k-Nearest Neighbors

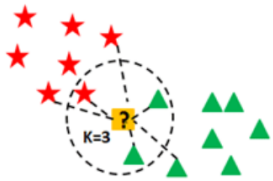
K-nearest neighbors (KNN) is a type of supervised learning algorithm which is used for both regression and classification purposes, but mostly it is used for the later. Given a dataset with different classes, KNN tries to predict the correct class of test data by calculating the distance between the test data and all the training points. It then selects the k points which are closest to the test data. Once the points are selected, the algorithm calculates the probability (in case of classification) of the test point belonging to the classes of the k training points and the class with the highest probability is selected. In the case of a regression problem, the predicted value is the mean of the k selected training points.

Let's understand this with an illustration:

1) Given a training dataset as given below. We have a new test data that we need to assign to one of the two classes.

2) Now, the k-NN algorithm calculates the distance between the test data and the given training data.

3) After calculating the distance, it will select the k training points which are nearest to the test data. Let's assume the value of k is 3 for our example.



4) Now, 3 nearest neighbors are selected, as shown in the figure above. Let's see in which class our test data will be assigned :

Number of Green class values = 2 Number of Red class values = 1 Probability(Green) = 2/3 Probability(Red) = 1/3

Since the probability for Green class is higher than Red, the k-NN algorithm will assign the test data to the Green class.

Similarly, if this were the case of a regression problem, the predicted value for the test data will simply be the mean of all the 3 nearest values.

This is the basic working algorithm for k-NN. Let's understand how the distance is calculated :

Euclidean Distance:

It is the most commonly used method to calculate the distance between two points. The Euclidean distance between two points 'p(p1,p2)' and 'q(q1,q2)' is calculated as :

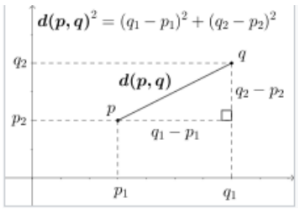


image source : Wikipedia

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

Similarly,for n-dimensional space, the Euclidean distance is given as :

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

Lazy Learners

k-NN algorithms are often termed as Lazy learners. Let's understand why is that. Most of the algorithms like Bayesian classification, logistic regression, SVM etc., are called Eager learners. These algorithms generalize over the training set before receiving the test data i.e. they create a model based on the training data before receiving the test data and then do the prediction/classification on the test data. But this is not the case with the k-NN algorithm. It doesn't create a generalized model for the training set but waits for the test data. Once test data is provided then only it starts generalizing the training data to classify the test data. So, a lazy learner just stores the training data and waits for the test set. Such algorithms work less while training and more while classifying a given test dataset.

Pros and Cons of k-NN Algorithm

Pros:

- It can be used for both regression and classification problems.
- It is very simple and easy to implement.
- Mathematics behind the algorithm is easy to understand.
- There is no need to create model or do hyperparameter tuning.
- KNN doesn't make any assumption for the distribution of the given data.
- There is not much time cost in training phase.

Cons:

- Finding the optimum value of 'k'
- It takes a lot of time to compute the distance between each test sample and all training samples.
- Since the model is not saved beforehand in this algorithm (lazy learner), so every time one predicts a test value, it follows the same steps again and again.
- Since, we need to store the whole training set for every test set, it requires a lot of space.
- It is not suitable for high dimensional data.
- Expensive in testing phase

Different ways to perform k-NN

Above we studied the way k-NN classifies the data by calculating the distance of test data from each of the observations and selecting 'k' values. This approach is also known as "Brute Force k-NN". This is computionally very expensive. So, there are other ways as well to perfrom k-NN which are comparatively less expensive than Brute force approach. The idea behind using other algorithms for k-NN classifier is to reduce the time during test period by preprocessing the training data in such a way that the test data can be easily classified in the appropriate clusters.

Let's discuss and understand the two most famous algorithms:

k-Dimensional Tree (kd tree)

k-d tree is a hierarchical binary tree. When this algorithm is used for k-NN classficaition, it rearranges the whole dataset in a binary tree structure, so that when test data is provided, it would give out the result by traversing through the tree, which takes less time than brute search.

The dataset is divided like a tree as shown in the above figure. Say we have 3 dimensional data i.e. (x,y,z) then the tree is formed with root node being one of the dimensions, here we start with 'x'. Then on the next level the split is done on basis of the second dimension, 'y' in our case. Similarly, third level with 3rd dimension and so on. And in case of 'k' dimensions, each split is made on basis of 'k' dimensions. Let's understand how k-d trees are formed with an example:

Training Data ⇒ {(1,2), (2,3), (2,4), (3,6), (4,2), (5,7), (6,8), (7,5), (8,5), (9,1), (9,3)}

here, K = 2

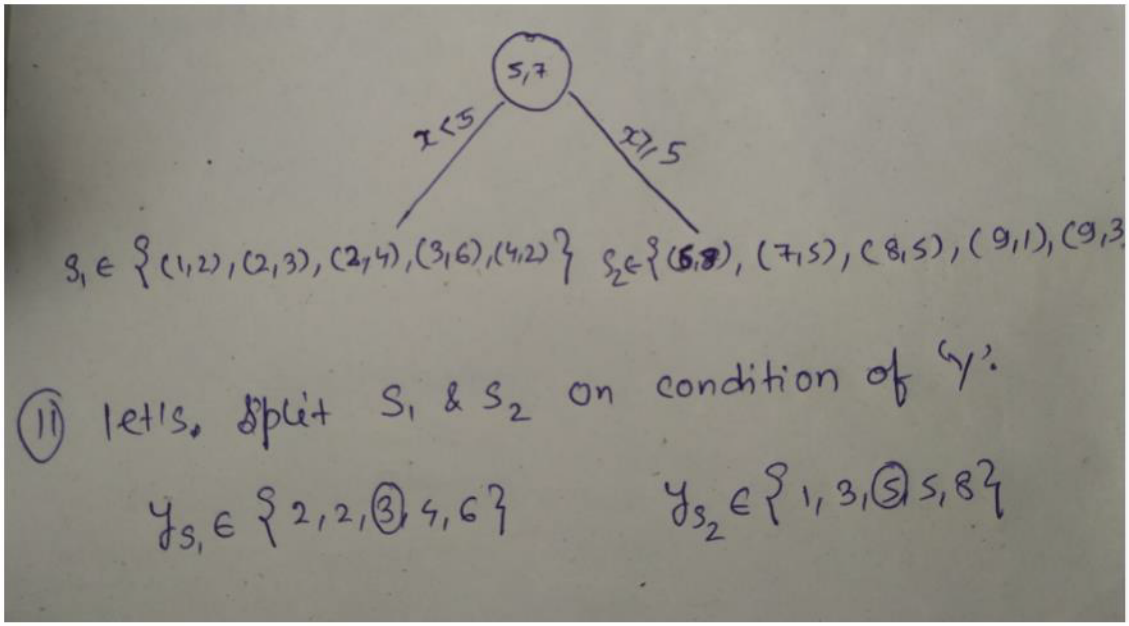
let's build our 2-d tree

let's sort our data and choose the median to be the split point :-

$x \in \{1, 2, 2, 3, 4, (5), 6, 7, 8, 9, 9\}$

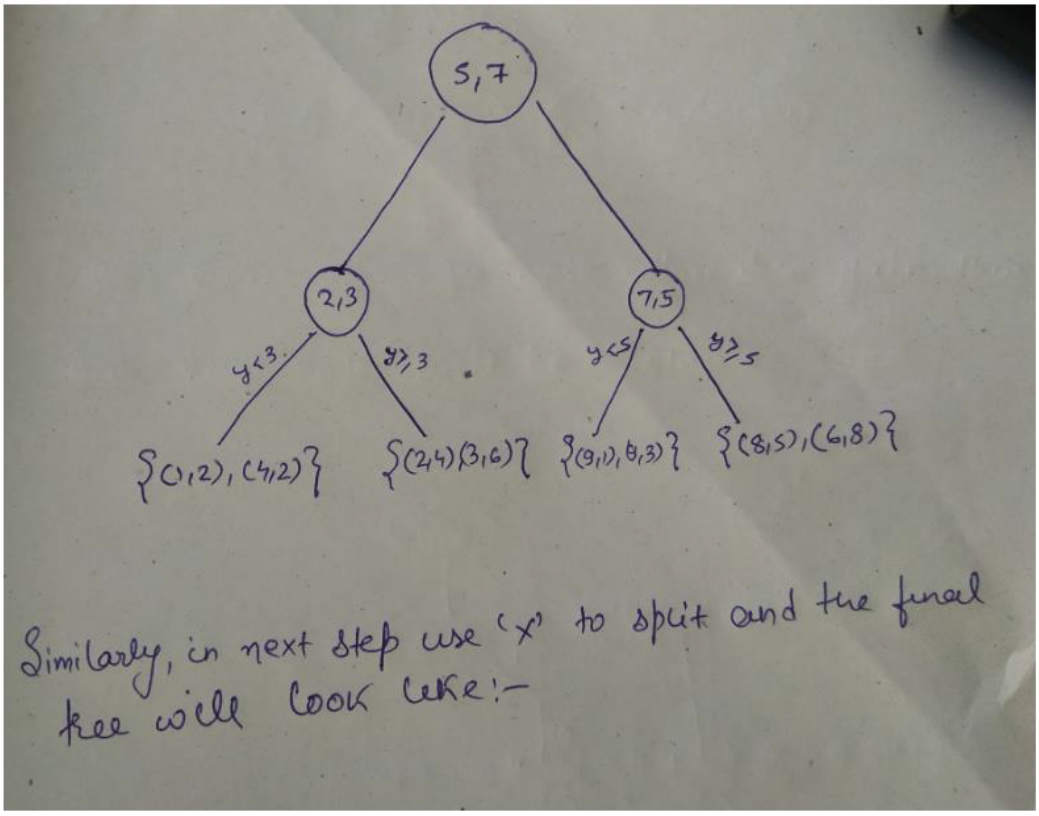
↳ median

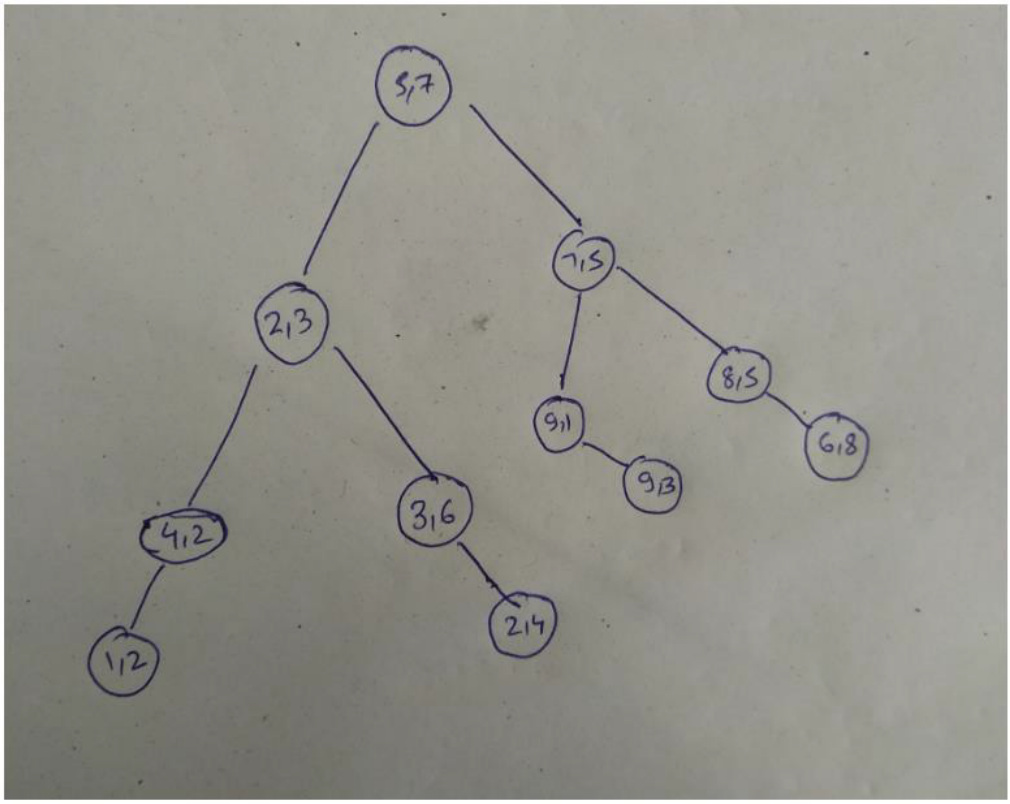
(i) our first node will be, (5,7), $x > 5$ → split condition



(ii) let's split S_1 & S_2 on condition of 'y'.

$y_{S_1} \in \{2, 2, (3), 4, 6\}$ $y_{S_2} \in \{1, 3, (5), 5, 8\}$





Once the tree is formed , it is easy for algorithm to search for the probable nearest neighbor just by traversing the tree. The main problem k-d trees is that it gives probable nearest neighbors but can miss out actual nearest neighbors.

In [1]:

```
# import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
#import plotly

import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
# Get the CSV data here and print head

df=pd.read_csv('https://raw.githubusercontent.com/training-ml/Files/main/breast%20cancer.cs
df.head()
```

Out[2]:

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mea
842302	M	17.99	10.38	122.80	1001.0	0.118
842517	M	20.57	17.77	132.90	1326.0	0.084
84300903	M	19.69	21.25	130.00	1203.0	0.109
84348301	M	11.42	20.38	77.58	386.1	0.142
84358402	M	20.29	14.34	135.10	1297.0	0.100

5 rows × 32 columns

In [3]:

```
# print summary
print('shape  ------------->',df.shape)
print('Each column and data type and its count','\n')
print(df.info())
```

```
shape  -------------> (569, 32)
Each column and data type and its count

<class 'pandas.core.frame.DataFrame'>
Int64Index: 569 entries, 842302 to 92751
Data columns (total 32 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   diagnosis                569 non-null    object 
 1   radius_mean              569 non-null    float64
 2   texture_mean             569 non-null    float64
 3   perimeter_mean           569 non-null    float64
 4   area_mean                569 non-null    float64
 5   smoothness_mean          569 non-null    float64
 6   compactness_mean         569 non-null    float64
 7   concavity_mean           569 non-null    float64
 8   concave points_mean      569 non-null    float64
 9   symmetry_mean            569 non-null    float64
 10  fractal_dimension_mean   569 non-null    float64
 11  radius_se                569 non-null    float64
 12  texture_se               569 non-null    float64
 13  perimeter_se             569 non-null    float64
 14  area_se                  569 non-null    float64
 15  smoothness_se            569 non-null    float64
 16  compactness_se           569 non-null    float64
 17  concavity_se             569 non-null    float64
 18  concave points_se        569 non-null    float64
 19  symmetry_se              569 non-null    float64
 20  fractal_dimension_se     569 non-null    float64
 21  radius_worst             569 non-null    float64
 22  texture_worst            569 non-null    float64
 23  perimeter_worst          569 non-null    float64
 24  area_worst               569 non-null    float64
 25  smoothness_worst         569 non-null    float64
 26  compactness_worst        569 non-null    float64
 27  concavity_worst          569 non-null    float64
 28  concave points_worst     569 non-null    float64
 29  symmetry_worst           569 non-null    float64
 30  fractal_dimension_worst  569 non-null    float64
 31  Unnamed: 32              0 non-null      float64
dtypes: float64(31), object(1)
memory usage: 146.7+ KB
None
```

In [4]:

```
# DROP ALERT 1 : Unnmaed : 32 column has all nulls. safe to remove the column.

df=df.drop(['Unnamed: 32'], axis=1)
```

In [5]:

```
# Dataframe statistics

df.describe()
```

Out[5]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactne
count	569.000000	569.000000	569.000000	569.000000	569.000000	5(
mean	14.127292	19.289649	91.969033	654.889104	0.096360	
std	3.524049	4.301036	24.298981	351.914129	0.014064	
min	6.981000	9.710000	43.790000	143.500000	0.052630	
25%	11.700000	16.170000	75.170000	420.300000	0.086370	
50%	13.370000	18.840000	86.240000	551.100000	0.095870	
75%	15.780000	21.800000	104.100000	782.700000	0.105300	
max	28.110000	39.280000	188.500000	2501.000000	0.163400	

8 rows × 30 columns

In [6]:

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

Out[6]:

```
diagnosis                 0
radius_mean               0
texture_mean              0
perimeter_mean            0
area_mean                 0
smoothness_mean           0
compactness_mean          0
concavity_mean            0
concave points_mean       0
symmetry_mean             0
fractal_dimension_mean    0
radius_se                 0
texture_se                0
perimeter_se              0
area_se                   0
smoothness_se             0
compactness_se            0
concavity_se              0
concave points_se         0
symmetry_se               0
fractal_dimension_se      0
radius_worst              0
texture_worst             0
perimeter_worst           0
area_worst                0
smoothness_worst          0
compactness_worst         0
concavity_worst           0
concave points_worst      0
symmetry_worst            0
fractal_dimension_worst   0
dtype: int64
```

seems no other cols have nulls. it's safe to proceed.

In [7]:

```
df.diagnosis.value_counts()
```

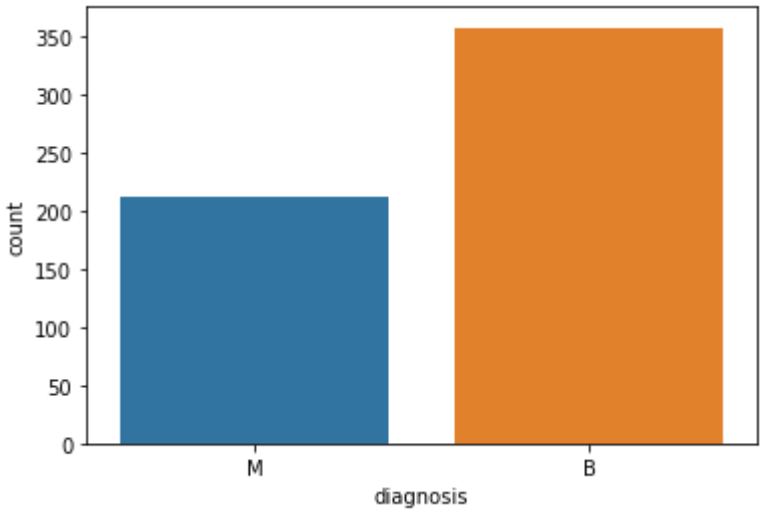
Out[7]:

```
B    357
M    212
Name: diagnosis, dtype: int64
```

In [8]:

```
# plot each class frequency

sns.countplot(x='diagnosis', data =df)
plt.show()
```



In [9]:

```
df.shape
```

Out[9]:

```
(569, 31)
```

Using SelectKBest feature Selection method

selectKBest use f_classif function to find best features, where f_classif uses ANOVA test.

In [10]:

```
from sklearn.feature_selection import SelectKBest, f_classif
```

In [11]:

```
#Replace Label column (diagnosis) into binary codes
df['diagnosis']=df['diagnosis'].replace({'M':1,'B':0})
```

In [12]:

```
x= df.drop('diagnosis',axis=1)
y=df.diagnosis
```

In [13]:

```
best_features=SelectKBest(score_func=f_classif, k=17)

fit= best_features.fit(x,y)
df_scores=pd.DataFrame(fit.scores_)
df_columns=pd.DataFrame(x.columns)

# concatenate dataframes

feature_scores=pd.concat([df_columns,df_scores], axis=1)
feature_scores.columns=['Feature_Name','Score']# name output columns
print(feature_scores.nlargest(17,'Score'))  # print 17 best features


# export selected features to .csv for later use
# df_backup= feature_scores.nlargest(17,'Score')
# df_backup.to_csv('Selected_features.csv', index=False)
```

```
            Feature_Name       Score
27  concave points_worst  964.385393
22       perimeter_worst  897.944219
7     concave points_mean  861.676020
20          radius_worst  860.781707
2         perimeter_mean  697.235272
23            area_worst  661.600206
0            radius_mean  646.981021
3              area_mean  573.060747
6         concavity_mean  533.793126
26       concavity_worst  436.691939
5       compactness_mean  313.233079
25     compactness_worst  304.341063
10             radius_se  268.840327
12          perimeter_se  253.897392
13               area_se  243.651586
21         texture_worst  149.596905
24      smoothness_worst  122.472880
```

Model Building

In [14]:

```
new_x=df[['concave points_worst','perimeter_worst','concave points_mean','radius_worst','pe
```

In [15]:

```
new_x
```

Out[15]:

id	concave points_worst	perimeter_worst	concave points_mean	radius_worst	perimeter_mean	area_wor
842302	0.2654	184.60	0.14710	25.380	122.80	2019
842517	0.1860	158.80	0.07017	24.990	132.90	1956
84300903	0.2430	152.50	0.12790	23.570	130.00	1709
84348301	0.2575	98.87	0.10520	14.910	77.58	567
84358402	0.1625	152.20	0.10430	22.540	135.10	1575
...	...	...	...	...	...	
926424	0.2216	166.10	0.13890	25.450	142.00	2027
926682	0.1628	155.00	0.09791	23.690	131.20	1731
926954	0.1418	126.70	0.05302	18.980	108.30	1124
927241	0.2650	184.60	0.15200	25.740	140.10	1821
92751	0.0000	59.16	0.00000	9.456	47.92	268

569 rows × 17 columns

In [16]:

```
scalar=StandardScaler()
X_scalar=scalar.fit_transform(new_x)
```

In [17]:

```
from time import time

# Building model to test unexposed data
x_train,x_test,y_train,y_test=train_test_split(X_scalar,y,test_size=0.25, random_state=355)
knn=KNeighborsClassifier()

# checking training and testing time (Lazy Lrarner)

start=time()
knn.fit(x_train,y_train)
print("knn training Time   :" ,(time()  -  start))

start=time()
y_pred = knn.predict(x_test)
print("knn test time  :  ",(time() - start))
```

```
knn training Time   : 0.015578269958496094
knn test time  :   0.24486851692199707
```

In [18]:

```
cfm=confusion_matrix(y_test,y_pred)
cfm
```

Out[18]:

```
array([[91,  2],
       [ 4, 46]], dtype=int64)
```

In [19]:

```
print(classification_report(y_test,y_pred,digits=2))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        93
           1       0.96      0.92      0.94        50

    accuracy                           0.96       143
   macro avg       0.96      0.95      0.95       143
weighted avg       0.96      0.96      0.96       143
```

Cross validation

Kfold method (for demo)

In [20]:

```
from sklearn.model_selection import KFold, cross_val_score

k_f= KFold(n_splits=3,shuffle= True)

k_f
```

Out[20]:

```
KFold(n_splits=3, random_state=None, shuffle=True)
```

In [21]:

```
for train, test in k_f.split([1,2,3,4,5,6,7,8,9,10]):
    print('train :', train, 'test  :',test)
```

```
train : [0 1 3 5 8 9] test  : [2 4 6 7]
train : [0 2 4 5 6 7 9] test  : [1 3 8]
train : [1 2 3 4 6 7 8] test  : [0 5 9]
```

Cross validation score to check if the model is overlifting

In [22]:

```
cross_val_score(knn, X_scalar,y,cv=10)
```

Out[22]:

```
array([0.98245614, 0.94736842, 0.94736842, 0.98245614, 0.96491228,
       1.        , 0.96491228, 1.        , 0.96491228, 0.96428571])
```

In [23]:

```
cross_val_score(KNeighborsClassifier(),X_scalar,y,cv=5).mean()
```

Out[23]:

```
0.9701133364384411
```

Hyperparameter Tuning

Let's use GridSearchCV for the best parameter to improve the accuracy

In [24]:

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

In [25]:

```
param_grid={ 'algorithm' :  ['kd_tree','brute'],
             'leaf_size':  [3,5,6,7,8],
              'n_neighbors' :[3,5,7,9,11,13]}
```

In [26]:

```
gridsearch=GridSearchCV(estimator=knn,param_grid=param_grid)
```

In [27]:

```
gridsearch.fit(x_train,y_train)
```

Out[27]:

```
GridSearchCV(estimator=KNeighborsClassifier(),
             param_grid={'algorithm': ['kd_tree', 'brute'],
                         'leaf_size': [3, 5, 6, 7, 8],
                         'n_neighbors': [3, 5, 7, 9, 11, 13]})
```

In [28]:

```
gridsearch.best_params_
```

Out[28]:

```
{'algorithm': 'kd_tree', 'leaf_size': 3, 'n_neighbors': 3}
```

In [32]:

```
# we will use the best parameters in our k-NN algorithm and check if accuracy is increasing
knn=KNeighborsClassifier(algorithm='brute',leaf_size=3,n_neighbors=3)
```

In [33]:

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

Out[33]:

```
KNeighborsClassifier(algorithm='brute', leaf_size=3, n_neighbors=3)
```

In [34]:

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

In [35]:

```
cfm=confusion_matrix(y_test,y_pred)
cfm
```

Out[35]:

```
array([[91,  2],
       [ 4, 46]], dtype=int64)
```

In [36]:

```
print(classification_report(y_test,y_pred,digits=2))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        93
           1       0.96      0.92      0.94        50

    accuracy                           0.96       143
   macro avg       0.96      0.95      0.95       143
weighted avg       0.96      0.96      0.96       143
```

In [ ]:

```

```