Dharavath Ramdas

Datset link: https://github.com/dharavathramdas101/Machine-learning-Algorithms-Practical-implementation
Source Code link: https://github.com/dharavathramdas101/Machine-learning-Algorithms-Practical-implementation
linkedin: https://www.linkedin.com/in/dharavath-ramdas-a283aa213/

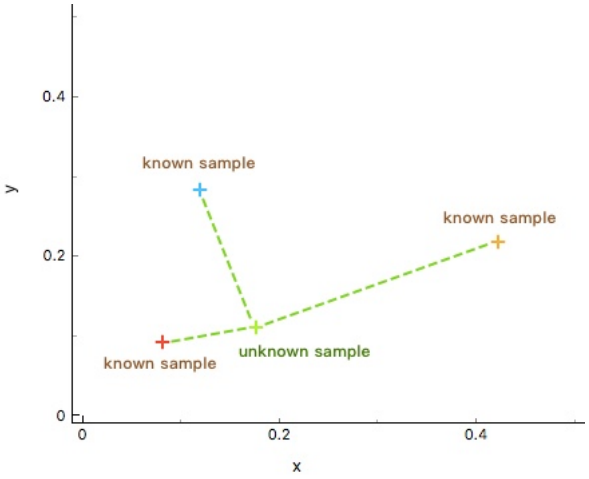
K Nearest Neighbour Algorithm

Nearest Neighbor Algorithm
Before introducing the K-nearest neighbors algorithm, let's first talk about the nearest neighbor (NN) algorithm. It finds the training sample y most similar to x in the training set and uses the category of y as the category of x to achieve the classification aim:

In [61]:

```
Xy1i7cZaTlBL-gUs8ke8iK5XinmB8k7lYb1ZxZNeOgF4INnqQIaBS8q13lyCmUGFcsUy0AGU.-G9kpQJJo43UrFSpwWO1Mg/__results___files/__results___13_0.jpeg")
```

Out[61]:



As shown in the figure above, by calculating the distance between data Xu (unknown sample) and the known categories ω1,ω2,ω3 (known samples), we judge the similarity between Xu and different training sets, and finally determine the category of Xu . Obviously, it is more appropriate to match the green unknown sample and the red known samples.

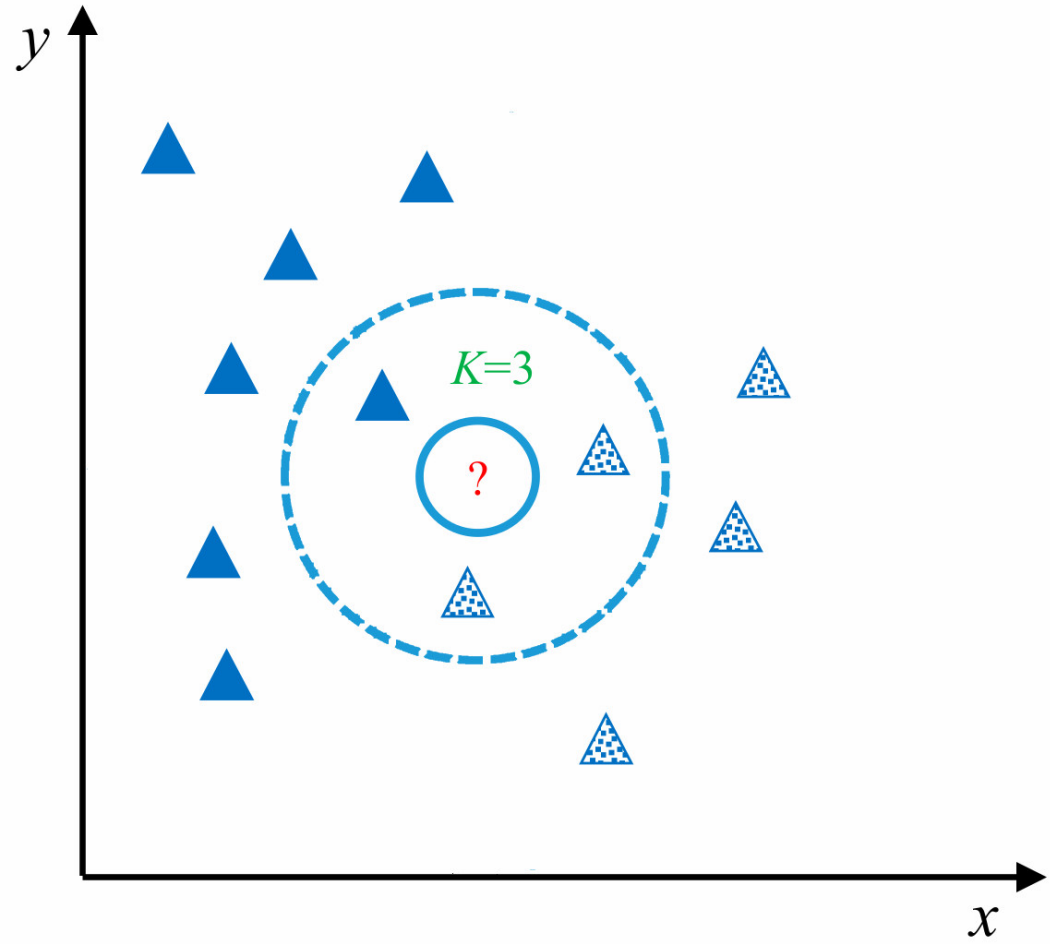
K-Nearest Neighbors Algorithm

The K-Nearest Neighbors (KNN) algorithm is a generalization of the nearest neighbor (NN) algorithm and one of the simplest methods in the machine learning classification algorithm. The core idea of the KNN algorithm is similar to that of the nearest neighbor algorithm, which is classified by finding categories similar to the unknown samples. However, in the NN algorithm, only one sample is used for decision. When the classification is too absolute, the classification effect is poor. To overcome the defect, the KNN algorithm uses K neighboring samples to jointly decide the categories of unknown samples. In this way, the fault tolerance rate in decision-making is much higher than that of the NN algorithm and the classification effect is better.

In [63]:

```
Us8ke8iK5XinmB8k7lYb1ZxZNeOgF4INnqQIaBS8q13lyCmUGFcsUy0AGU.-G9kpQJJo43UrFSpwWO1Mg/__results___files/__results___17_0.jpeg",width="800px")
```

Out[63]:



As shown in the figure above, for the unknown test sample (shown in ?), the KNN algorithm is used for classification. First calcu late the similarity between the unknown sample and the training sample, and then find out the nearest K adjacent samples. (K valu e is 3 in the figure, and ? is circled by K nearest points.) Finally the category of the unknown sample is judged based on the ne arest K samples.

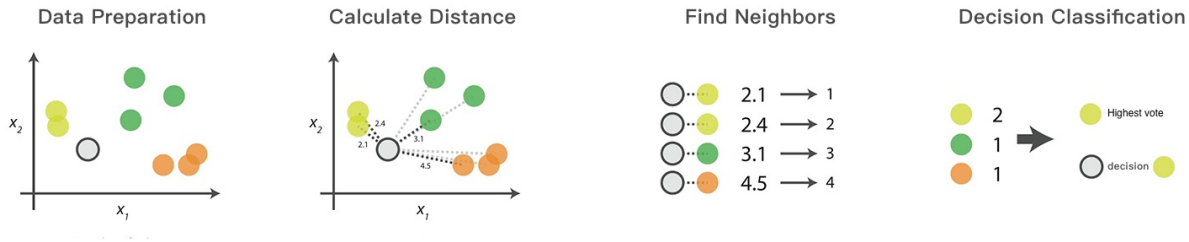
Implementation of KNN
The KNN algorithm is very mature in theory. Its simple, easy-to-understand ideas and good classification accuracy make it widely adopted. The specific process of the algorithm mainly contains the following four steps:

Data Preparation: Through data cleaning, data processing, each piece of data is organized into vectors.
Calculate Distance: Calculate the distance between test data and training data.
Find Neighbors: Find the K training data samples closest to the test data.
Decision Classification: According to the decision rule, the category of test data is obtained from K neighbors.

In [64]:

```
Xy1i7cZaTlBL-gUs8ke8iK5XinmB8k7lYb1ZxZNeOgF4INnqQIaBS8q13lyCmUGFcsUy0AGU.-G9kpQJJo43UrFSpwWO1Mg/__results___files/__results___23_0.jpeg")
```

Out[64]:



In [ ]:

Dataset information

The different columns present in the dataset are:

1.Pregnancies -> Number of times Pregnant

2.Glucose -> Plasma glucose concentration

3.BloodPressure -> Diastolic blood pressure (mm Hg)

4.SkinThickness -> Triceps skin fold thickness (mm)

5.Insulin -> 2-Hour serum insulin (mu U/ml)

6.BMI -> Body Mass Index

7.DiabetesPedigreeFunction -> Diabetes pedigree function

8.Age -> Age in years

9.Outcome -> Whether the lady is diabetic or not, 0 represents the person is not diabetic and 1 represents that the person is dia
betic.

In [ ]:

Load the python liberies

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Load the dataset

In [3]:

```
df = pd.read_csv(r"C:\Users\DHARAVATH RAMDAS\Downloads\archive (5)\diabetes.csv")
```

First five rows

In [5]:

```
df.head()
```

Out[5]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Shape of dataframe

In [7]:

```
df.shape
```

Out[7]:

```
(768, 9)
```

Observations:

we observed the total 768 rows and 9 columns (8 columns are independent and 9th columns is dependent/label)

Information

In [51]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Describe

In [54]:

```
df.describe().T
```

Out[54]:

	count	mean	std	min	25%	50%	75%	max
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

In [58]:

```
df.isnull().sum()
```

Out[58]:

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Countplot

Outcome

In [57]:

```
df['Outcome'].value_counts()
```

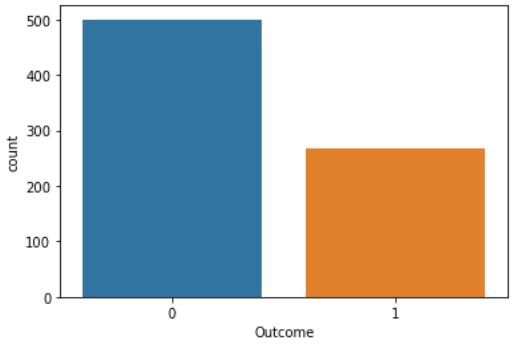
Out[57]:

```
0    500
1    268
Name: Outcome, dtype: int64
```

In [56]:

```
sns.countplot(df['Outcome'])
plt.show()
```

```
C:\Users\DHARAVATH RAMDAS\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable
as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments with
out an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```



In [ ]:

```

```

Seperate dependent and independent features

In [11]:

```
X = df.drop(['Outcome'],axis=1)
y = df['Outcome']
```

In [12]:

```
print(X.shape,y.shape)
```

```
(768, 8) (768,)
```

Split dataset into train and test split

In [14]:

```
from sklearn.model_selection import train_test_split
```

In [15]:

```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state = 16)
```

In [16]:

```
print("train shape :", X_train.shape,y_train.shape)
```

```
train shape : (537, 8) (537,)
```

In [17]:

```
print("test shape :",X_test.shape,y_test.shape)
```

```
test shape : (231, 8) (231,)
```

Import KNeighborsClassifier

In [18]:

```
from sklearn.neighbors import KNeighborsClassifier

#Setup arrays to store training and test accuracies
neighbors = np.arange(1,9)
train_accuracy =np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

for i,k in enumerate(neighbors):
    #Setup a knn classifier with k neighbors
    knn = KNeighborsClassifier(n_neighbors=k)

    #Fit the model
    knn.fit(X_train, y_train)

    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)

    #Compute accuracy on the test set
    test_accuracy[i] = knn.score(X_test, y_test)
```

In [19]:

```
train_accuracy
```

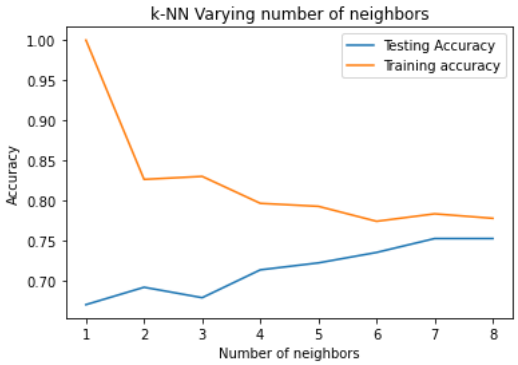
Out[19]:

```
array([1.        , 0.82681564, 0.83054004, 0.79702048, 0.79329609,
       0.77467412, 0.7839851 , 0.77839851])
```

Generate PLot

In [20]:

```
plt.title('k-NN Varying number of neighbors')
plt.plot(neighbors, test_accuracy, label='Testing Accuracy')
plt.plot(neighbors, train_accuracy, label='Training accuracy')
plt.legend()
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
plt.show()
```



In [21]:

```
#Setup a knn classifier with k neighbors
knn = KNeighborsClassifier(n_neighbors=7)
```

In [22]:

```
knn.fit(X_train,y_train)
```

Out[22]:

```
KNeighborsClassifier(n_neighbors=7)
```

Prediction

In [29]:

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

Out[29]:

```
array([1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0,
       1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
       1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,
       0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
       1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0], dtype=int64)
```

In [47]:

```
from sklearn.metrics import accuracy_score
```

Accuracy Score

In [49]:

```
accuracy_score(y_pred,y_test)
```

Out[49]:

```
0.7532467532467533
```

In [24]:

```
#Get accuracy.
knn.score(X_test,y_test)
```

Out[24]:

```
0.7532467532467533
```

Confusion Matrix

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. Scikit-learn provides facility to calculate confusion matrix using the confusion_matrix method.

In [25]:

```
from sklearn.metrics import confusion_matrix
```

In [31]:

```
## confusion matrix
```

In [33]:

```
confusion_matrix(y_pred,y_test)
```

Out[33]:

```
array([[127,  37],
       [ 20,  47]], dtype=int64)
```

Considering confusion matrix above:

True negative = 127

False positive = 37

True postive = 20

Fasle negative = 47

In [34]:

```
#Confusion matrix can also be obtained using crosstab method of pandas.

pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True)
```

Out[34]:

Predicted	0	1	All
True			
0	127	20	147
1	37	47	84
All	164	67	231

Classification Report

Another important report is the Classification report. It is a text summary of the precision, recall, F1 score for each class. Sc ikit-learn provides facility to calculate Classification report using the classification_report method.

In [35]:

```
from sklearn.metrics import classification_report
```

In [37]:

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

```
              precision    recall  f1-score   support

           0       0.77      0.86      0.82       147
           1       0.70      0.56      0.62        84

    accuracy                           0.75       231
   macro avg       0.74      0.71      0.72       231
weighted avg       0.75      0.75      0.75       231
```

ROC (Reciever Operating Charecteristic) curve

It is a plot of the true positive rate against the false positive rate for the different possible cutpoints of a diagnostic test.

An ROC curve demonstrates several things:

1) It shows the tradeoff between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in sp ecificity).

2) The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test.

3)The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.

4) The area under the curve is a measure of test accuracy.

In [38]:

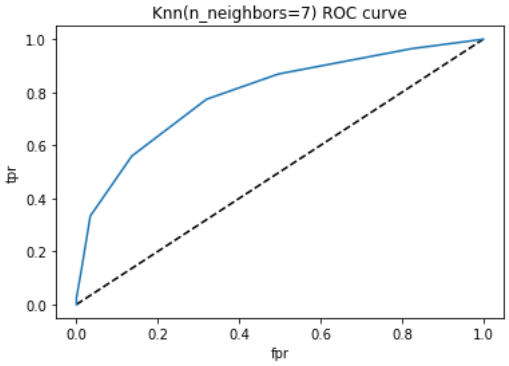
```
y_pred_proba = knn.predict_proba(X_test)[:,1]
```

In [39]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
```

In [40]:

```
plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='Knn')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('Knn(n_neighbors=7) ROC curve')
plt.show()
```



In [41]:

```
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred_proba)
```

Out[41]:

```
0.7923145448655653
```

Cross Validation

Now before getting into the details of Hyperparamter tuning, let us understand the concept of Cross validation.

The trained model's performance is dependent on way the data is split. It might not representative of the model's ability to gene ralize.

The solution is cross validation.

Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train th e model, and a test set to evaluate it.

In k-fold cross-validation, the original sample is randomly partitioned into k equal size subsamples. Of the k subsamples, a sing le subsample is retained as the validation data for testing the model, and the remaining k-1 subsamples are used as training dat a. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the vali dation data. The k results from the folds can then be averaged (or otherwise combined) to produce a single estimation. The advant age of this method is that all observations are used for both training and validation, and each observation is used for validatio n exactly once.

Hyperparameter tuning

The value of k (i.e 7) we selected above was selected by observing the curve of accuracy vs number of neighbors. This is a primit ive way of hyperparameter tuning.

There is a better way of doing it which involves:

1) Trying a bunch of different hyperparameter values

2) Fitting all of them separately

3) Checking how well each performs

4) Choosing the best performing one

5) Using cross-validation every time

Scikit-learn provides a simple way of achieving this using GridSearchCV i.e Grid Search cross-validation.

In [42]:

```
from sklearn.model_selection import GridSearchCV
```

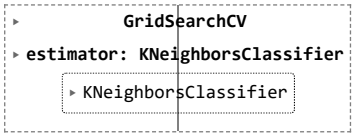
In [43]:

```
#In case of classifier like knn the parameter to be tuned is n_neighbors
param_grid = {'n_neighbors':np.arange(1,50)}
```

In [44]:

```
knn = KNeighborsClassifier()
knn_cv= GridSearchCV(knn,param_grid,cv=5)
knn_cv.fit(X,y)
```

Out[44]:



In [45]:

```
knn_cv.best_score_
```

Out[45]:

```
0.7578558696205755
```

In [46]:

```
knn_cv.best_params_
```

Out[46]:

```
{'n_neighbors': 14}
```

Thus a knn classifier with number of neighbors as 14 achieves the best score/accuracy of 0.7578

Thank you