K-Nearest Neighbor (KNN)

Based on KNN Classification using SciKit-learn

K-Nearest Neighbor (KNN) is a simple, versatile and one of the topmost machine learning algorithms. KNN is used in a variety of applications such as finance, healthcare, political science, and image recognition. For example, in loan disbursement banking institutes will predict whether the loan is safe or risky. In political science, classifying potential voters in two classes will vote or won't vote.

KNN algorithm is used for both classification and regression problems. KNN algorithms are based on the feature similarity approach.

Learning outcomes:

- KNN algorithm
- Eager Vs Lazy learners
- How to decide the number of neighbors K?
- Curse of Dimensionality
- Building KNN classifiers using Python Scikit-learn package
- KNN performance
- Pros and Cons

K-Nearest Neighbors

KNN is a non-parametric and lazy learning algorithm.

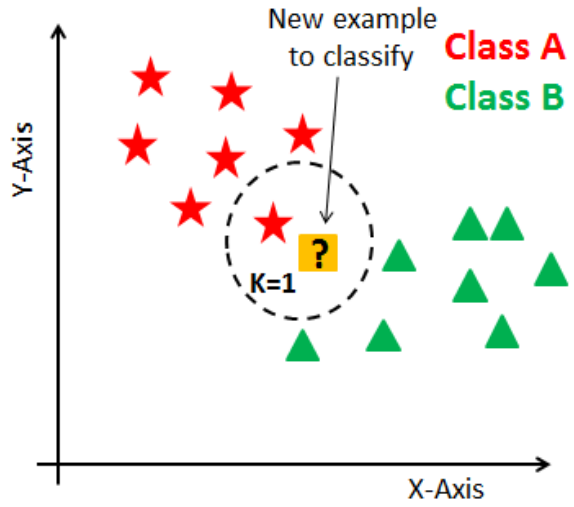
Non-parametric means there is no assumption for the underlying data distribution. In other words, the model structure is determined from the dataset. This will be very helpful in practice since most of the real world datasets do not follow mathematical theoretical assumptions.

Lazy algorithm means it does not need any training data points for model generation. All training data is used in the testing phase. This makes the training phase faster and the testing phase slower and costlier in terms of memory requirements (dataset = training data + test data).

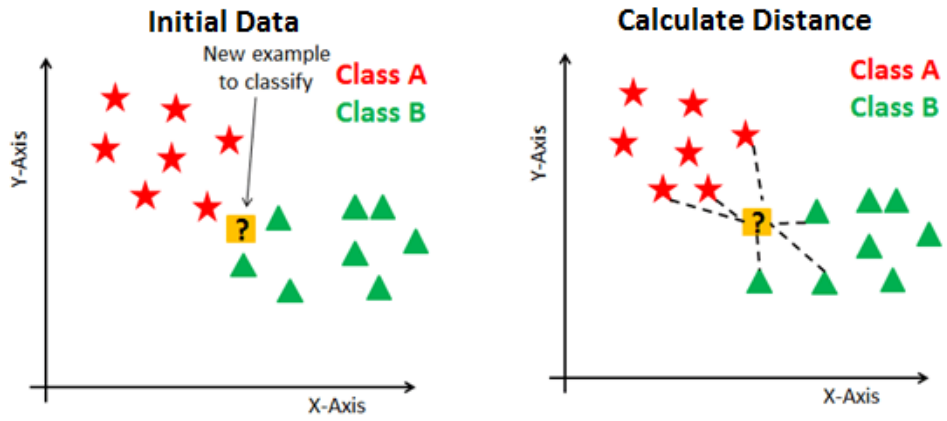
How does the KNN algorithm work?

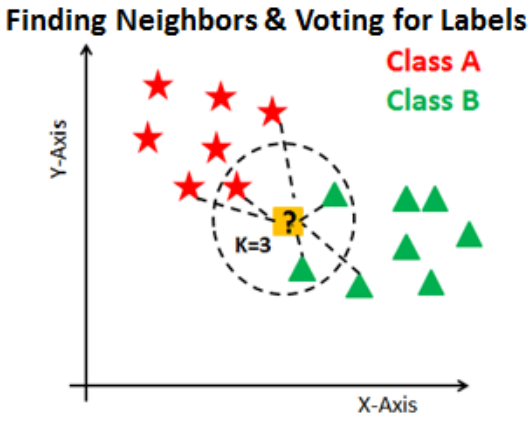
The value K represents the number of nearest neighbors. This value is the core deciding factor of the algorithm.

When K=1, the algorithm is known as the Nearest Neighbor algorithm. This is the simplest case. Suppose P1 (represented with ? in the figure below) is the point we want to predict the label to. To do so, you find the closest point to P1 and then you label P1 with the nearest point's label.



In general, given k you find the k closest neighbor points to P1 and then you classify P1 depending on the k closest neighbors' labels. Each point votes for its label (since it represents its class) and the label with the most votes is taken to be the predicted label of P1.





To evaluate the closest points distance, you can use a variety of distance measures such as Euclidean distance, Hamming distance, Manhattan distance and Minkowski distance.

The KNN algorithm has the following basic steps:

1. Calculate distance
2. Find closest neighbors
3. Vote for labels

Eager Vs. Lazy Learners

A learner is called eager when it constructs the predictive model from the training set before performing any prediction. One can think of such learners as being ready, active and eager to classify the training data.

In contrast, lazy learning do not construct any prediction model at all and all data points are used at prediction time. Lazy learners wait until the last minute before classifying any data point. Lazy learners merely store the entire training set and wait until new data need to be classified. Only then

the lazy learner performs generalization to classify the new data based on its similarity to the stored training data. Unlike eager learning methods, lazy learners do less work in the training phase and all the work in the testing phase. Lazy learners are also known as instance-based learners because lazy learners store the training points or instances, and all learning is based on instances.

Curse of Dimensionality

KNN performs better with a lower number of features rather than a large number. When the number of features increases, KNN requires more data. The increase in features dimensions may lead to the problem of overfitting. To avoid it, the training data has to grow exponentially as you increase the number of dimensions. This problem of higher dimension is known as the Curse of Dimensionality.

To deal with the problem of the curse of dimensionality, you can perform the Principal Component Analysis (PCA) before applying any machine learning algorithm. Alternatively, you can use a feature selection approach.

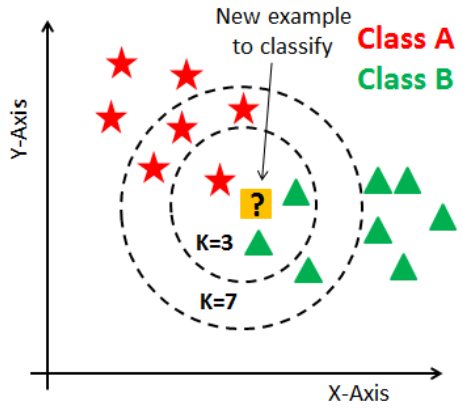
Research studies have shown that in large dimensions Euclidean distance is not useful. Therefore, you can prefer other measures such as cosine similarity which is less affected by high dimensions.

How do you decide the number of neighbors?

At this point a question arises: How to choose the optimal number of neighbors? And what are its effects on the classifier? The number of neighbors k in KNN is a hyperparameter that you need to choose at the time of the model building. You can think of k as a controlling variable for the prediction model.

Unfortunately, no optimal number of neighbors suits all kind of data sets. Each dataset has its own requirements. In the case of a small number k, the noise will have a higher influence on the result. Large value of k makes the algorithm computationally expensive. Also small values for k produce models with the most flexible fit which will have low bias but high variance. Instead, a large value of k will produce a smoother decision boundary with lower variance but higher bias.

Generally, data analitycs choose an odd number for k if the number of classes is even. Then, they generate the model for different values of k and test the corresponding model performance. One can also try the Elbow method here.



Building a KNN Classifier in Scikit-learn

Defining dataset

Let's first create our own dataset. Here we need two kinds of attributes or columns in our data: Feature and Label. The reason for two type of column is the supervised nature of KNN algorithm.

```
# Assigning features and label variables

# First Feature

weather=['Sunny','Sunny','Overcast','Rainy','Rainy','Rainy','Overcast','Sunny','Sunny',

'Rainy','Sunny','Overcast','Overcast','Rainy']

# Second Feature

temp=['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild','Mild','Hot','Mild']

# Label or target variable

play=['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Yes','No']
```

In this dataset, you have two features (weather and temperature) and one label (play).

Encoding data columns

Various machine learning algorithms require numerical input data so you need to represent categorical columns in numerical columns.

In order to encode this data, you could map each value to a number
Overcast:0, Rainy:1, Sunny:2

This process is known as label encoding. sklearn conveniently does this for you with Label Encoder.

```
# Import LabelEncoder
from sklearn import preprocessing
# creating labelEncoder
le = preprocessing.LabelEncoder()
# Converting string labels into numbers
weather_encoded=le.fit_transform(weather)
print(weather_encoded)
```

```
[2 2 0 1 1 1 0 2 2 1 2 0 0 1]
```

Here, you imported the preprocessing module and created the LabelEncoder object. It can be used to fit and transform the "weather" column into a corresponding numeric column. Similarly, you can encode temperature and play.

```
# converting string labels into numbers
temp_encoded=le.fit_transform(temp)
label=le.fit_transform(play)
```

Combining Features

You can combine multiple columns (or features) into a single set of data using the zip function

```
#combinig weather and temp into single listof tuples
features=list(zip(weather_encoded,temp_encoded))
```

Generating the Model

Now let's build the KNN classifier model.

First, import the KNeighborsClassifier module and create KNN classifier object by specifying the number of neighbors k. Then, fit the model with the train set using the fit( ) function and perform prediction on the test set with predict( ).

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training sets

model.fit(features,label)

# Predict Output

predicted= model.predict([[0,2]])    # 0:Overcast,  2:Mild

print("Prediction: " ,predicted)
```

```
[1]
```

In the above example, you have given input [0,2], where 0 means Overcast weather and 2 means Mild temperature. The model predicts 1, which means play.

KNN with Multiple Labels

Here you will learn about KNN with multiple labels where each label corresponds to a different class.

We will use the wine dataset (that is available from scikit-learn) as a running example since it is a famous multi-class classification problem. The data from wine dataset is the result of a chemical analysis of wines grown in the same region in Italy using three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The dataset comprises 13 features ('alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline') and a target (type of cultivars).

The data has three types of cultivar classes: 'class_0', 'class_1', and 'class_2'. Here, you want to build a model to classify the type of cultivar.

Loading Data

Let's first load the required wine dataset from scikit-learn datasets.

```
# Import scikit-learn dataset library

from sklearn import datasets

# Load dataset

wine = datasets.load_wine()
```

Exploring Data

After you loaded the dataset, you might want to know a little bit more about it. You can check feature and target names.

```
# print the names of the features

print(wine.feature_names)
```

```
['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids',

'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue',

 'od280/od315_of_diluted_wines', 'proline']
```

```
# print the label species(class_0, class_1, class_2)

print(wine.target_names)
```

```
['class_0' 'class_1' 'class_2']
```

Let's check top 5 records of the feature set.

```
# print the wine data (top 5 records)

print(wine.data[0:5])
```

```
[[  1.42300000e+01   1.71000000e+00   2.43000000e+00   1.56000000e+01

    1.27000000e+02   2.80000000e+00   3.06000000e+00   2.80000000e-01
```

```
   2.29000000e+00   5.64000000e+00   1.04000000e+00   3.92000000e+00
   1.06500000e+03]
 [  1.32000000e+01   1.78000000e+00   2.14000000e+00   1.12000000e+01
   1.00000000e+02   2.65000000e+00   2.76000000e+00   2.60000000e-01
   1.28000000e+00   4.38000000e+00   1.05000000e+00   3.40000000e+00
   1.05000000e+03]
 [  1.31600000e+01   2.36000000e+00   2.67000000e+00   1.86000000e+01
   1.01000000e+02   2.80000000e+00   3.24000000e+00   3.00000000e-01
   2.81000000e+00   5.68000000e+00   1.03000000e+00   3.17000000e+00
   1.18500000e+03]
 [  1.43700000e+01   1.95000000e+00   2.50000000e+00   1.68000000e+01
   1.13000000e+02   3.85000000e+00   3.49000000e+00   2.40000000e-01
   2.18000000e+00   7.80000000e+00   8.60000000e-01   3.45000000e+00
   1.48000000e+03]
 [  1.32400000e+01   2.59000000e+00   2.87000000e+00   2.10000000e+01
   1.18000000e+02   2.80000000e+00   2.69000000e+00   3.90000000e-01
   1.82000000e+00   4.32000000e+00   1.04000000e+00   2.93000000e+00
   7.35000000e+02]]
```

Let's check the records of the target set.

```
# print the wine labels (0:Class_0, 1:Class_1, 2:Class_3)
print(wine.target)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

Let's explore it for a bit more. You can check the shape of the dataset.

```
# print data(feature)shape

print(wine.data.shape)
```

```
(178, 13)
```

```
# print target(or label)shape

print(wine.target.shape)
```

```
(178,)
```

Splitting Data

To understand the model performance you can divide the dataset into a training set and a test set .

Let's split the dataset by using the function train_test_split( ). You need to specify three parameters: the features, the target, and the test_set size. Additionally, you can use random_state to select records randomly.

```
# Import train_test_split function

from sklearn.model_selection import train_test_split

# Split dataset into training set and test set: 70% training and 30% test

X_train, X_test, y_train, y_test = train_test_split(wine.data, wine.target, test_size=0.3)
```

Generating the Model for K=5

Let's build KNN classifier model for k=5.

```
from sklearn.neighbors import KNeighborsClassifier

# Create KNN Classifier

knn = KNeighborsClassifier(n_neighbors=5)

# Train the model using the training sets

knn.fit(X_train, y_train)

# Predict the response for test dataset

y_pred = knn.predict(X_test)
```

Model Evaluation for k=5

Let's estimate how accurately the classifier or model can predict the type of cultivars.

Accuracy can be computed by comparing actual test set values and the predicted values.

```
# Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

# Model Accuracy, how often is the classifier correct?

print("Accuracy for k=5: ",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy for k=5: 0.685185185185
```

Well, you got a classification rate of 68.51%, considered as good accuracy.

For further evaluation, you can also create a model for a different number of neighbors.

Regenerating Model for K=7

Let's build KNN classifier model for k=7.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=7)

knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
```

Model Evaluation for k=7

Let's again estimate how accurately the classifier can predict the type of cultivars.

```
from sklearn import metrics

print("Accuracy for k=7: ",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy for k=7: 0.777777777778
```

You received a classification rate of 77.77%, considered as good accuracy. Here you have increased the number of neighbors in the model and the accuracy increased. Unfortunately, this is not always the case. For selecting a good value for k, you can use the Elbow method.

Pros

The training phase of K-nearest neighbor classification is much faster compared to other classification algorithms. There is no need to train a model for generalization at all. That is why KNN is also known as instance-based learning algorithm. KNN can be useful in case of nonlinear data. It can also be used with regression problems. The predicted value of the new data point is computed by calculating the average of the k closest neighbors values.

Cons

The testing phase of KNN is slower and costlier in terms of time and memory. It requires large memory for storing the entire training dataset. KNN requires the rescaling of the data set in particular if the Euclidean distance measure is used. The Euclidean distance is sensitive to magnitudes; the features with high magnitudes will weight more than the features with low magnitudes. KNN is not suitable for large dimensional data sets.

How to improve KNN?

For better results, normalizing the data on the same scale is highly recommended. Generally, the normalization range considered isbetween 0 and 1. KNN is not suitable for large dimensional data set. In such cases, dimensions need to be reduced to improve the performance. Also, handling missing values will help us in improving the results.