementations from Sub-question 1 to this dataset.

2.1. Preprocess the dataset, including handling missing values, encoding categorical features (if necessary), and splitting the data into training and testing sets.

2.2. Train your k-NN classifiers using the training set and predict the classes for the testing set. Calculate the accuracy of your classifiers on the testing set.

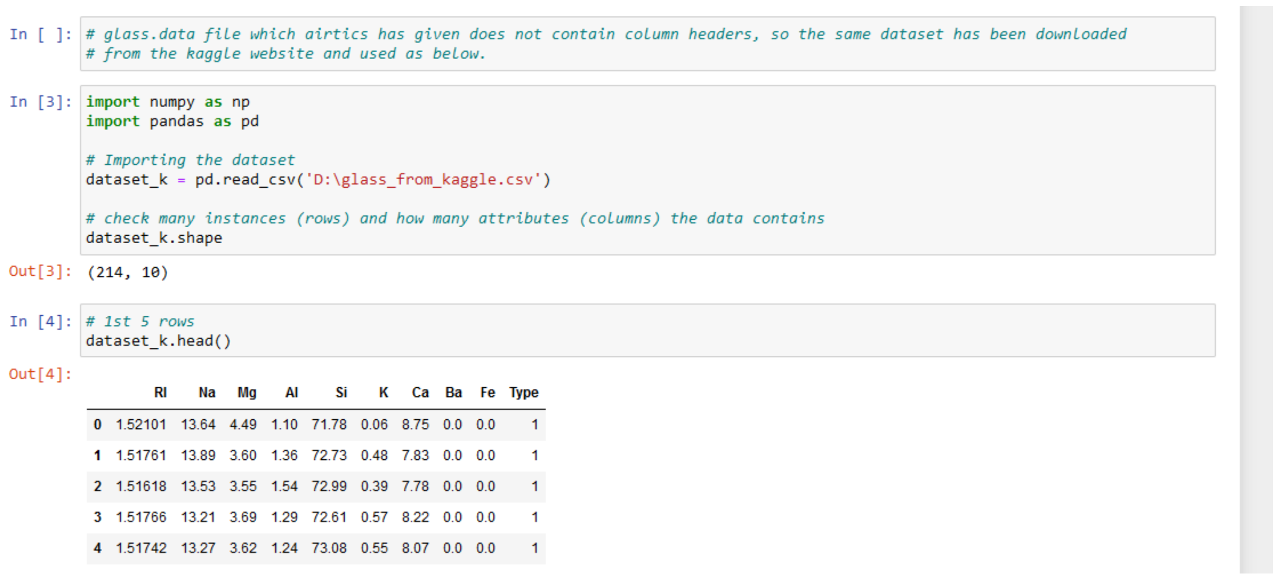
2.3. Experiment with different values of k for the k-NN classifier and discuss how the choice of k affects the classifier's performance. Choose an optimal value of k based on your experiments.

Please provide a brief explanation of the dataset you chose and any preprocessing steps you performed.

Note that for k-NN classifiers, the training process is minimal, as it only involves storing the training instances. The classification is done during the prediction step by finding the nearest neighbor(s) for each test instance.

Displaying the number of rows and columns of the glass dataset.

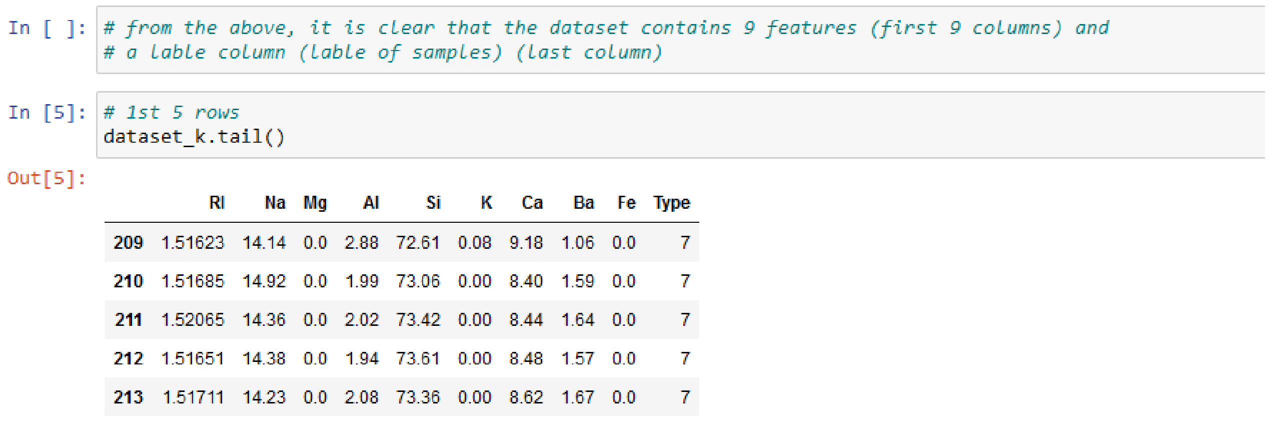
Viewing the 1st five rows of the glass dataset.



```
In [ ]: # glass.data file which airtics has given does not contain column headers, so the same dataset has been downloaded
        # from the kaggle website and used as below.

In [3]: import numpy as np
        import pandas as pd

        # Importing the dataset
        dataset_k = pd.read_csv('D:\glass_from_kaggle.csv')

        # check many instances (rows) and how many attributes (columns) the data contains
        dataset_k.shape

Out[3]: (214, 10)

In [4]: # 1st 5 rows
        dataset_k.head()

Out[4]:
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

Displaying the last five rows of the glass dataset and finding basic details.



```
In [ ]: # from the above, it is clear that the dataset contains 9 features (first 9 columns) and
        # a lable column (lable of samples) (last column)

In [5]: # 1st 5 rows
        dataset_k.tail()

Out[5]:
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
209	1.51623	14.14	0.0	2.88	72.61	0.08	9.18	1.06	0.0	7
210	1.51685	14.92	0.0	1.99	73.06	0.00	8.40	1.59	0.0	7
211	1.52065	14.36	0.0	2.02	73.42	0.00	8.44	1.64	0.0	7
212	1.51651	14.38	0.0	1.94	73.61	0.00	8.48	1.57	0.0	7
213	1.51711	14.23	0.0	2.08	73.36	0.00	8.62	1.67	0.0	7

In [6]:
```
# get common statistics
dataset_k.describe()
```

Out[6]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000
mean	1.518365	13.407850	2.684533	1.444907	72.650935	0.497056	8.956963	0.175047	0.057009	2.780374
std	0.003037	0.816604	1.442408	0.499270	0.774546	0.652192	1.423153	0.497219	0.097439	2.103739
min	1.511150	10.730000	0.000000	0.290000	69.810000	0.000000	5.430000	0.000000	0.000000	1.000000
25%	1.516522	12.907500	2.115000	1.190000	72.280000	0.122500	8.240000	0.000000	0.000000	1.000000
50%	1.517680	13.300000	3.480000	1.360000	72.790000	0.555000	8.600000	0.000000	0.000000	2.000000
75%	1.519157	13.825000	3.600000	1.630000	73.087500	0.610000	9.172500	0.000000	0.100000	3.000000
max	1.533930	17.380000	4.490000	3.500000	75.410000	6.210000	16.190000	3.150000	0.510000	7.000000

In [7]:
```
# find number of instances (rows) that belong to each class. (absolute count)
dataset_k.groupby('Type').size()
```

Out[7]:
```
Type
1    70
2    76
3    17
5    13
6     9
7    29
dtype: int64
```

Split data into two arrays as X features and y the target value column.

Dividing the dataset into train and test sets.

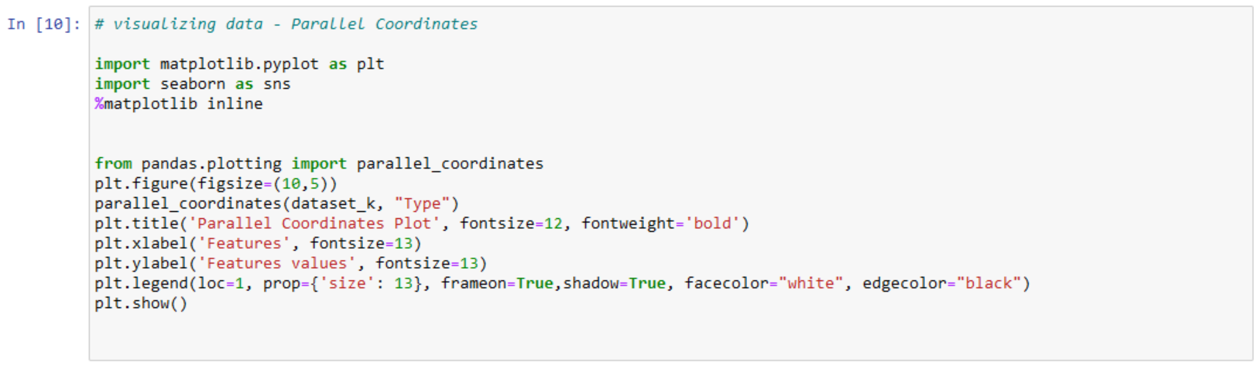
In [8]:
```
# Spliting data into two arrays: X (features) and y (labels : there are 7 lables).

feature_columns = ['RI', 'Na', 'Mg','Al','Si','K','Ca','Ba','Fe']
X = dataset_k[feature_columns].values
y = dataset_k['Type'].values

# to select features and labels arrays, the above python code can be re-write as below as well
# X = dataset.iloc[:, 1:9].values
# y = dataset.iloc[:, 9].values
```

In [ ]:
```
"""
in this dataset, label encoding is not required because lables are in numeric form.
it is mandatory to use LabelEncoder() when the labels are in categorical form.
the LabelEncoder() converts the categorical labels in to numerical labels.
the KNeighborsClassifier accepts labels in numerical form only.
"""
```

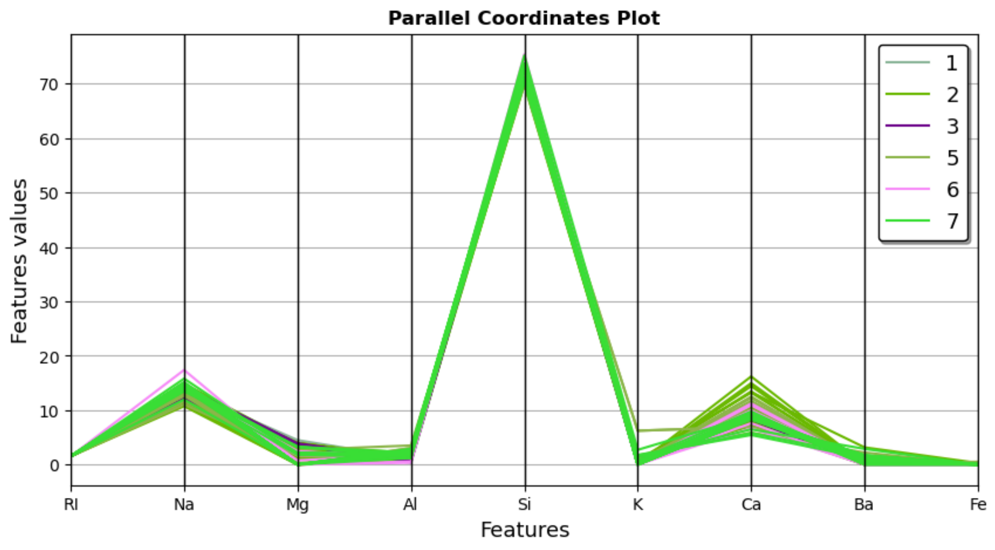
In [9]:
```
# deviding dataset into training and test sets.
# this is to check the classifier works correctly or not.

from sklearn.model_selection import train_test_split
#from sklearn.cross_validation import train_test_split

# train 80% : test 20% scenario
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Parallel Coordinates:

Plotting multivariate data using parallel coordinates is one approach. It allows for the visual computation of additional statistics and makes data clusters visible. In parallel coordinates, points are represented as segments of connected lines. Every vertical line represents one attribute. A single set of connected line segments serves as the representation of a data point. Points that tend to cluster together will seem closer together.



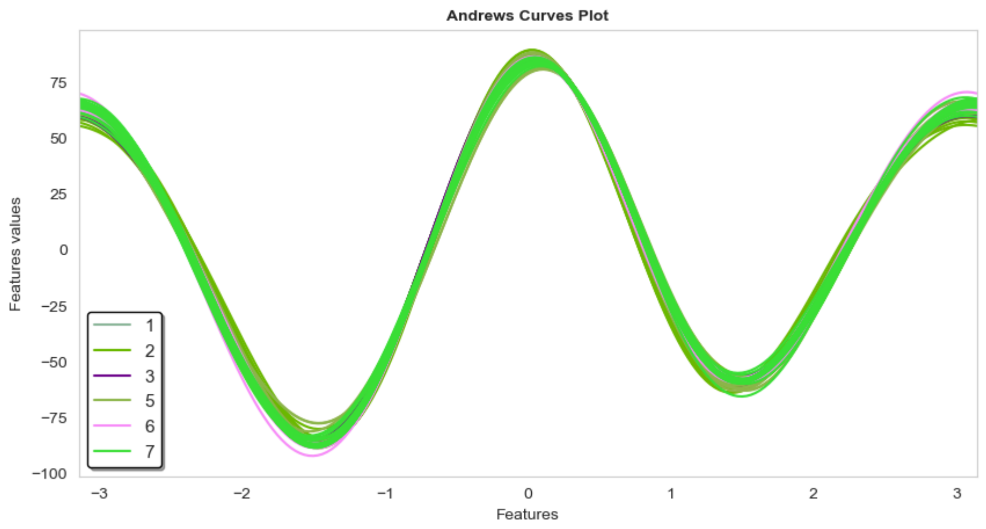
```
In [10]: # visualizing data - Parallel Coordinates

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline


from pandas.plotting import parallel_coordinates
plt.figure(figsize=(10,5))
parallel_coordinates(dataset_k, "Type")
plt.title('Parallel Coordinates Plot', fontsize=12, fontweight='bold')
plt.xlabel('Features', fontsize=13)
plt.ylabel('Features values', fontsize=13)
plt.legend(loc=1, prop={'size': 13}, frameon=True,shadow=True, facecolor="white", edgecolor="black")
plt.show()
```



Andrews Curves:

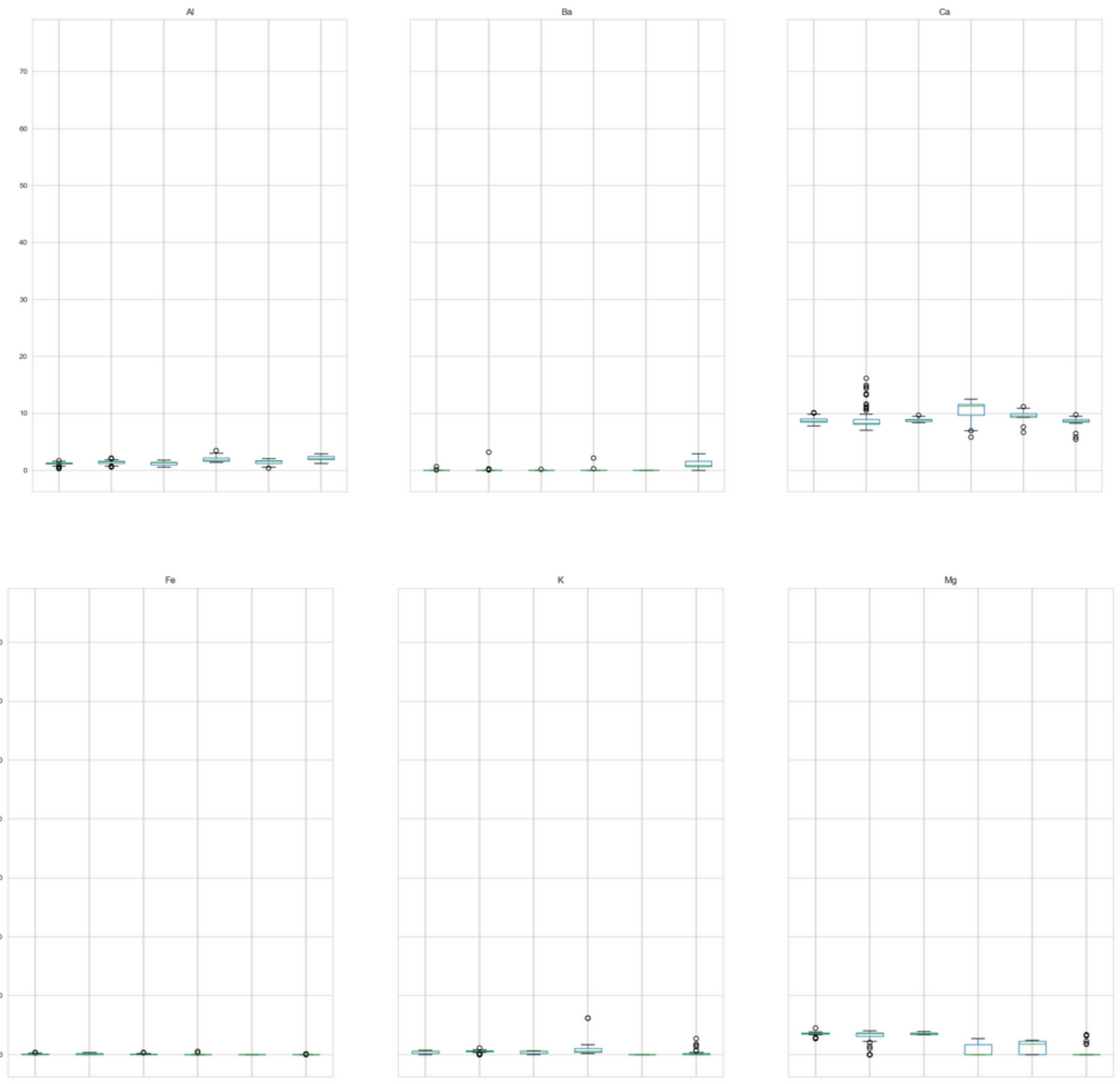
Multivariate data can be plotted as a large number of curves using Andrews curves, which use the features of the samples as coefficients for Fourier series. By applying distinct colors to these curves for every class, one can observe data clustering. Curves from samples belonging to the same class tend to be closer together and create larger structures.

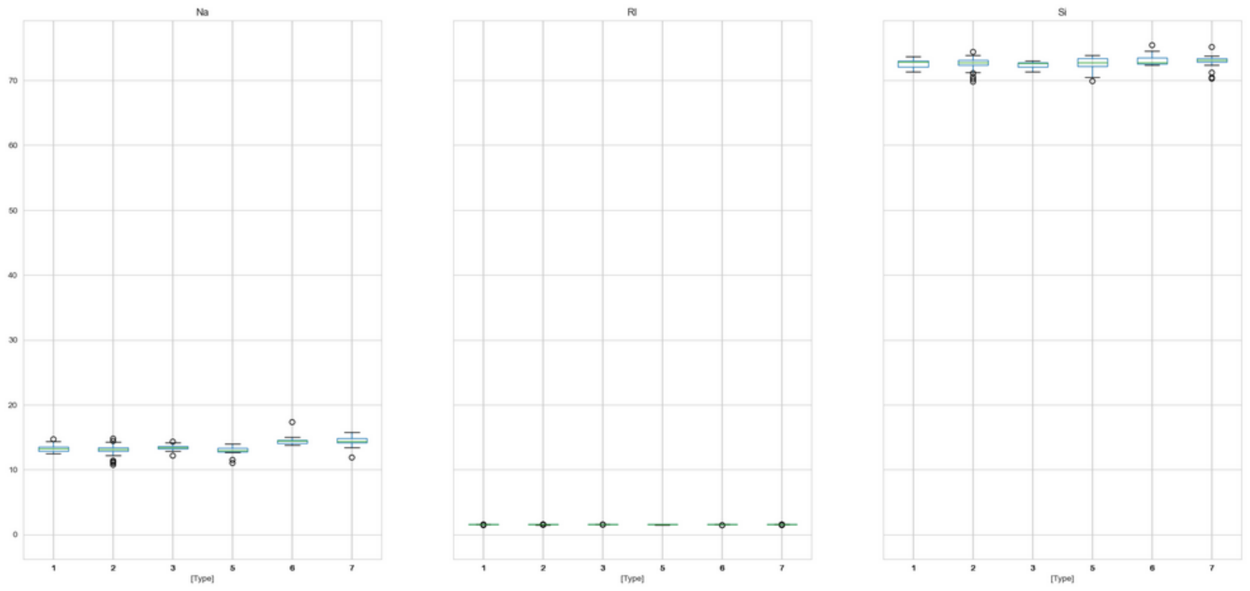
```
In [23]: from pandas.plotting import andrews_curves

         plt.figure(figsize=(10,5))

         andrews_curves(dataset_k, "Type")

         plt.title('Andrews Curves Plot', fontsize=10, fontweight='bold')
         plt.xlabel('Features', fontsize=10)
         plt.ylabel('Features values', fontsize=10)
         plt.legend(loc=3, prop={'size': 11}, frameon=True,shadow=True, facecolor="white", edgecolor="black")
         plt.show()
```



Generating a Boxplot to find outliers of each feature.

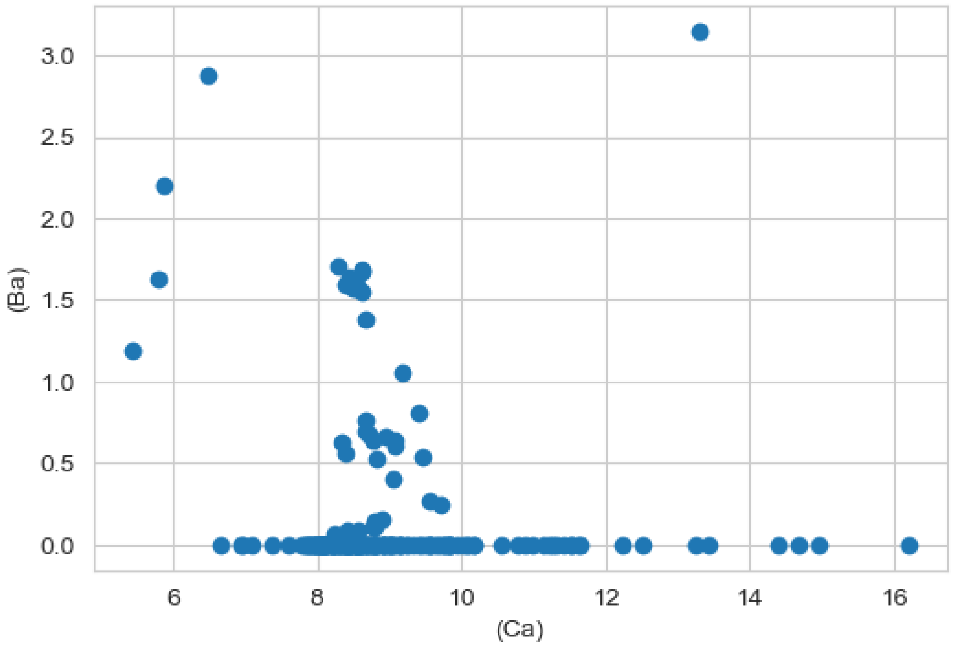
```
In [25]: # Boxplots
         # to check outliers

         plt.figure()
         dataset_k.boxplot(by="Type", figsize=(25, 40))
         plt.show()
```


Al
Ba
Ca
Fe
K
Mg
70
60
50
40
30
20
10
0

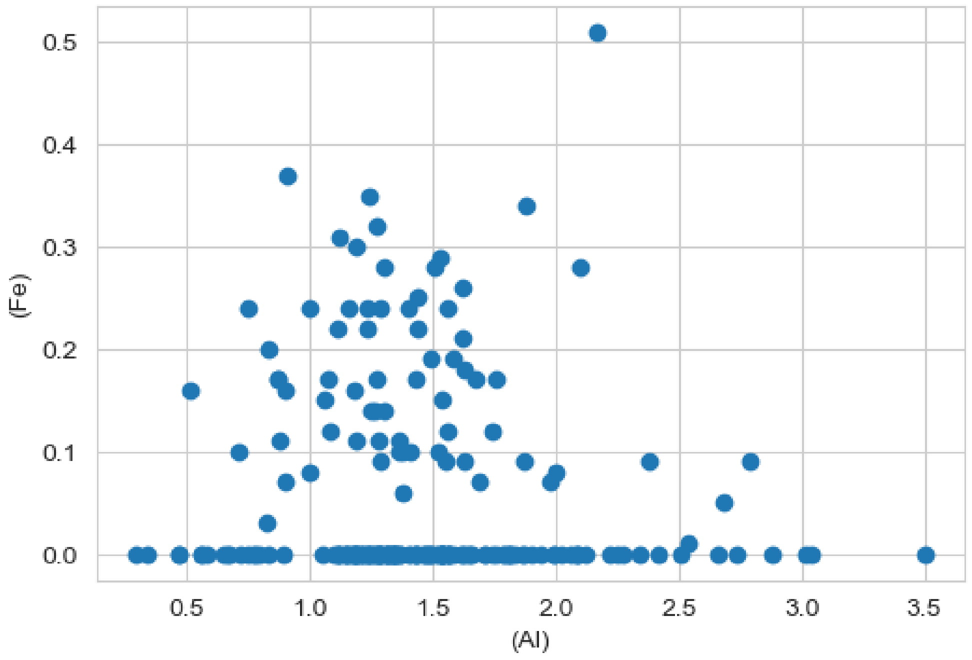


Determining outliers of each feature using scatter plots.

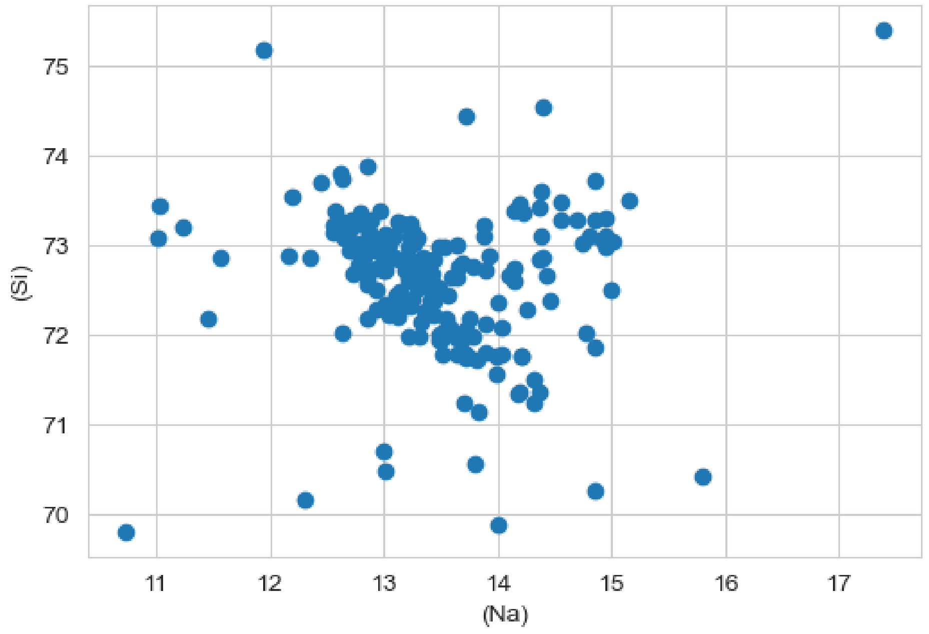
```
In [30]: # detecting outliers using a scatter plot
         # Using Ba and Ca attributes

         fig, ax = plt.subplots(figsize=(6, 4))
         ax.scatter(dataset_k['Ca'], dataset_k['Ba'])

         # x-axis label
         ax.set_xlabel('(Ca)')

         # y-axis label
         ax.set_ylabel('(Ba)')
         plt.show()
```



```
In [31]: # detecting outliers using a scatter plot
         # Using Al and Fe attributes

         fig, ax = plt.subplots(figsize=(6, 4))
         ax.scatter(dataset_k['Al'], dataset_k['Fe'])

         # x-axis label
         ax.set_xlabel('(Al)')

         # y-axis label
         ax.set_ylabel('(Fe)')
         plt.show()
```



```
In [35]: # detecting outliers using a scatter plot
         # Using Na and Si attributes

         fig, ax = plt.subplots(figsize=(6, 4))
         ax.scatter(dataset_k['Na'], dataset_k['Si'])

         # x-axis label
         ax.set_xlabel('(Na)')

         # y-axis label
         ax.set_ylabel('(Si)')
         plt.show()
```

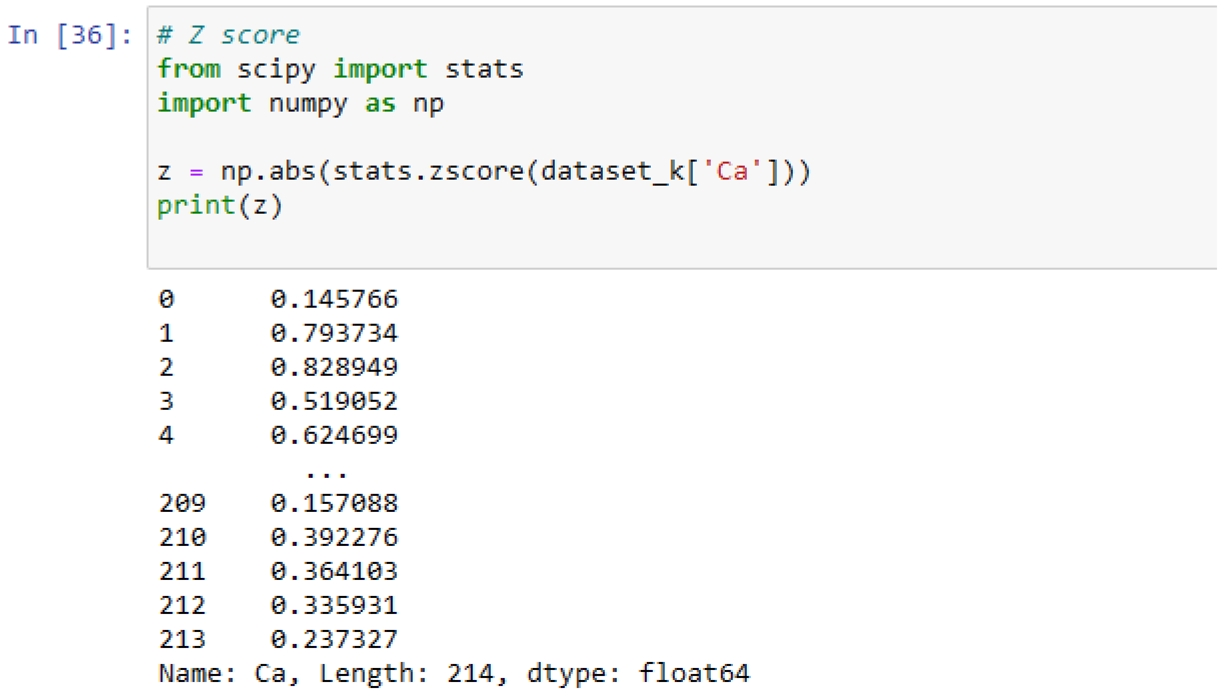


Defining outliers using the Z-score:

Another name for the Z-score is a standard score. This number or score makes it easier to determine how far a data point deviates from the mean. Additionally, z score values of data points can be used to define the outliers when a threshold value has been defined.

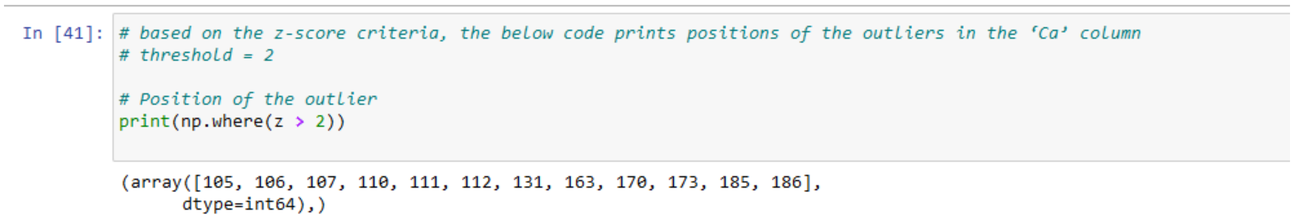
Calculation:

Zscore = (data_point -mean) / standard deviation



```
In [36]: # Z score
         from scipy import stats
         import numpy as np

         z = np.abs(stats.zscore(dataset_k['Ca']))
         print(z)
```

```
0      0.145766
1      0.793734
2      0.828949
3      0.519052
4      0.624699
         ...
209    0.157088
210    0.392276
211    0.364103
212    0.335931
213    0.237327
Name: Ca, Length: 214, dtype: float64
```



```
In [41]: # based on the z-score criteria, the below code prints positions of the outliers in the ‘Ca’ column
         # threshold = 2

         # Position of the outlier
         print(np.where(z > 2))
```

```
(array([105, 106, 107, 110, 111, 112, 131, 163, 170, 173, 185, 186],
      dtype=int64),)
```

The z-score values for each data item in the column "Ca" are displayed in the snapshot above.

3.0 is the typical value utilized to define an outlier threshold. Since 99.7% of the data points (as determined by the Gaussian Distribution method) fall between +/- 3 standard deviation.

Removing the outliers:

This is the process of removing unwanted values (an entry) from the dataset using its exact position.

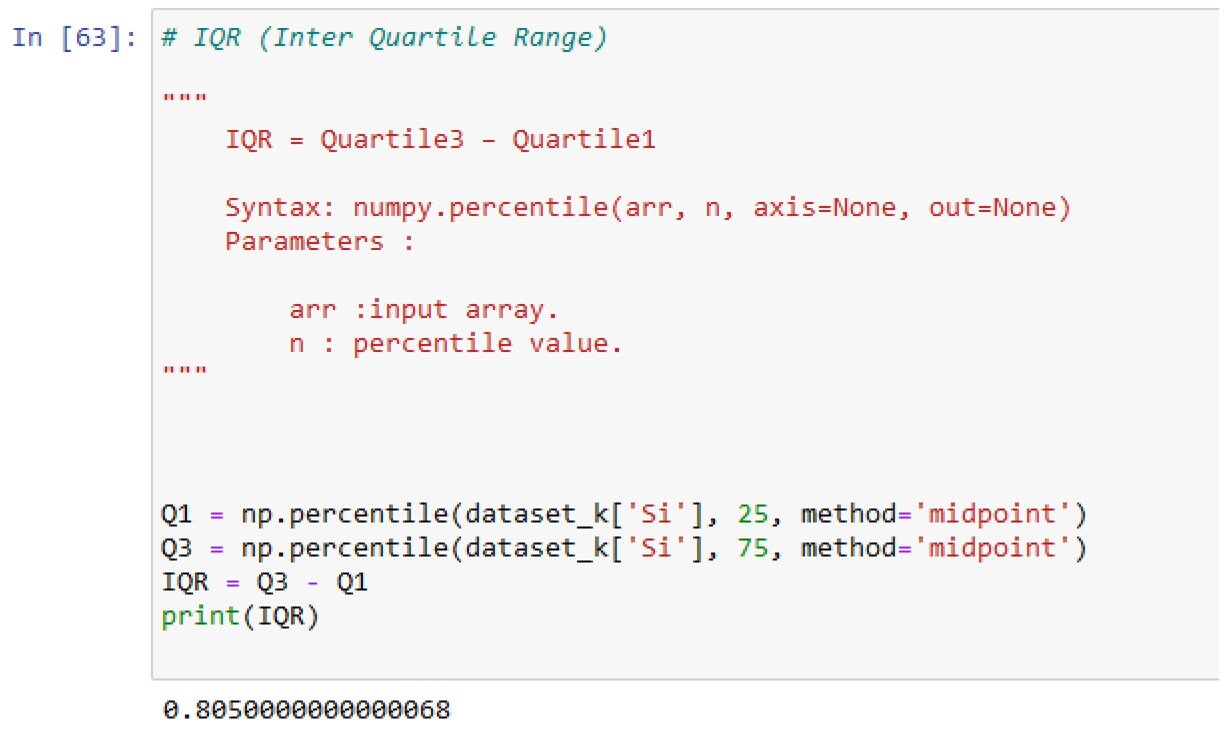
Syntax: dataframe.drop(row index,inplace=True)

Removing outliers using IQR:

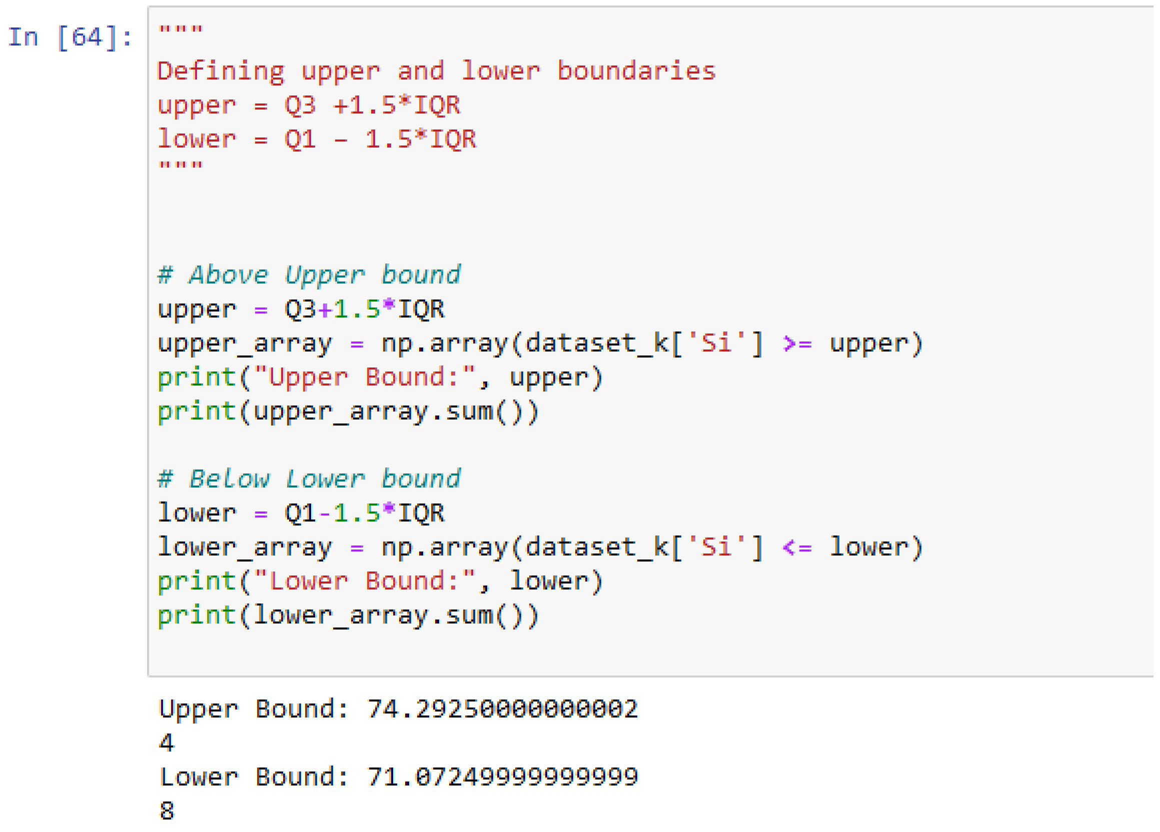
It is possible to use the IQR (Inter Quartile Range) method to remove outliers.

Removing outliers from the 'Si' column:

Calculating the IQR value:

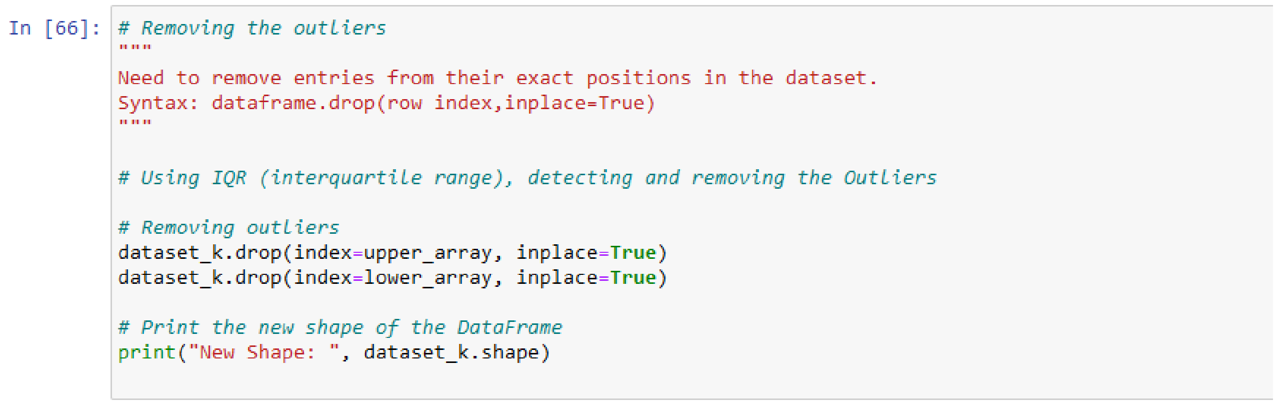


```
In [63]: # IQR (Inter Quartile Range)
         """
             IQR = Quartile3 – Quartile1

             Syntax: numpy.percentile(arr, n, axis=None, out=None)
             Parameters :

                 arr :input array.
                 n : percentile value.
         """



         Q1 = np.percentile(dataset_k['Si'], 25, method='midpoint')
         Q3 = np.percentile(dataset_k['Si'], 75, method='midpoint')
         IQR = Q3 - Q1
         print(IQR)

         0.8050000000000068
```



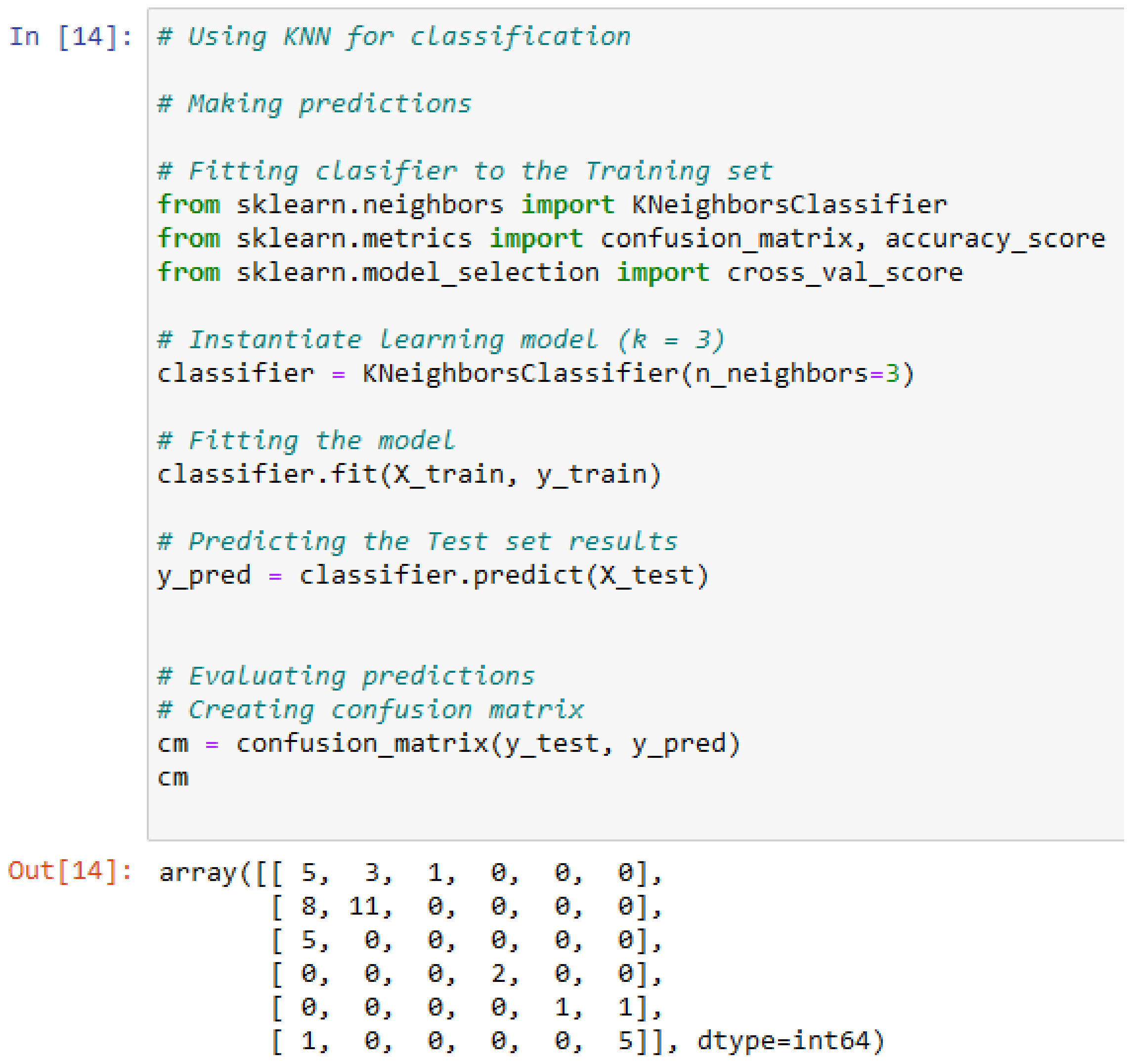
```
In [64]: """
         Defining upper and lower boundaries
         upper = Q3 +1.5*IQR
         lower = Q1 - 1.5*IQR
         """


         # Above Upper bound
         upper = Q3+1.5*IQR
         upper_array = np.array(dataset_k['Si'] >= upper)
         print("Upper Bound:", upper)
         print(upper_array.sum())

         # Below Lower bound
         lower = Q1-1.5*IQR
         lower_array = np.array(dataset_k['Si'] <= lower)
         print("Lower Bound:", lower)
         print(lower_array.sum())

         Upper Bound: 74.29250000000002
         4
         Lower Bound: 71.07249999999999
         8
```

It determines the upper and lower bounds using the IQR, uses Boolean arrays to find the outlier indices, and then deletes the relevant rows from the DataFrame to create a new DataFrame that has the outliers removed.

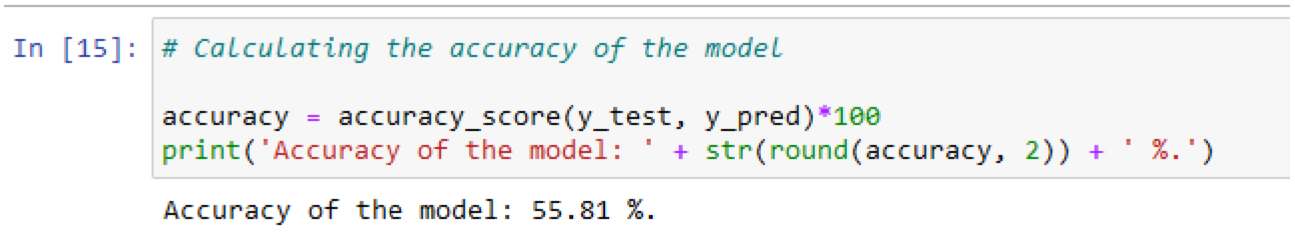


```
In [66]: # Removing the outliers
         """
         Need to remove entries from their exact positions in the dataset.
         Syntax: dataframe.drop(row index,inplace=True)
         """

         # Using IQR (interquartile range), detecting and removing the Outliers

         # Removing outliers
         dataset_k.drop(index=upper_array, inplace=True)
         dataset_k.drop(index=lower_array, inplace=True)

         # Print the new shape of the DataFrame
         print("New Shape: ", dataset_k.shape)
```

Before removing the outliers from the ‘Ca’ column, the accuracy of the model:

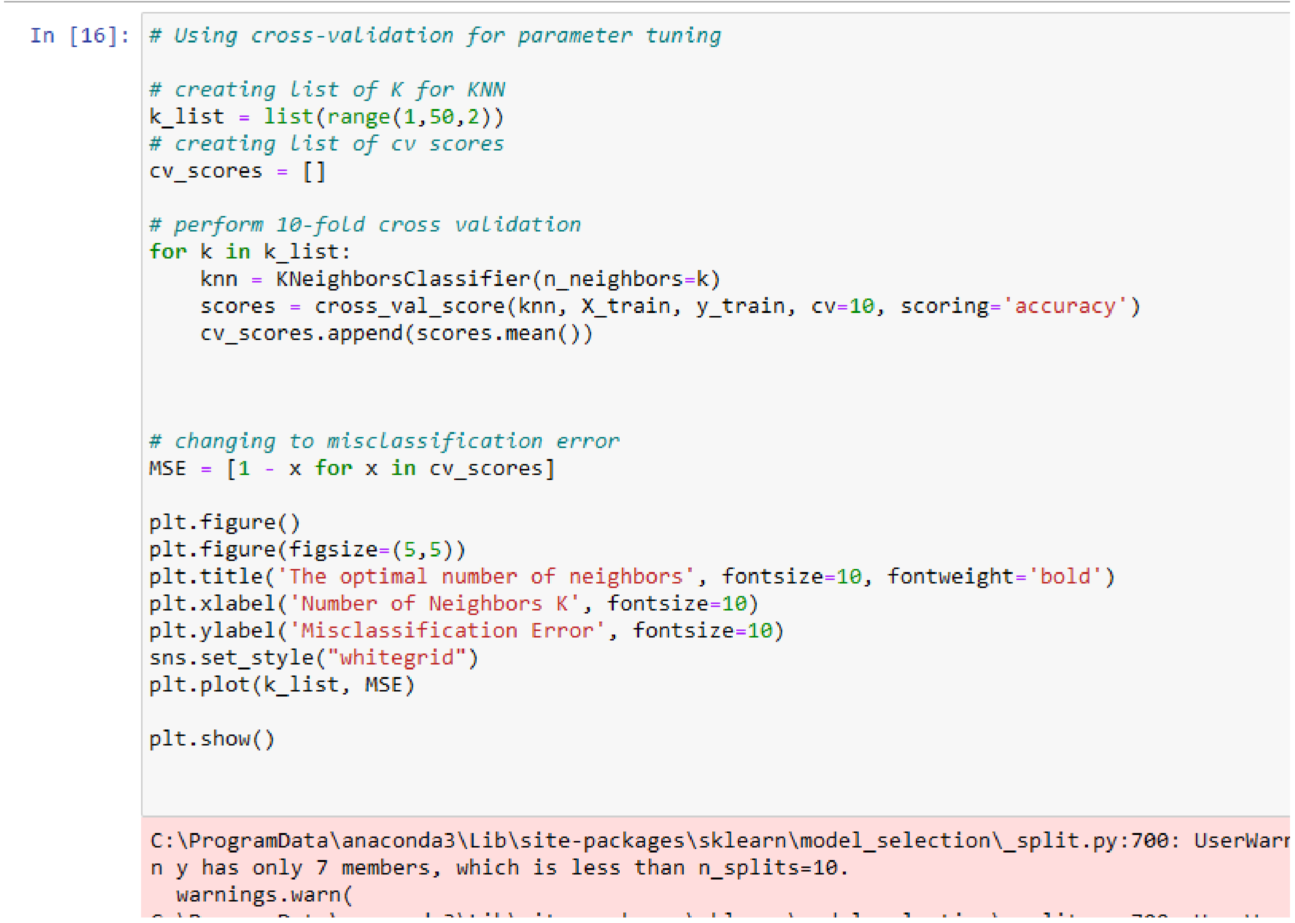


```
In [14]: # Using KNN for classification

         # Making predictions

         # Fitting clasifier to the Training set
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.metrics import confusion_matrix, accuracy_score
         from sklearn.model_selection import cross_val_score

         # Instantiate learning model (k = 3)
         classifier = KNeighborsClassifier(n_neighbors=3)

         # Fitting the model
         classifier.fit(X_train, y_train)

         # Predicting the Test set results
         y_pred = classifier.predict(X_test)


         # Evaluating predictions
         # Creating confusion matrix
         cm = confusion_matrix(y_test, y_pred)
         cm

Out[14]: array([[ 5,  3,  1,  0,  0,  0],
                [ 8, 11,  0,  0,  0,  0],
                [ 5,  0,  0,  0,  0,  0],
                [ 0,  0,  0,  2,  0,  0],
                [ 0,  0,  0,  0,  1,  1],
                [ 1,  0,  0,  0,  0,  5]], dtype=int64)
```



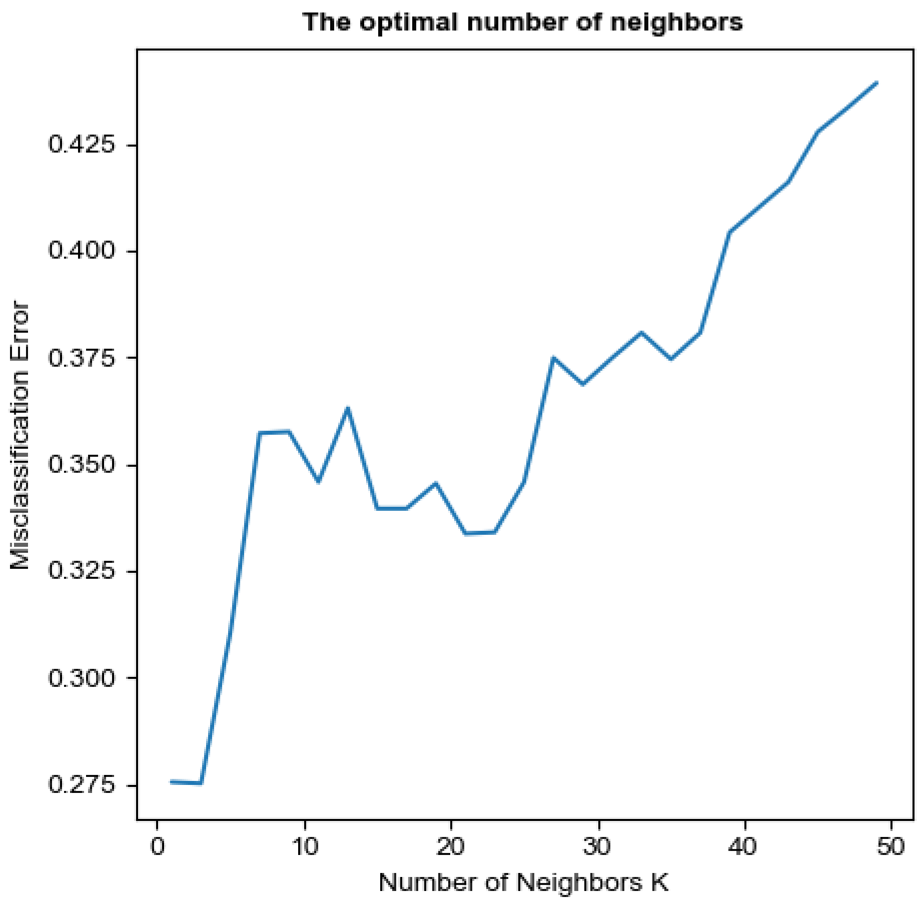
```
In [15]: # Calculating the accuracy of the model

         accuracy = accuracy_score(y_test, y_pred)*100
         print('Accuracy of the model: ' + str(round(accuracy, 2)) + ' %.')

         Accuracy of the model: 55.81 %.
```

Parameter tuning.



```
In [16]: # Using cross-validation for parameter tuning

         # creating list of K for KNN
         k_list = list(range(1,50,2))
         # creating list of cv scores
         cv_scores = []

         # perform 10-fold cross validation
         for k in k_list:
             knn = KNeighborsClassifier(n_neighbors=k)
             scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='accuracy')
             cv_scores.append(scores.mean())



         # changing to misclassification error
         MSE = [1 - x for x in cv_scores]

         plt.figure()
         plt.figure(figsize=(5,5))
         plt.title('The optimal number of neighbors', fontsize=10, fontweight='bold')
         plt.xlabel('Number of Neighbors K', fontsize=10)
         plt.ylabel('Misclassification Error', fontsize=10)
         sns.set_style("whitegrid")
         plt.plot(k_list, MSE)

         plt.show()
```

```
C:\ProgramData\anaconda3\Lib\site-packages\sklearn\model_selection\_split.py:700: UserWarr
n y has only 7 members, which is less than n_splits=10.
  warnings.warn(
```

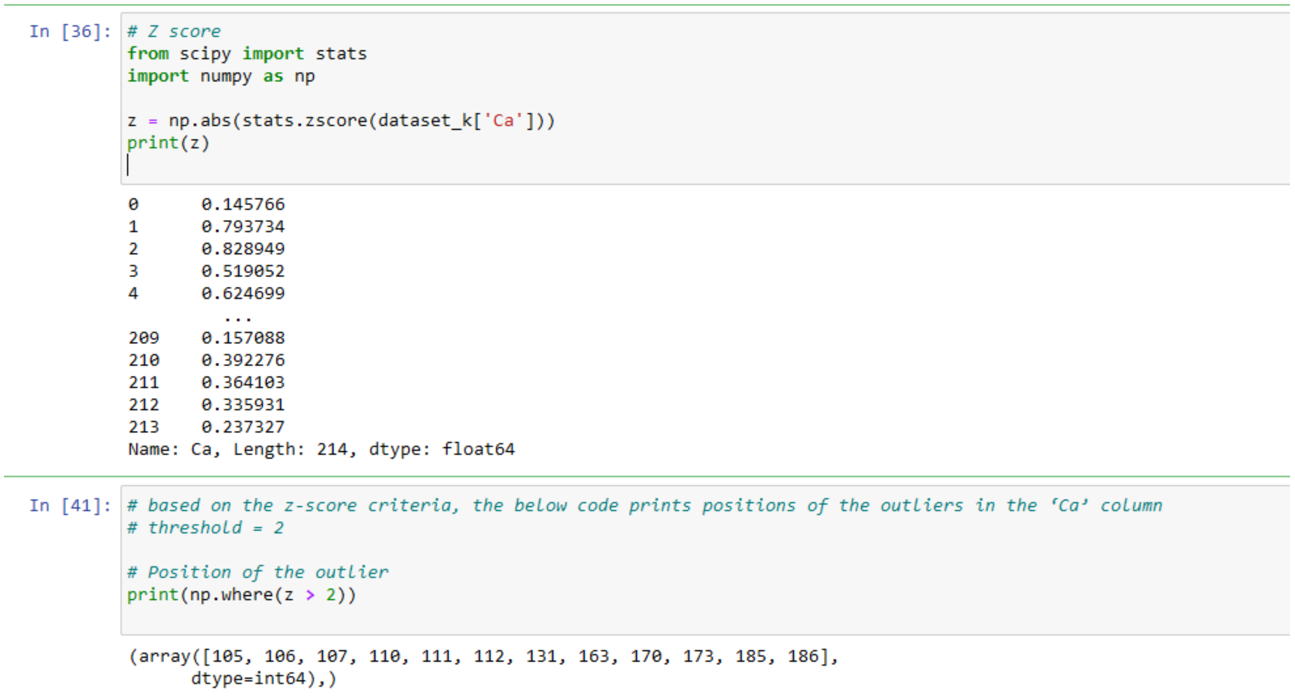


Calculating the optimal k.

```
In [17]: # calculating the best k

         best_k = k_list[MSE.index(min(MSE))]

         print("The optimal number of neighbors is %d." % best_k)
```

```
The optimal number of neighbors is 3.
```

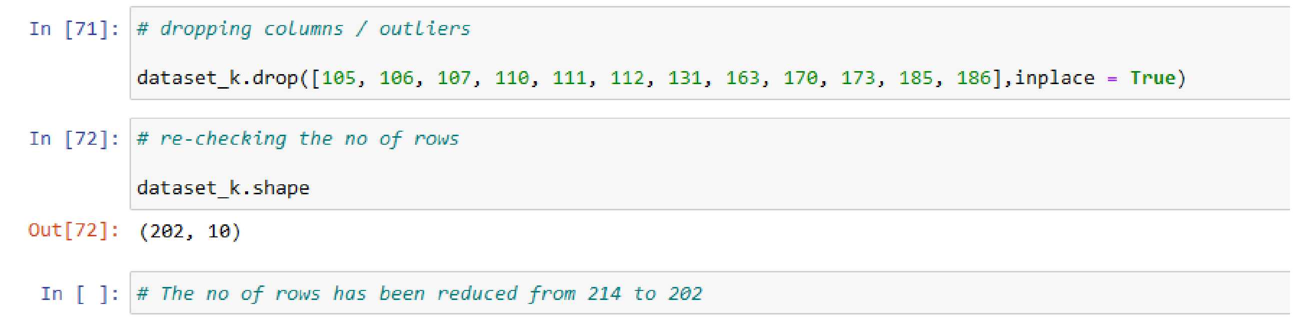
But, after removing outliers from the column ‘Ca’, the accuracy of the model is increased.



```
In [36]: # Z score
         from scipy import stats
         import numpy as np

         z = np.abs(stats.zscore(dataset_k['Ca']))
         print(z)

0      0.145766
1      0.793734
2      0.828949
3      0.519052
4      0.624699
         ...
209    0.157088
210    0.392276
211    0.364103
212    0.335931
213    0.237327
Name: Ca, Length: 214, dtype: float64
```

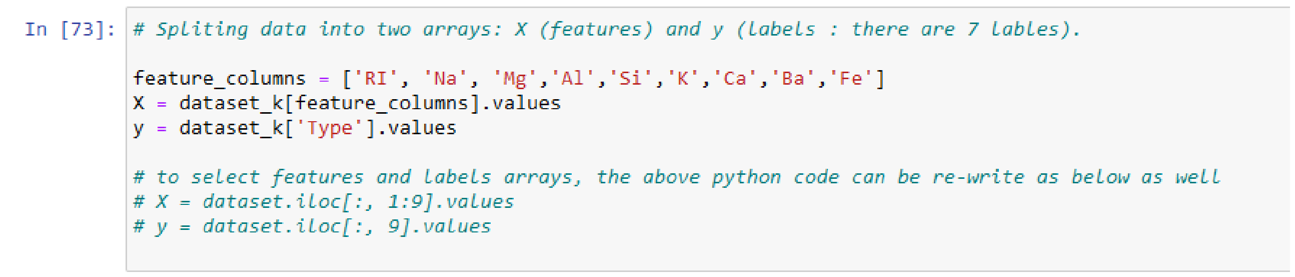
```
In [41]: # based on the z-score criteria, the below code prints positions of the outliers in the 'Ca' column
         # threshold = 2

         # Position of the outlier
         print(np.where(z > 2))

(array([105, 106, 107, 110, 111, 112, 131, 163, 170, 173, 185, 186],
      dtype=int64),)
```

Removing outliers.



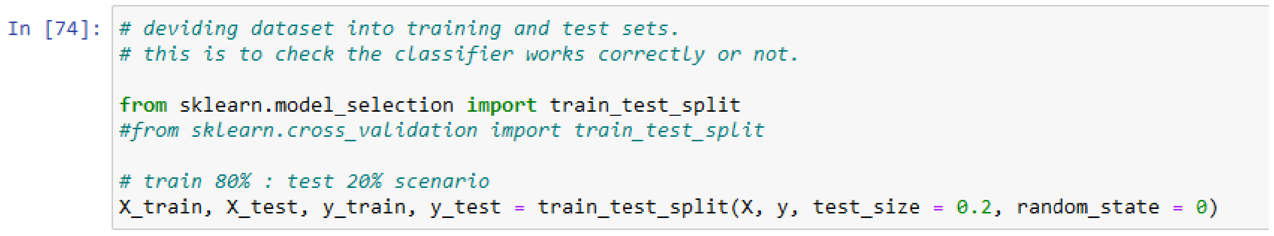
```
In [71]: # dropping columns / outliers

         dataset_k.drop([105, 106, 107, 110, 111, 112, 131, 163, 170, 173, 185, 186],inplace = True)

In [72]: # re-checking the no of rows

         dataset_k.shape

Out[72]: (202, 10)

In [ ]: # The no of rows has been reduced from 214 to 202
```

Split data into two arrays.



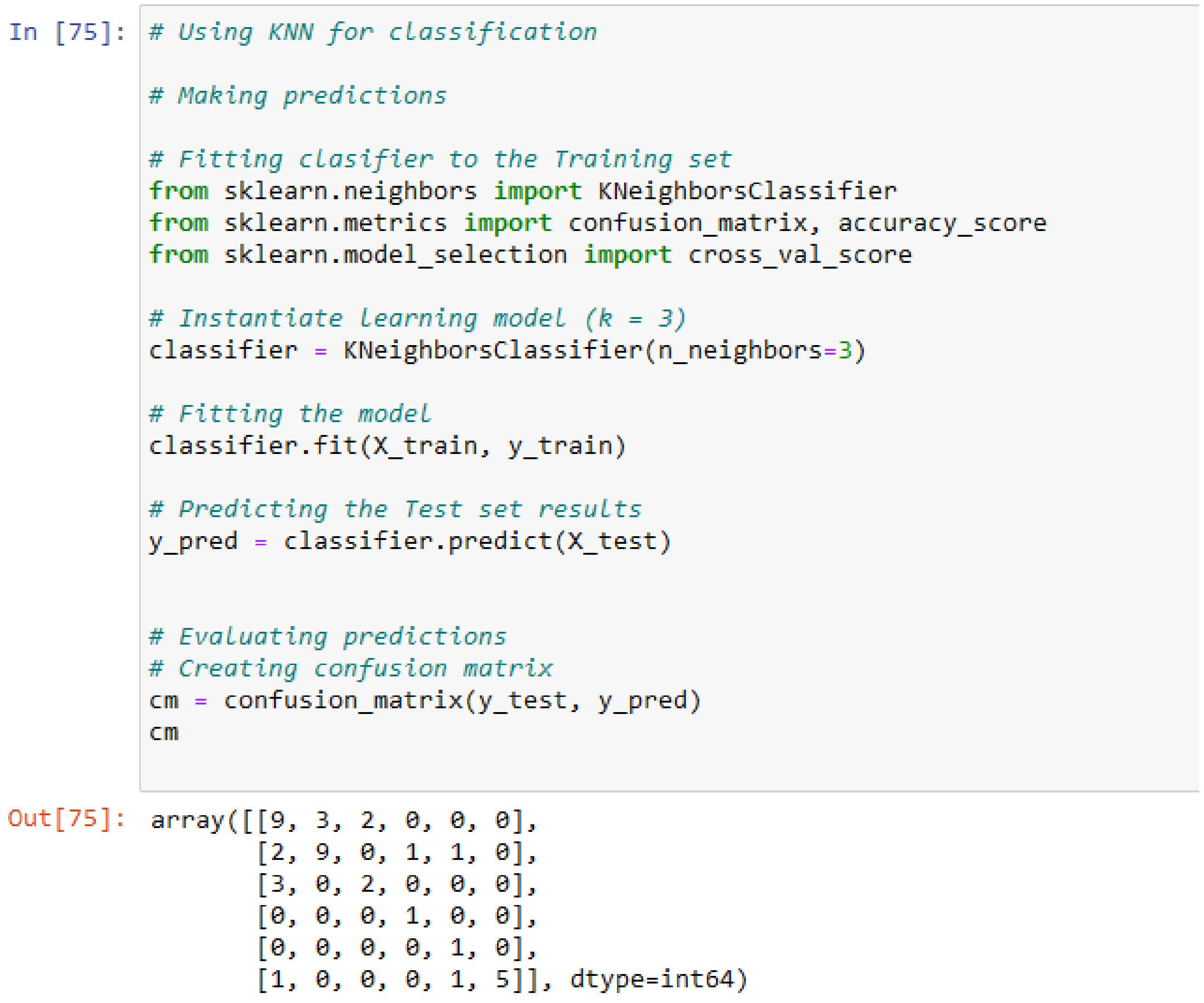
```
In [73]: # Spliting data into two arrays: X (features) and y (labels : there are 7 lables).

         feature_columns = ['RI', 'Na', 'Mg','Al','Si','K','Ca','Ba','Fe']
         X = dataset_k[feature_columns].values
         y = dataset_k['Type'].values

         # to select features and labels arrays, the above python code can be re-write as below as well
         # X = dataset.iloc[:, 1:9].values
         # y = dataset.iloc[:, 9].values
```

Dividing dataset into training and test sets.



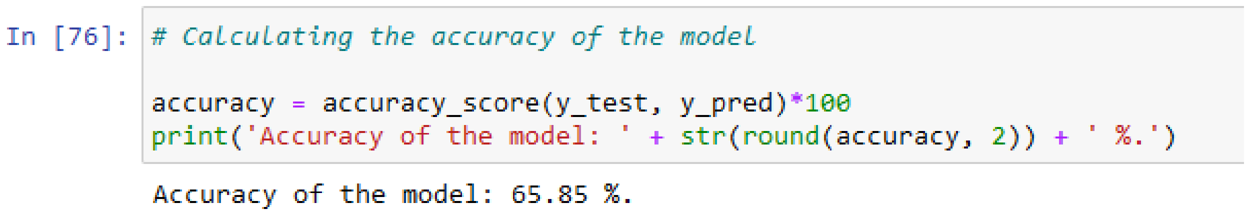
```
In [74]: # deviding dataset into training and test sets.
         # this is to check the classifier works correctly or not.

         from sklearn.model_selection import train_test_split
         #from sklearn.cross_validation import train_test_split

         # train 80% : test 20% scenario
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Using KNN for classification and making predictions.



```
In [75]: # Using KNN for classification

         # Making predictions

         # Fitting clasifier to the Training set
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.metrics import confusion_matrix, accuracy_score
         from sklearn.model_selection import cross_val_score

         # Instantiate learning model (k = 3)
         classifier = KNeighborsClassifier(n_neighbors=3)

         # Fitting the model
         classifier.fit(X_train, y_train)

         # Predicting the Test set results
         y_pred = classifier.predict(X_test)


         # Evaluating predictions
         # Creating confusion matrix
         cm = confusion_matrix(y_test, y_pred)
         cm

Out[75]: array([[9, 3, 2, 0, 0, 0],
                [2, 9, 0, 1, 1, 0],
                [3, 0, 2, 0, 0, 0],
                [0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 0],
                [1, 0, 0, 0, 1, 5]], dtype=int64)
```

Calculating the accuracy of the model.



```
In [76]: # Calculating the accuracy of the model

         accuracy = accuracy_score(y_test, y_pred)*100
         print('Accuracy of the model: ' + str(round(accuracy, 2)) + ' %.')

         Accuracy of the model: 65.85 %.
```

The model's accuracy rose to 65.85% from 55.81%. For the model to be as accurate as possible, all outliers must be eliminated from every column.

train/test split:

As any model's goal is to produce accurate predictions on unknown data, it is crucial that the model has a high out-of-sample accuracy. It is possible to increase out-of-sample accuracy by utilizing Train/Test Split. Splitting the dataset into training and testing sets, which are mutually exclusive, is known as the "train/test split."

Summary of Categorical attributes and encoding categorical features.

Before using any machine learning models, it is necessary to address the issue that certain models do not understand: categorical attributes.

Because each value in a categorical dataset indicates a distinct category, categorical data—variables with label values instead of numeric values—are sometimes referred to as nominal data.

Ordinal variables are a sort of categorical variable that have a natural ordering or other natural relationship among its categories.

Discreteization is the process of splitting a numerical variable's range into bins and allocating values to each bin in order to transform the numerical variable to an ordinal variable.

There are four typical methods for transforming category and ordinal variables into numerical values.

1. Ordinal Encoding

2. One-Hot Encoding

3. Dummy Variable Encoding

4. Label Encoder

References

Jason Brownlee. (2023, 10 04). *A Gentle Introduction to k-fold Cross-Validation*. Retrieved from machinelearningmastery.com: https://machinelearningmastery.com/k-fold-cross-validation/

Ajitesh Kumar . (2023, 11 18). *Machine Learning – Sensitivity vs Specificity Differences*. Retrieved from vitalflux.com: https://vitalflux.com/ml-metrics-sensitivity-vs-specificity-difference/

Aniruddha Bhandari . (2023, 10 27). *Feature Engineering: Scaling, Normalization, and Standardization*. Retrieved from www.analyticsvidhya.com: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

Hamming Distance. (n.d.). Retrieved from people.revoledu.com: https://people.revoledu.com/kardi/tutorial/Similarity/HammingDistance.html#google_vignette

Imputation of missing values. (n.d.). Retrieved from scikit-learn.org: https://scikit-learn.org/stable/modules/impute.html

Jason Brownlee. (2020, 08 19). *4 Distance Measures for Machine Learning*. Retrieved from machinelearningmastery.com: https://machinelearningmastery.com/distance-measures-for-machine-learning/

Jason Brownlee. (2020, 02 24). *Develop k-Nearest Neighbors in Python From Scratch*. Retrieved from machinelearningmastery.com: https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

MANAN KHURMA. (n.d.). *Euclidean Distance Formula*. Retrieved from www.cuemath.com: https://www.cuemath.com/euclidean-distance-formula/

Maximum Manhattan distance between a distinct pair from N coordinates. (2023, 01 04). Retrieved from www.geeksforgeeks.org: https://www.geeksforgeeks.org/maximum-manhattan-distance-between-a-distinct-pair-from-n-coordinates/

Rajeshsha. (2023, 11 30). *Detect and Remove the Outliers using Python*. Retrieved from www.geeksforgeeks.org: https://www.geeksforgeeks.org/detect-and-remove-the-outliers-using-python/

Rijk de Wet. (2023, 06 05). *Manhattan Distance Calculator*. Retrieved from www.omnicalculator.com: https://www.omnicalculator.com/math/manhattan-distance

Sarang Narkhede. (2018, 05 09). *Understanding Confusion Matrix*. Retrieved from towardsdatascience.com: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

What is Overfitting? (n.d.). Retrieved from aws.amazon.com: https://aws.amazon.com/what-is/overfitting/

www.shiksha.com. (2023, 03 06). *How to Compute Euclidean Distance in Python*. Retrieved from www.shiksha.com: https://www.shiksha.com/online-courses/articles/how-to-compute-euclidean-distance-in-python/

Zach. (2020, 12 17). *How to Calculate Hamming Distance in Python (With Examples)*. Retrieved from www.statology.org: https://www.statology.org/hamming-distance-python/