Course name: **Python programming and analytics by Rahul Sir**

Topic name: **KNN classification theory and iris case study (Sumit batch)**

Video name: **KNN classification theory and iris case study**

Video length: **1 hour 15 minutes 56 seconds**

The K-nearest neighbors (KNN) algorithm is a type of supervised machine learning algorithms. KNN is extremely easy to implement in its most basic form, and yet performs quite complex classification tasks. It is a lazy learning algorithm since it doesn't have a specialized training phase. Rather, it uses all of the data for training while classifying a new data point or instance. KNN is a non-parametric learning algorithm, which means that it doesn't assume anything about the underlying data. This is an extremely useful feature since most of the real-world data doesn't really follow any theoretical assumption e.g. linear-separability, uniform distribution, etc.

The intuition behind the KNN algorithm is one of the simplest of all the supervised machine learning algorithms. It simply calculates the distance of a new data point to all other training data points. The distance can be of any type e.g Euclidean or Manhattan etc. It then selects the K-nearest data points, where K can be any integer. Finally, it assigns the data point to the class to which the majority of the K data points belong.

Pros

1. It is extremely easy to implement
2. As said earlier, it is lazy learning algorithm and therefore requires no training prior to making real time predictions. This

makes the KNN algorithm much faster than other algorithms that require training e.g SVM, linear regression, etc.

3. Since the algorithm requires no training before making predictions, new data can be added seamlessly.
4. There are only two parameters required to implement KNN i.e. the value of K and the distance function (e.g. Euclidean or Manhattan etc.)

Cons

1. The KNN algorithm doesn't work well with high dimensional data because with large number of dimensions, it becomes difficult for the algorithm to calculate distance in each dimension.
2. The KNN algorithm has a high prediction cost for large datasets. This is because in large datasets the cost of calculating distance between new point and each existing point becomes higher.
3. Finally, the KNN algorithm doesn't work well with categorical features since it is difficult to find the distance between dimensions with categorical features.

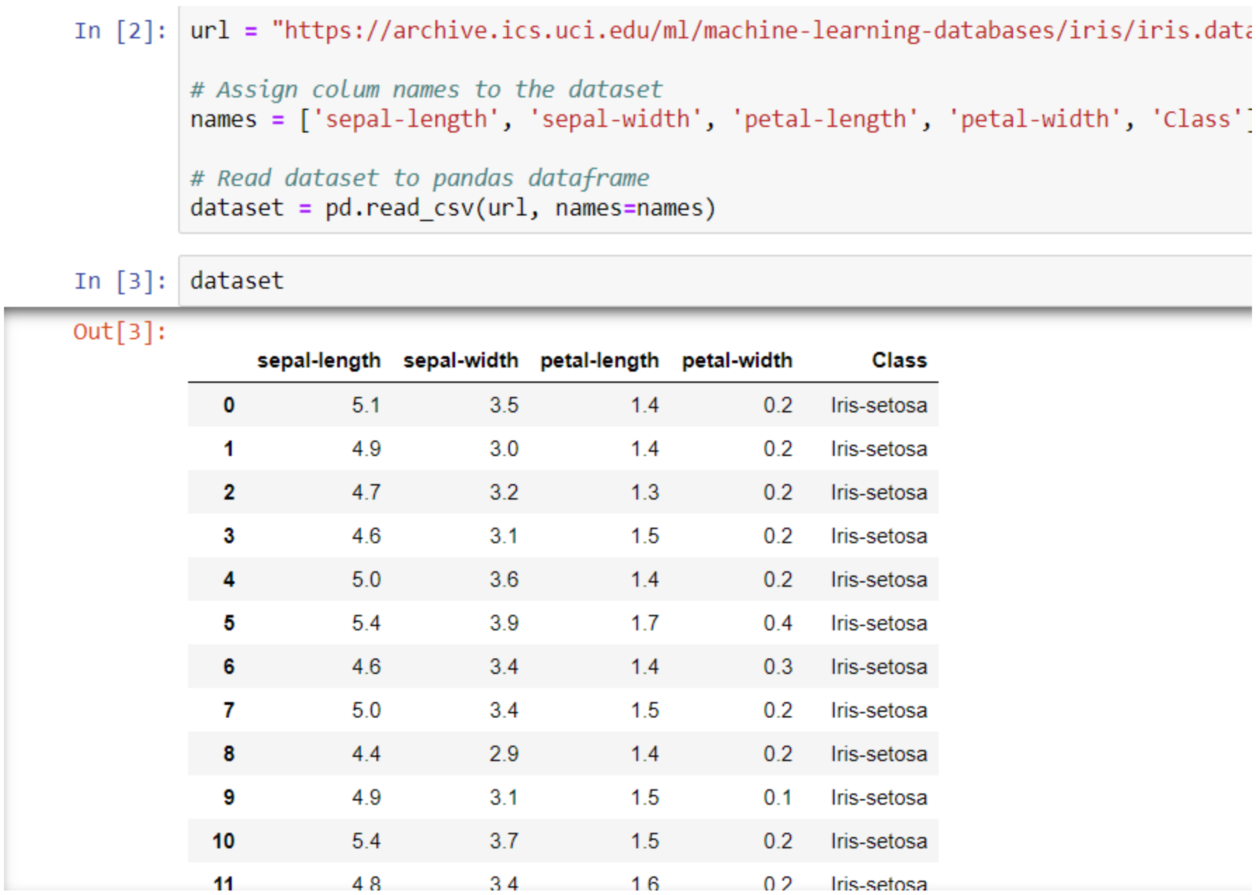
Importing Libraries

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
```

The Dataset

We are going to use the famous iris data set for our KNN example. The dataset consists of four attributes: sepal-width, sepal-length, petal-width and petal-length. These are the attributes of specific types of iris plant. The task is to predict the class to which these plants belong. There are three classes in the dataset: *Iris setosa*, *Iris versicolor* and *Iris virginica*.

To import the dataset and load it into our Pandas data frame, execute the following code:



```
In [2]: url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

        # Assign colum names to the dataset
        names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

        # Read dataset to pandas dataframe
        dataset = pd.read_csv(url, names=names)
```

```
In [3]: dataset
```

```
Out[3]:
        sepal-length  sepal-width  petal-length  petal-width        Class
     0           5.1          3.5           1.4          0.2  Iris-setosa
     1           4.9          3.0           1.4          0.2  Iris-setosa
     2           4.7          3.2           1.3          0.2  Iris-setosa
     3           4.6          3.1           1.5          0.2  Iris-setosa
     4           5.0          3.6           1.4          0.2  Iris-setosa
     5           5.4          3.9           1.7          0.4  Iris-setosa
     6           4.6          3.4           1.4          0.3  Iris-setosa
     7           5.0          3.4           1.5          0.2  Iris-setosa
     8           4.4          2.9           1.4          0.2  Iris-setosa
     9           4.9          3.1           1.5          0.1  Iris-setosa
    10           5.4          3.7           1.5          0.2  Iris-setosa
    11           4.8          3.4           1.6          0.2  Iris-setosa
```

PREPROCESSING:

• Pre-processing refers to the transformations applied to our data before feeding it to the algorithm.
• Data Preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is

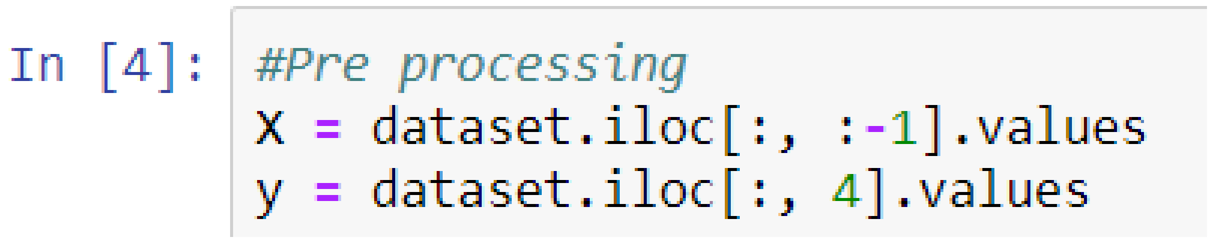
gathered from different sources it is collected in raw format which is not feasible for the analysis.

Need of Data Preprocessing

• For achieving better results from the applied model in Machine Learning projects the format of the data has to be in a proper manner.

• Another aspect is that data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithms are executed in one data set, and best out of them is chosen.

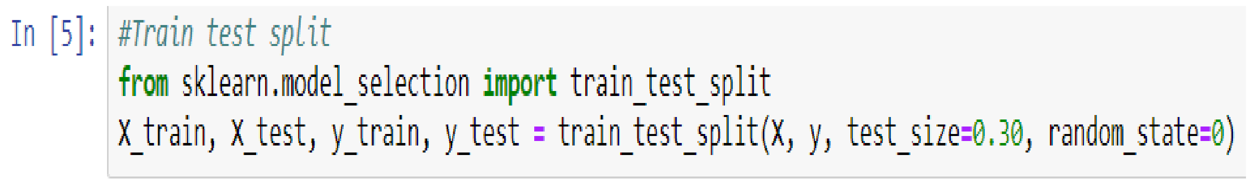
Pandas provide a unique method to retrieve rows from a Data frame. **Data.iloc[]** method is used when the index label of a data frame is something other than numeric series of 0, 1, 2, 3….n or in case the user doesn't know the index label. Rows can be extracted using an imaginary index position which isn't visible in the data frame. iloc is used for indexing or selecting based on position i.e., by row number and column number



```
In [4]: #Pre processing
        X = dataset.iloc[:, :-1].values
        y = dataset.iloc[:, 4].values
```

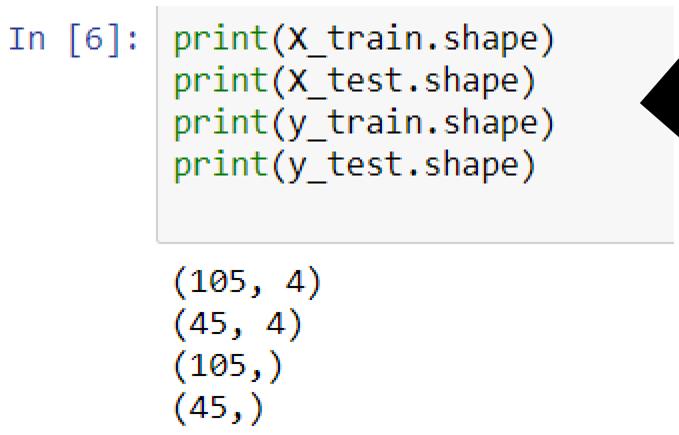
split the data

You can't possibly manually split the dataset into two. And you also have to make sure you split the data in a random manner. To help us with this task, the SciKit library provides a tool, called the Model Selection library. There's a class in the library which is, aptly, named 'train_test_split.' Using this we can easily split the dataset into the training and the testing datasets in various proportions.



```
In [5]: #Train test split
        from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)
```

- **test_size** — This parameter decides the size of the data that has to be split as the test dataset. This is given as a fraction. For example, if you pass 0.2 as the value, the dataset will be split 20% as the test dataset.
- **train_size** — You have to specify this parameter only if you're not specifying the test_size. This is the same as test_size, but instead you tell the class what percent of the dataset you want to split as the training set.
- **random_state** — Here you pass an integer, which will act as the seed for the random number generator during the split.



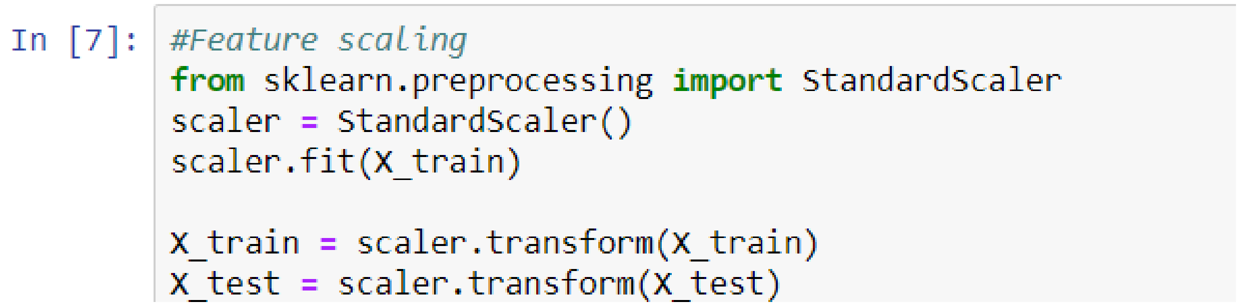
```
In [6]: print(X_train.shape)
        print(X_test.shape)
        print(y_train.shape)
        print(y_test.shape)

        (105, 4)
        (45, 4)
        (105,)
        (45,)
```

if you want to check how many rows and columns are there in train and test data for X and Y

feature scaling

Feature Scaling is a technique to standardize the independent features present in the data in a fixed range. It is performed during the data pre-processing to handle highly varying magnitudes or values or units. If feature scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values.

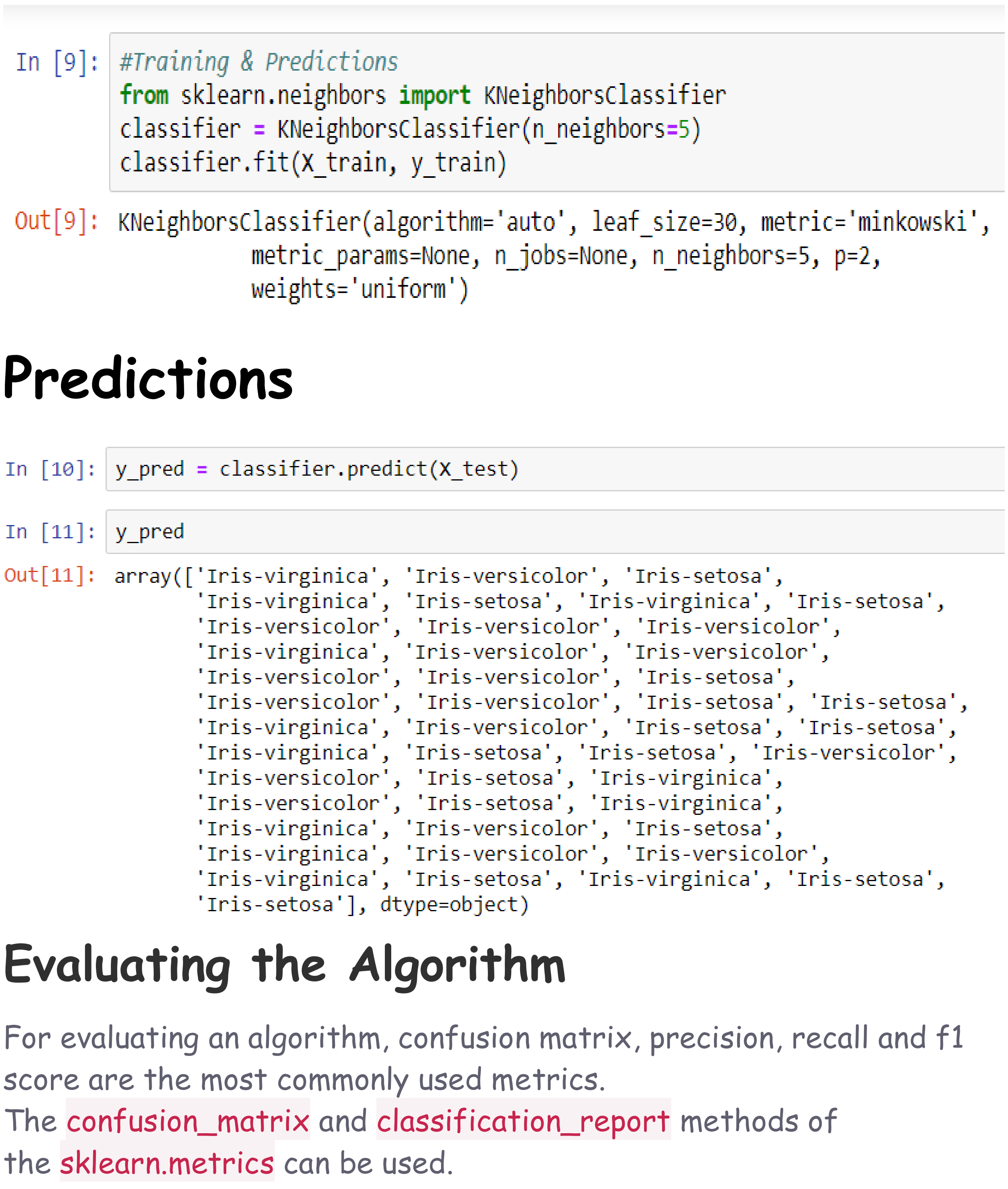
Example: If an algorithm is not using feature scaling method then it can consider the value 3000 meter to be greater than 5 km but that's actually not true and, in this case, the algorithm will give wrong predictions. So, we use Feature Scaling to bring all values to same magnitudes and thus, tackle this issue.



```
In [7]: #Feature scaling
        from sklearn.preprocessing import StandardScaler
        scaler = StandardScaler()
        scaler.fit(X_train)

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

Training the model

The first step is to import the KNeighborsClassifier class from the sklearn.neighbors library. In the second line, this class is initialized with one parameter, i.e. n_neigbours. This is basically the value for the K. There is no ideal value for K and it is selected after testing and evaluation, however to start out, 5 seems to be the most commonly used value for KNN algorithm



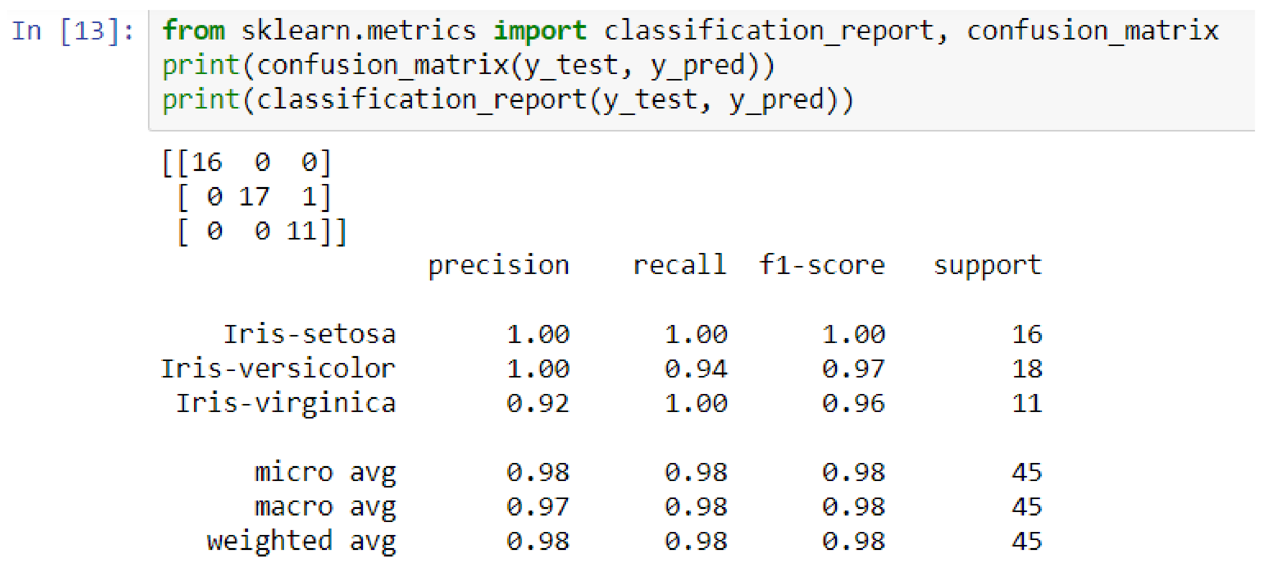
```
In [9]:  #Training & Predictions
          from sklearn.neighbors import KNeighborsClassifier
          classifier = KNeighborsClassifier(n_neighbors=5)
          classifier.fit(X_train, y_train)

Out[9]:  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

Predictions

```
In [10]: y_pred = classifier.predict(X_test)

In [11]: y_pred

Out[11]: array(['Iris-virginica', 'Iris-versicolor', 'Iris-setosa',
                'Iris-virginica', 'Iris-setosa', 'Iris-virginica', 'Iris-setosa',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-virginica', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-setosa',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-setosa', 'Iris-setosa',
                'Iris-virginica', 'Iris-versicolor', 'Iris-setosa', 'Iris-setosa',
                'Iris-virginica', 'Iris-setosa', 'Iris-setosa', 'Iris-versicolor',
                'Iris-versicolor', 'Iris-setosa', 'Iris-virginica',
                'Iris-versicolor', 'Iris-setosa', 'Iris-virginica',
                'Iris-virginica', 'Iris-versicolor', 'Iris-setosa',
                'Iris-virginica', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-virginica', 'Iris-setosa', 'Iris-virginica', 'Iris-setosa',
                'Iris-setosa'], dtype=object)
```

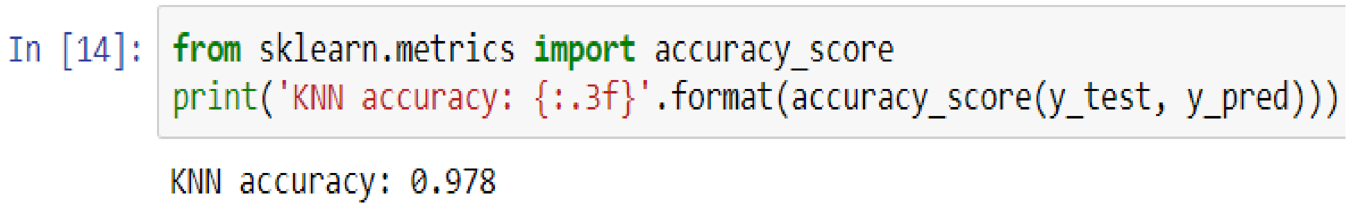
Evaluating the Algorithm

For evaluating an algorithm, confusion matrix, precision, recall and f1 score are the most commonly used metrics.
The confusion_matrix and classification_report methods of the sklearn.metrics can be used.



```
In [13]: from sklearn.metrics import classification_report, confusion_matrix
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred))

         [[16  0  0]
          [ 0 17  1]
          [ 0  0 11]]
                          precision    recall  f1-score   support

             Iris-setosa       1.00      1.00      1.00        16
         Iris-versicolor       1.00      0.94      0.97        18
          Iris-virginica       0.92      1.00      0.96        11

               micro avg       0.98      0.98      0.98        45
               macro avg       0.97      0.98      0.98        45
            weighted avg       0.98      0.98      0.98        45
```

Accuracy

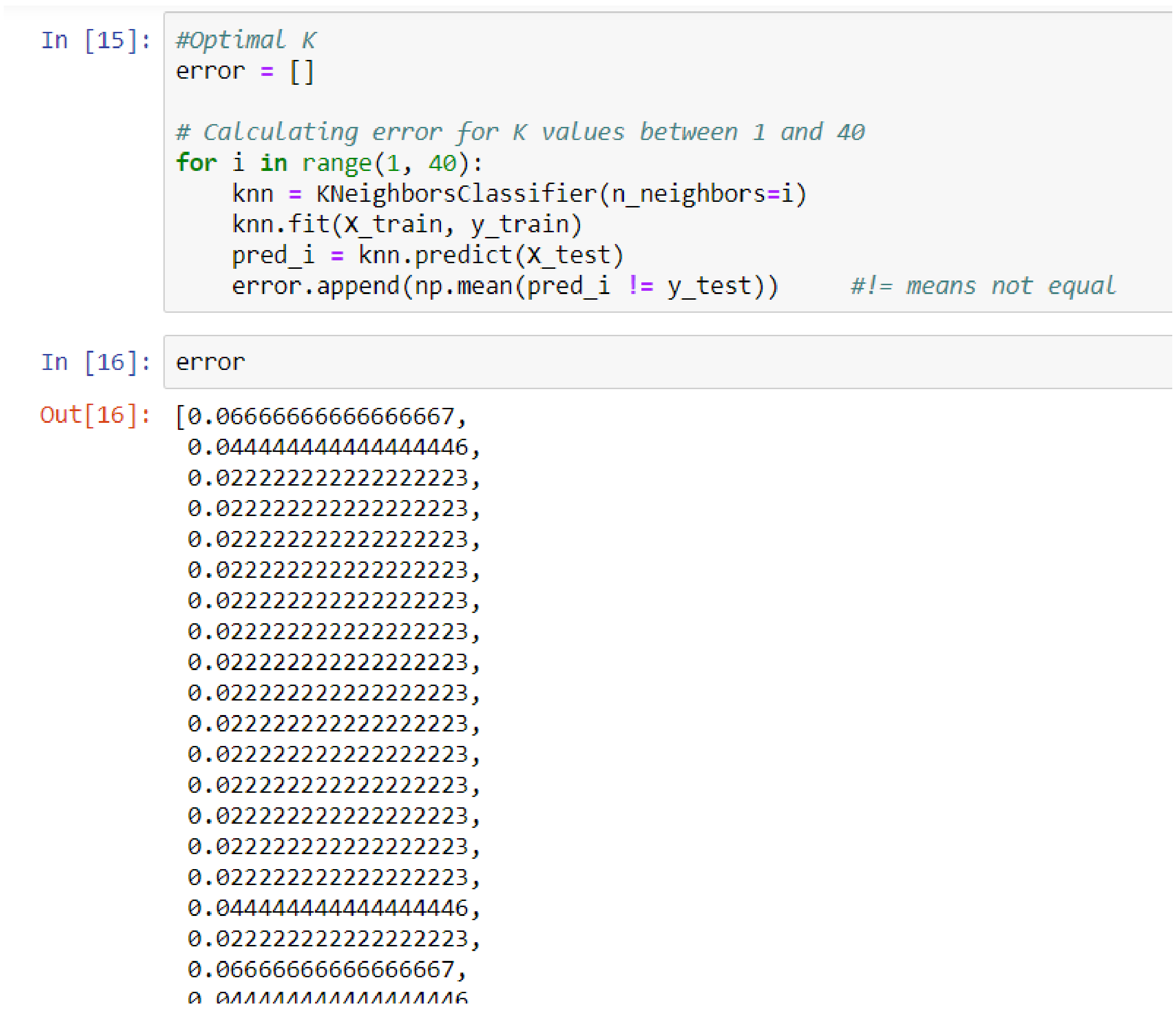


```
In [14]: from sklearn.metrics import accuracy_score
         print('KNN accuracy: {:.3f}'.format(accuracy_score(y_test, y_pred)))

         KNN accuracy: 0.978
```

Comparing Error Rate with the K Value

One way to help you find the best value of K is to plot the graph of K value and the corresponding error rate for the dataset.

In this section, we will plot the mean error for the predicted values of test set for all the K values between 1 and 40.

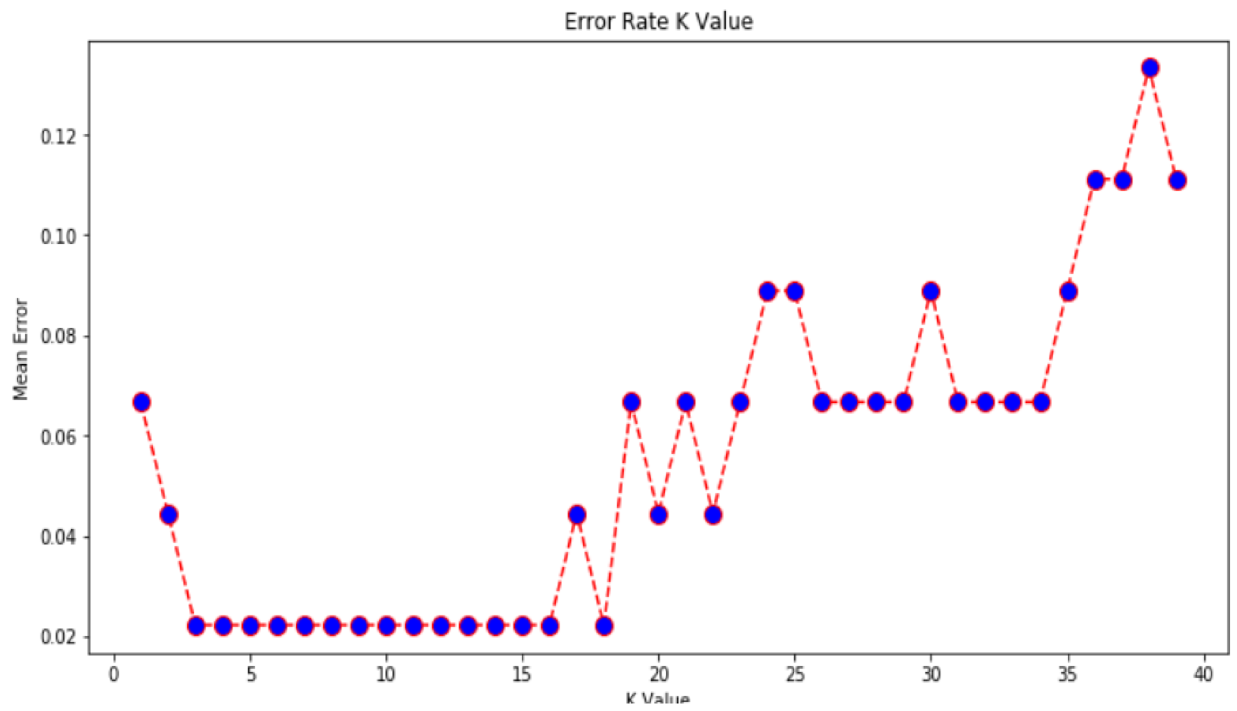


```
In [15]: #Optimal K
         error = []

         # Calculating error for K values between 1 and 40
         for i in range(1, 40):
             knn = KNeighborsClassifier(n_neighbors=i)
             knn.fit(X_train, y_train)
             pred_i = knn.predict(X_test)
             error.append(np.mean(pred_i != y_test))     #!= means not equal

In [16]: error

Out[16]: [0.06666666666666667,
          0.04444444444444446,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.022222222222222223,
          0.04444444444444446,
          0.022222222222222223,
          0.06666666666666667,
          0.04444444444444446
```

The above script executes a loop from 1 to 40. In each iteration the mean error for predicted values of test set is calculated and the result is appended to the error list.

The next step is to plot the error values against K values. Execute the following script to create the plot:

```
In [17]: plt.figure(figsize=(12, 6))
         plt.plot(range(1, 40), error, color='red', linestyle='dashed', marker='o',
                  markerfacecolor='blue', markersize=10)
         plt.title('Error Rate K Value')
         plt.xlabel('K Value')
         plt.ylabel('Mean Error')

Out[17]: Text(0, 0.5, 'Mean Error')
```



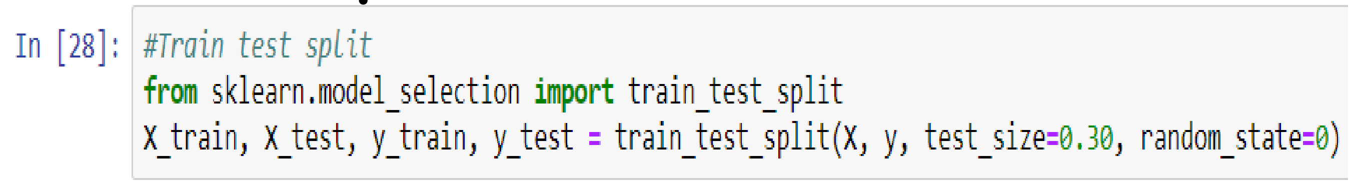
KNN REGRESSION:

After importing the libraries and the data as you have done in classification

the next step is **defining X and Y variables**:

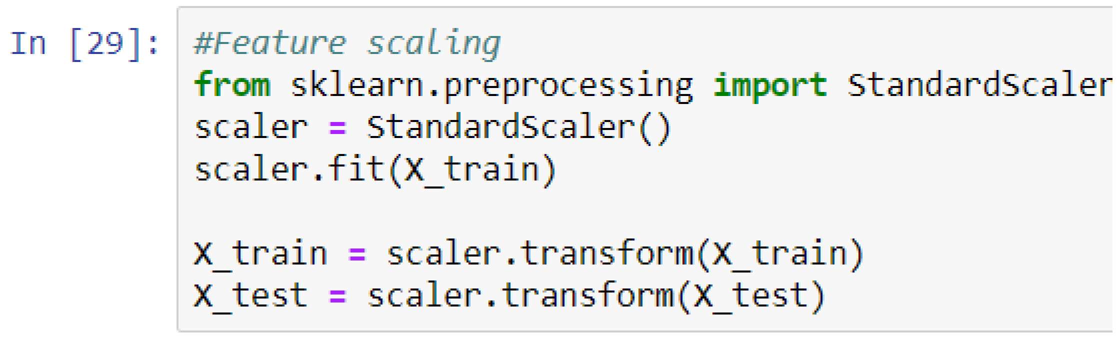
```
In [20]: X = dataset[["sepal-length", "sepal-width", "petal-length"]]
         y = dataset["petal-width"]
```

the next step is to **split the data**:



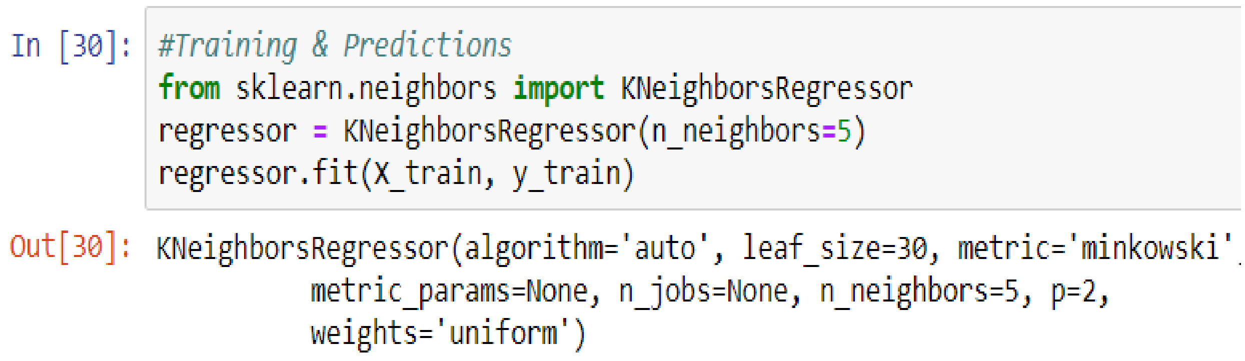
```
In [28]: #Train test split
         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)
```

feature scaling

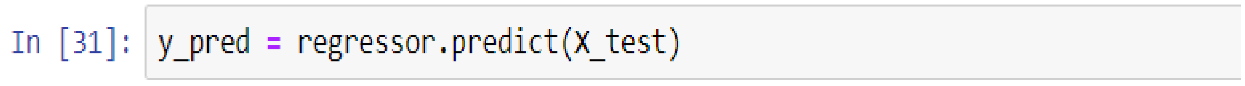


```
In [29]: #Feature scaling
         from sklearn.preprocessing import StandardScaler
         scaler = StandardScaler()
         scaler.fit(X_train)

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

Training and predictions

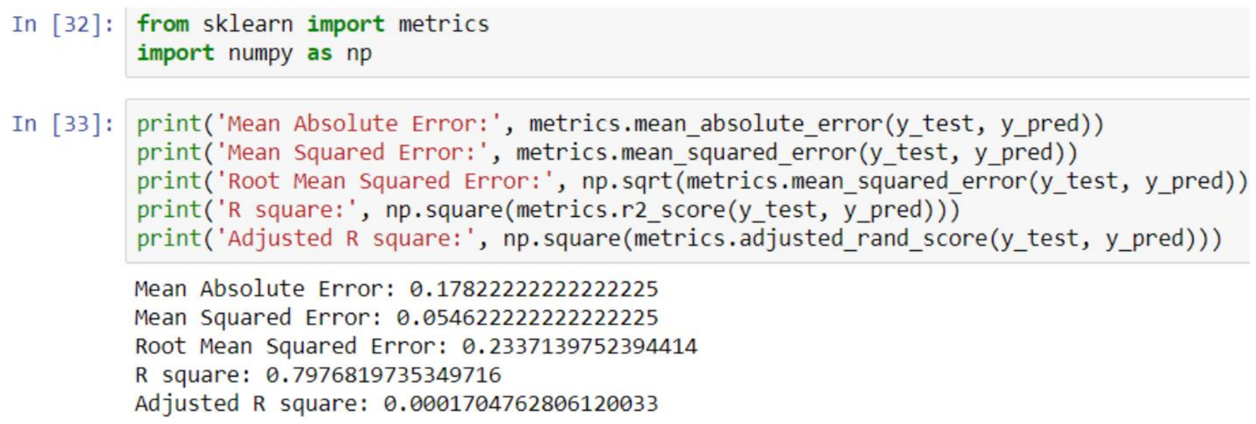


```
In [30]: #Training & Predictions
         from sklearn.neighbors import KNeighborsRegressor
         regressor = KNeighborsRegressor(n_neighbors=5)
         regressor.fit(X_train, y_train)

Out[30]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                   metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                   weights='uniform')
```



```
In [31]: y_pred = regressor.predict(X_test)
```

Evaluation



```
In [32]: from sklearn import metrics
         import numpy as np
```

```
In [33]: print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
         print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
         print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred))
         print('R square:', np.square(metrics.r2_score(y_test, y_pred)))
         print('Adjusted R square:', np.square(metrics.adjusted_rand_score(y_test, y_pred)))

         Mean Absolute Error: 0.17822222222222225
         Mean Squared Error: 0.054622222222222225
         Root Mean Squared Error: 0.2337139752394414
         R square: 0.7976819735349716
         Adjusted R square: 0.0001704762806120033
```