Implementing KNN Algorithm from Scratch in Python

Gist of the Theory

K-Nearest Neighbours algorithm is a simple, but powerful statistical learning method that is based on the Bayes Classifier, which uses the Bayes' Theorem of Conditional Probability to classify the training data. KNN algorithm is not limited to classification setting, it can also be applied to regression problems.

Bayes Classifier finds the probability that the response/output label *Y* is equal to *j* given the input attribute $X = x'$ and it's given by $P(Y=j \mid X=x')$.

K-Nearest Neighbour is an extension of this idea and calculates the above conditional probability as follows:

$$P(Y = j \mid X = x') = \frac{1}{K}\sum_{i \in \emptyset} I(y[i] = j)$$

Here, the outcome j is given by the K number of data points nearest to the input observation $X=x'$. The notation y[i] implies the i^{th} response variable, and the range of *i* is from 1 to K (denoted by the set ϕ in the above equation).
Here, measure of "nearness" is decided by Euclidean distance.

Prediction Rule for KNN Classifier:

Predicts the output class *Y* of an input *X* as the class which has the highest probability given by the above equation. Another way to get the predicted class is by taking a majority vote. For K = 3, the KNN algorithm selects 3 datapoints from the training set nearest to the test observation x'. Then, it picks the most-common class (represented by the output variable). In case all 3 classes are different, there are different ways to select only one of these 3 classes. In this scenario, the outcome depends on the problem at hand and the discretion of the data scientist.

Prediction Rule for KNN Regressor:

Predicts the output *Y* of an input *X* as the mean of the K-Nearest labels, and is given by a modified version of the above equation:

$$f(x') = \frac{1}{K}\sum_{i \in \emptyset} y[i]$$

Dataset

We are using the "glass data" dataset from Kaggle. You can download the dataset here: https://www.kaggle.com/prashant111/glass-identification-dataset

The dataset has details for 7 different types of glass (7-class-problem). Using the attributes, which are the various proportions of the chemical constituents of glass, we need to identify the type of the glass.

The attributes include:
ID – counter to track the number of observations
RI – Refractive Index
Na – Sodium
Mg – Magnesium
Al – Aluminium
Si – Silicon
K – Potassium
Ca – Calcium
Ba - Barium
Fe – Iron

Label:
Type of Glass – classes ranging from 1 to 7

1 – building windows float-processed
2 – building windows non-float-processed
3 – vehicle windows float processed
4 – vehicle windows non-float-processed (none in this database)
5 – containers
6 – tableware
7 – headlamps

The first column is not used in the data analysis as it has no meaning and does not contribute to the identification of the type of glass.

Implementation in Python

Step 1: Load the main Python Libraries and the dataset

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

X = np.genfromtxt("glass_data.csv", delimiter=",", usecols=np.arange(1,9), skip_header=1)
y = np.genfromtxt("glass_data.csv", delimiter=",", usecols=10, skip_header=1)
```

Step 2: Split dataset into training set and test set

We will use the trusty package from the sklearn, train_test_split, that automatically splits the dataset randomly into the training and the test sets, based on the random_state parameter

```
In [20]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=99)
```

Step 3: KNN Classifier in sklearn

SciPy library (sklearn package) in Python already has a fully implemented version of the KNN Classifier.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
print("Test set accuracy for 5NN: {}%".format(knn.score(X_test,y_test)*100))
print("Test Error Rate: {}%".format(np.mean(knn.predict(X_test) != y_test)*100))
```

```
Test set accuracy for 5NN: 74.07407407407408%
Test Error Rate: 25.925925925925924%
```

Step 5: Building the KNN Classifier from scratch

KNN Classifier and Regressor are examples of transduction. It means that there is no learning machine. We don't actually create a statistical model. We directly use the datapoints to create predictions.

1. We will combine the X_train and y_train numpy arrays into a matrix called model.

```
rows = X_train.shape[0]
columns = len(y_train.shape) + X_train.shape[1]
self.model = np.ones((rows,columns))
for i,x in enumerate(X_train):
    self.model[i] = self.model[i] * np.insert(x,len(x), np.array([y_train[i]]))
```

2. Then we will write a function that gives the Euclidean distance between 2 points.

```
def euclideanDistance(self,P1,P2):
    '''
    returns the Euclidean distance between two points
    '''
    return np.linalg.norm(P1-P2)
```

3. We use the above function to compute the distances between the test observation and the datapoints. Then we use a separate function to compute the K number of nearest neighbours to the test sample i.e. the training samples having the K-minimum distance from the test sample.

```
def KNearest(self,distances, K):
    '''
    picks the first K minimum distances irrespetive of ties
    '''
    for i in range(K):
        self.minimum_dist.append(min(distances))
        self.minimum_dist_index.append(np.argmin(distances))
        distances.pop(self.minimum_dist_index[i])
    return getKNearest
```

```
        test_sample_length = X_test.shape[1]
        for sample in X_test:
            #distance between the training sample from the training set
            #and the test sample from the test set
            distances = [ self.euclideanDistance(training_sample[:test_sample_length], sample)
                          for training_sample in self.model ]
            obj = getKNearest()
            obj.KNearest(distances, self.K)
            labels = [ self.model[i][-1] for i in obj.minimum_dist_index ]
            self.predicted_labels.append(max(set(labels), key = labels.count))
```

self.predicted_labels holds all the predicted classes for the observations in X_test. The prediction is made my selecting the most commonly occurring class among the K-neighbours.

4. Efficiency of the algorithm is paramount to assess whether it is making the correct prediction. For that we need to compare the predicted labels to the actual labels.

```
    score=[]
    score = [ self.predicted_labels[i] == y_test[i] for i in range(len(y_test))
              if self.predicted_labels[i] == y_test[i] ]
    return len(score)/len(y_test)
```

The variable score stores the percentage of correctly predicted labels. This is defined as accuracy.

Step 6: Using the KNN algorithm we created

We can now apply this algorithm to our dataset. We get the following results.

```
_3nn = KNN(K=3)
_3nn.fit(X_train, y_train)
y_pred = _3nn.predict(X_test)
score = _3nn.score(y_test)
print("Test set accurancy 5NN: {}% ".format(score*100))
error = np.mean(y_pred != y_test)
print("Test Error Rate: {}%".format(error*100))
```

```
Test set accurancy 5NN: 64.81481481481481%
Test Error Rate: 35.18518518518518%
```

Note that the accuracy is not that high. In fact, it is a bit lower than the KNN Classifier algorithm in the Python Machine Learning package sklearn which we checked above.

Why is the accuracy lower?
There can be a couple of reasons for that:

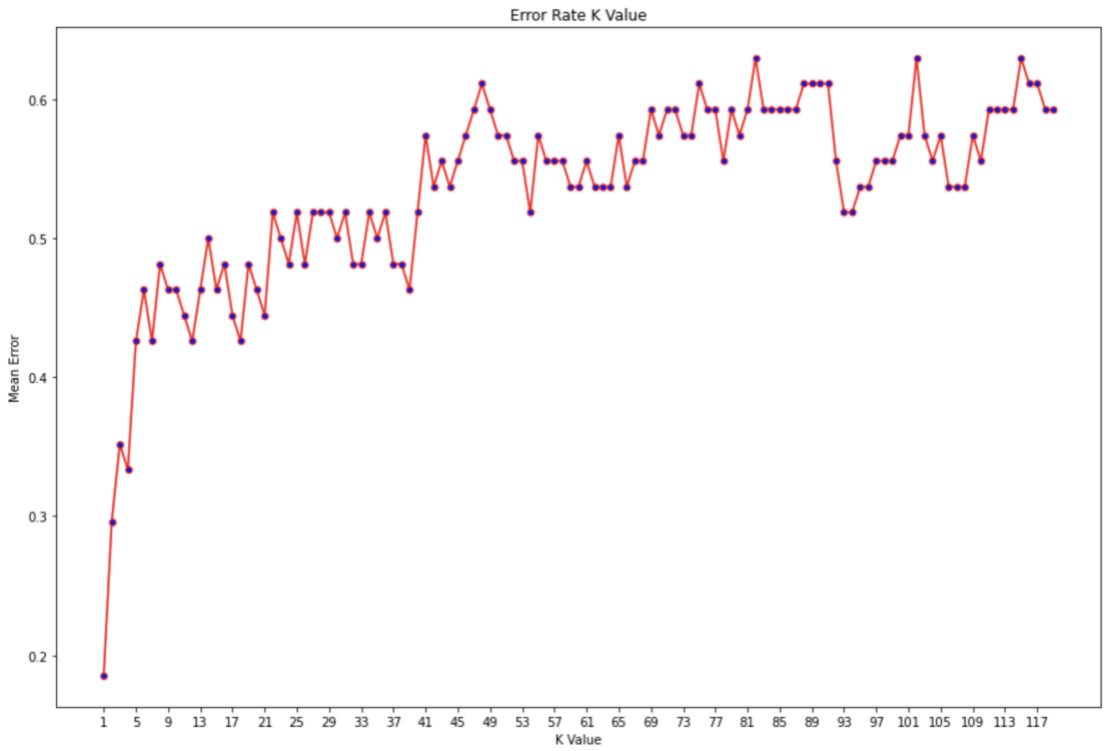
1. Note that in our KNN algorithm, we are not considering any tie-breaking technique. Let's consider a scenario, K=3, and we pick 3 nearest neighbours based on Euclidean distances [3,4,5]. What if the 4th neighbour is also having distance = 5 from the test sample. In this case, should we pick the 3rd training observation or the 4th one as the nearest neighbour? These 2 observations can have different labels.
2. We are also not considering the scenario where each of the K-nearest output labels is having a different class. In that case, which one should we pick?

Analyzing the Dependence on K

Is K = 3 an optimal value of K? What happens when we increase or decrease the value of K?

```
errors,scores, accuracy = [],[], []
for i in range(1, 120):
    KNearest = KNN(K=i)
    KNearest.fit(X_train, y_train)
    prediction_i = KNearest.predict(X_test)
    scores.append(KNearest.score(y_test))
    errors.append(np.mean(prediction_i != y_test))
    accuracy.append(1-np.mean(prediction_i != y_test))
```

```
plt.figure(figsize=(20,16))
plt.plot(range(1, 120), errors, color='red', marker='o', markerfacecolor='blue', markersize=5)
plt.title('Error Rate K Value')
plt.xlabel('K Value')
plt.ylabel('Mean Error')
plt.xticks(np.arange(1,121,4))
plt.show()
```



The above graph shows that the mean error is lowest for lower values of K. But K=1 causes overfitting, because we use only 1 neighbour to predict the class outcome. So, the predictions will depend heavily on the selected training data and may not perform well on unseen test data. This is a situation of high variance.

In terms of bias-variance trade-off, it is empirically proven that K =3 or K =5 gives the best results. It is not too dependent on the training data and gives good prediction for the test data. However, the correct choice of K will also depend on the problem at hand.

Full Code: https://github.com/jayasmitachakraborty/KNN-from-scratch-python