# Breast Cancer Classification - kNN Classifier Algorithm in Python

Hello friends,

kNN or k-Nearest Neighbours Classifier is a very simple and easy to understand machine learning algorithm. In this kernel, I build a k Nearest Neighbours classifier to classify the patients suffering from Breast Cancer.

## Table of Contents

## 1. Introduction to k Nearest Neighbours algorithm

Table of Contents

In machine learning, k Nearest Neighbours or kNN is the simplest of all machine learning algorithms. It is a non-parametric algorithm used for classification and regression tasks. Non-parametric means there is no assumption required for data distribution. So, kNN does not require any underlying assumption to be made. In both classification and regression tasks, the input consists of the k closest training examples in the feature space. The output depends upon whether kNN is used for classification or regression purposes.

- In kNN classification, the output is a class membership. The given data point is classified based on the majority of type of its neighbours. The data point is assigned to the most frequent class among its k nearest neighbours. Usually k is a small positive integer. If k=1, then the data point is simply assigned to the class of that single nearest neighbour.
- In kNN regression, the output is simply some property value for the object. This value is the average of the values of k nearest neighbours.

kNN is a type of instance-based learning or lazy learning. Lazy learning means it does not require any training data points for model generation. All training data will be used in the testing phase. This makes training faster and testing slower and costlier. So, the testing phase requires more time and memory resources.

In kNN, the neighbours are taken from a set of objects for which the class or the object property value is known. This can be thought of as the training set for the kNN algorithm, though no explicit training step is required. In both classification and regression kNN algorithm, we can assign weight to the contributions of the neighbours. So, nearest neighbours contribute more to the average than the more distant ones.

## 2. k Nearest Neighbours intuition

Table of Contents

The kNN algorithm intuition is very simple to understand. It simply calculates the distance between a sample data point and all the other training data points. The distance can be Euclidean distance or Manhattan distance. Then, it selects the k nearest data points where k can be any integer. Finally, it assigns the sample data point to the class to which the majority of the k data points belong.

Now, we will see kNN algorithm in action. Suppose, we have a dataset with two variables which are classified as `Red` and `Blue`.

In kNN algorithm, k is the number of nearest neighbours. Generally, k is an odd number because it helps to decide the majority of the class. When k=1, then the algorithm is known as the nearest neighbour algorithm.

Now, we want to classify a new data point `X` into `Blue` class or `Red` class. Suppose the value of k is 3. The kNN algorithm starts by calculating the distance between `X` and all the other data points. It then finds the 3 nearest points with least distance to point `X`.

In the final step of the kNN algorithm, we assign the new data point `X` to the majority of the class of the 3 nearest points. If 2 of the 3 nearest points belong to the class `Red` while 1 belong to the class `Blue`, then we classify the new data point as `Red`.

## 3. How to decide the number of neighbours in kNN

Table of Contents

While building the kNN classifier model, one question that come to my mind is what should be the value of nearest neighbours (k) that yields highest accuracy. This is a very important question because the classification accuracy depends upon our choice of k.

The number of neighbours (k) in kNN is a parameter that we need to select at the time of model building. Selecting the optimal value of k in kNN is the most critical problem. A small value of k means that noise will have higher influence on the result. So, probability of overfitting is very high. A large value of k makes it computationally expensive in terms of time to build the kNN model. Also, a large value of k will have a smoother decision boundary which means lower variance but higher bias.

The data scientists choose an odd value of k if the number of classes is even. We can apply the elbow method to select the value of k. To optimize the results, we can use Cross Validation technique. Using the cross-validation technique, we can test the kNN algorithm with different values of k. The model which gives good accuracy can be considered to be an optimal choice. It depends on individual cases and at times best process is to run through each possible value of k and test our result.

## 4. Eager learners vs lazy learners

Table of Contents

Eager learners mean when giving training data points, we will construct a generalized model before performing prediction on given new points to classify. We can think of such learners as being ready, active and eager to classify new data points.

Lazy learning means there is no need for learning or training of the model and all of the data points are used at the time of prediction. Lazy learners wait until the last minute before classifying any data point. They merely store the training dataset and waits until classification needs to perform. 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.

Unlike eager learners, lazy learners do less work in the training phase and more work in the testing phase to make a classification.

## 5. Import libraries

Table of Contents

In [1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # for data visualization purposes
import seaborn as sns # for data visualization
%matplotlib inline
```

In [2]:
```
import warnings

warnings.filterwarnings('ignore')
```

## 6. Import dataset

Table of Contents

In [5]:
```
df= pd.read_csv(r'C:\Users\91939\Desktop\AI&DS\Data science projects\Project33_k
df.head()
```

Out[5]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1000025 | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 2 |
| **1** | 1002945 | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 2 |
| **2** | 1015425 | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 2 |
| **3** | 1016277 | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 2 |
| **4** | 1017023 | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 |

# 7. Exploratory data analysis

Table of Contents

Now, I will explore the data to gain insights about the data.

In [6]:
```
df.shape
```

Out[6]:
```
(699, 11)
```

We can see that there are 699 instances and 11 attributes in the data set.

In the dataset description, it is given that there are 10 attributes and 1 `Class` which is the target variable. So, we have 10 attributes and 1 target variable.

## Rename column names

We can see that the dataset does not have proper column names. The columns are merely labelled as 0,1,2.... and so on. We should give proper names to the columns. I will do it as follows:-

In [7]:
```
col_names = ['Id', 'Clump_thickness', 'Uniformity_Cell_Size', 'Uniformity_Cell_S
             'Single_Epithelial_Cell_Size', 'Bare_Nuclei', 'Bland_Chromatin', 'N

df.columns = col_names

df.columns
```

Out[7]:
```
Index(['Id', 'Clump_thickness', 'Uniformity_Cell_Size',
       'Uniformity_Cell_Shape', 'Marginal_Adhesion',
       'Single_Epithelial_Cell_Size', 'Bare_Nuclei', 'Bland_Chromatin',
       'Normal_Nucleoli', 'Mitoses', 'Class'],
      dtype='object')
```

We can see that the column names are renamed. Now, the columns have meaningful names.

In [8]:
```
df.head()
```

Out[8]:

| | Id | Clump_thickness | Uniformity_Cell_Size | Uniformity_Cell_Shape | Marginal_Adh |
|---|---|---|---|---|---|
| **0** | 1000025 | 5 | 1 | 1 | |
| **1** | 1002945 | 5 | 4 | 4 | |
| **2** | 1015425 | 3 | 1 | 1 | |
| **3** | 1016277 | 6 | 8 | 8 | |
| **4** | 1017023 | 4 | 1 | 1 | |

## Drop redundant columns

We should drop any redundant columns from the dataset which does not have any predictive power. Here, `Id` is the redundant column. So, I will drop it first.

In [24]:
```
# drop Id column from dataset

df.drop('Id', axis=1, inplace=True)
```

In [25]:
```
for col in df.columns:
    counts = df[col].value_counts()
    print(counts)
```

```
Clump_thickness
1     145
5     130
3     108
4      80
10     69
2      50
8      46
6      34
7      23
9      14
Name: count, dtype: int64
Uniformity_Cell_Size
1     384
10     67
3      52
2      45
4      40
5      30
8      29
6      27
7      19
9       6
Name: count, dtype: int64
Uniformity_Cell_Shape
1     353
2      59
10     58
3      56
4      44
5      34
6      30
7      30
8      28
9       7
Name: count, dtype: int64
Marginal_Adhesion
1     407
3      58
2      58
10     55
4      33
8      25
5      23
6      22
7      13
9       5
Name: count, dtype: int64
Single_Epithelial_Cell_Size
2     386
3      72
4      48
1      47
6      41
5      39
10     31
8      21
7      12
9       2
Name: count, dtype: int64
```

```
Bare_Nuclei
1     402
10    132
2      30
5      30
3      28
8      21
4      19
?      16
9       9
7       8
6       4
Name: count, dtype: int64
Bland_Chromatin
2     166
3     165
1     152
7      73
4      40
5      34
8      28
10     20
9      11
6      10
Name: count, dtype: int64
Normal_Nucleoli
1     443
10     61
3      44
2      36
8      24
6      22
5      19
4      18
7      16
9      16
Name: count, dtype: int64
Mitoses
1     579
2      35
3      33
10     14
4      12
7       9
8       8
5       6
6       3
Name: count, dtype: int64
Class
2    458
4    241
Name: count, dtype: int64
```

## View summary of dataset

In [26]:
```
# View summary of dataset

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 699 entries, 0 to 698
Data columns (total 10 columns):
 #   Column                       Non-Null Count  Dtype 
---  ------                       --------------  ----- 
 0   Clump_thickness              699 non-null    int64 
 1   Uniformity_Cell_Size         699 non-null    int64 
 2   Uniformity_Cell_Shape        699 non-null    int64 
 3   Marginal_Adhesion            699 non-null    int64 
 4   Single_Epithelial_Cell_Size  699 non-null    int64 
 5   Bare_Nuclei                  699 non-null    object
 6   Bland_Chromatin              699 non-null    int64 
 7   Normal_Nucleoli              699 non-null    int64 
 8   Mitoses                      699 non-null    int64 
 9   Class                        699 non-null    int64 
dtypes: int64(9), object(1)
memory usage: 54.7+ KB
```

The distribution of values shows that data type of `Bare_Nuclei` is of type integer. But the summary of the dataframe shows that it is type object. So, I will explicitly convert its data type to integer.

## Convert data type of Bare_Nuclei to integer

In [30]:
```python
df['Bare_Nuclei'] = pd.to_numeric(df['Bare_Nuclei'], errors='coerce')
```

## Check data types of columns of dataframe

In [31]:
```python
df.dtypes
```

Out[31]:
```
Clump_thickness                  int64
Uniformity_Cell_Size             int64
Uniformity_Cell_Shape            int64
Marginal_Adhesion                int64
Single_Epithelial_Cell_Size      int64
Bare_Nuclei                    float64
Bland_Chromatin                  int64
Normal_Nucleoli                  int64
Mitoses                          int64
Class                            int64
dtype: object
```

Now, we can see that all the columns of the dataframe are of type numeric.

## Summary of variables

- There are 10 numerical variables in the dataset.
- All of the variables are of discrete type.
- Out of all the 10 variables, the first 9 variables are feature variables and last variable `Class` is the target variable.

## Explore problems within variables

Now, I will explore problems within variables.

## Missing values in variables

In [33]:
```
df.isnull().sum()
```

Out[33]:
```
Clump_thickness                0
Uniformity_Cell_Size           0
Uniformity_Cell_Shape          0
Marginal_Adhesion              0
Single_Epithelial_Cell_Size    0
Bare_Nuclei                   16
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
Class                          0
dtype: int64
```

We can see that the `Bare_Nuclei` column contains missing values. We need to dig deeper to find the frequency distribution of values of `Bare_Nuclei`.

In [36]:
```
df['Bare_Nuclei'].value_counts()
```

Out[36]:
```
Bare_Nuclei
1.0     402
10.0    132
2.0      30
5.0      30
3.0      28
8.0      21
4.0      19
9.0       9
7.0       8
6.0       4
Name: count, dtype: int64
```

In [37]:
```
# check unique values in `Bare_Nuclei` column

df['Bare_Nuclei'].unique()
```

Out[37]:
```
array([ 1., 10.,  2.,  4.,  3.,  9.,  7., nan,  5.,  8.,  6.])
```

In [38]:
```
# check for nan values in `Bare_Nuclei` column

df['Bare_Nuclei'].isna().sum()
```

Out[38]:
```
16
```

We can see that there are 16 `nan` values in the dataset. I will impute missing values after dividing the dataset into training and test set.

## check frequency distribution of target variable `Class`

In [39]:
```
# view frequency distribution of values in `Class` variable
```

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

Out[39]:
```
Class
2    458
4    241
Name: count, dtype: int64
```

## check percentage of frequency distribution of `Class`

In [43]:
```
# view percentage of frequency distribution of values in `Class` variable

df['Class'].value_counts()/(len(df))*100
```

Out[43]:
```
Class
2    65.522175
4    34.477825
Name: count, dtype: float64
```

We can see that the `Class` variable contains 2 class labels - `2` and `4`. `2` stands for benign and `4` stands for malignant cancer.

## Outliers in numerical variables

In [46]:
```
# view summary statistics in numerical variables

print(round(df.describe(),2))
```

```
       Clump_thickness  Uniformity_Cell_Size  Uniformity_Cell_Shape  \
count           699.00                699.00                 699.00   
mean              4.42                  3.13                   3.21   
std               2.82                  3.05                   2.97   
min               1.00                  1.00                   1.00   
25%               2.00                  1.00                   1.00   
50%               4.00                  1.00                   1.00   
75%               6.00                  5.00                   5.00   
max              10.00                 10.00                  10.00   

       Marginal_Adhesion  Single_Epithelial_Cell_Size  Bare_Nuclei  \
count             699.00                       699.00       683.00   
mean                2.81                         3.22         3.54   
std                 2.86                         2.21         3.64   
min                 1.00                         1.00         1.00   
25%                 1.00                         2.00         1.00   
50%                 1.00                         2.00         1.00   
75%                 4.00                         4.00         6.00   
max                10.00                        10.00        10.00   

       Bland_Chromatin  Normal_Nucleoli  Mitoses   Class  
count           699.00           699.00   699.00  699.00  
mean              3.44             2.87     1.59    2.69  
std               2.44             3.05     1.72    0.95  
min               1.00             1.00     1.00    2.00  
25%               2.00             1.00     1.00    2.00  
50%               3.00             1.00     1.00    2.00  
75%               5.00             4.00     1.00    4.00  
max              10.00            10.00    10.00    4.00  
```

kNN algorithm is robust to outliers.

# 8. Data Visualization

Table of Contents

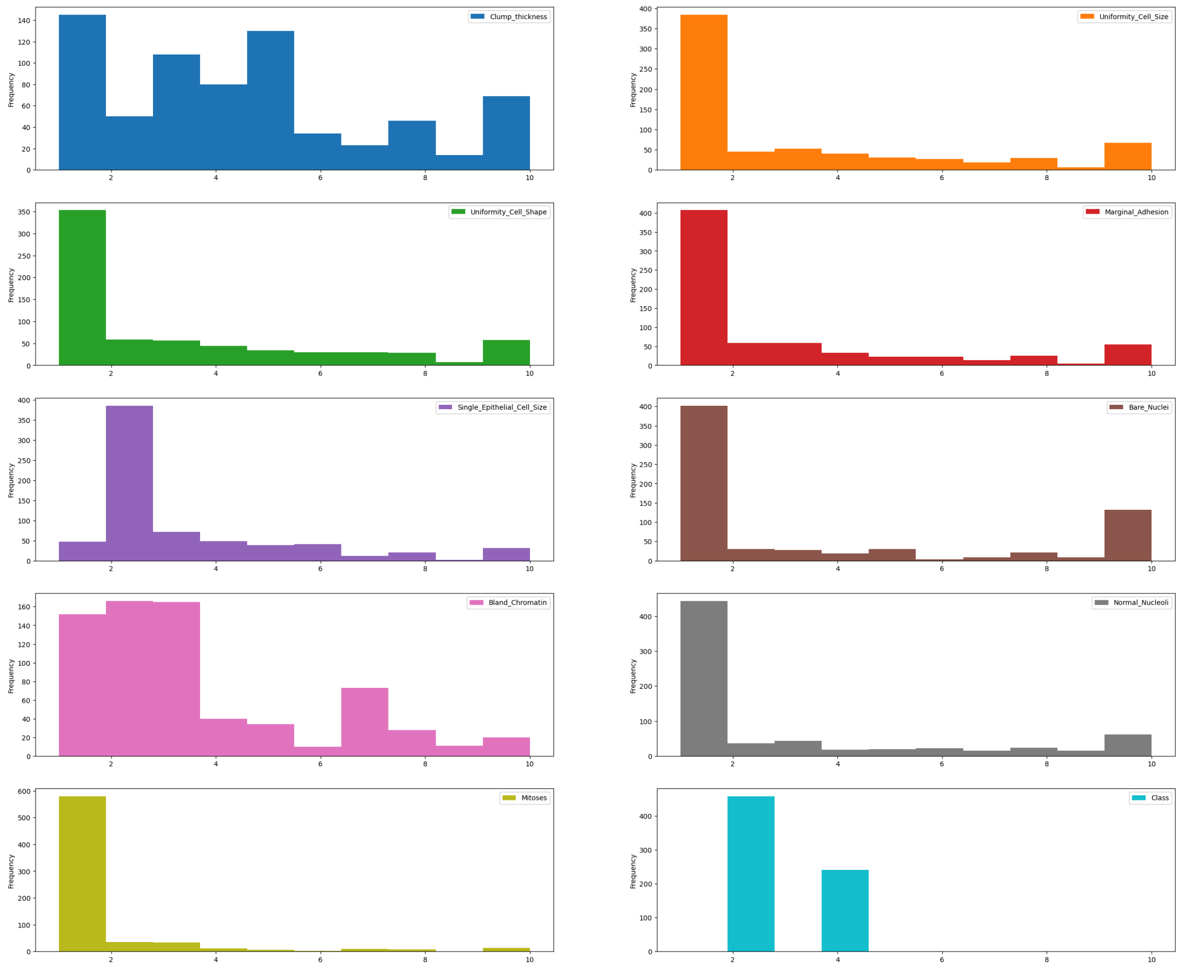
Now, we have a basic understanding of our data. I will supplement it with some data visualization to get better understanding of our data.

## Univariate plots

## Check the distribution of variables

Now, I will plot the histograms to check variable distributions to find out if they are normal or skewed.

```
In [48]: # plot histograms of the variables


plt.rcParams['figure.figsize']=(30,25)

df.plot(kind='hist', bins=10, subplots=True, layout=(5,2), sharex=False, sharey=

plt.show()
```



We can see that all the variables in the dataset are positively skewed.

## Multivariate plots

## Estimating correlation coefficients

Our dataset is very small. So, we can compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes. We can compute it using the `df.corr()` method as follows:-

```
In [49]: correlation = df.corr()
```

Our target variable is `Class`. So, we should check how each attribute correlates with the `Class` variable. We can do it as follows:-

```
In [50]: correlation['Class'].sort_values(ascending=False)
```

```
Out[50]: Class                          1.000000
         Bare_Nuclei                    0.822696
         Uniformity_Cell_Shape          0.818934
         Uniformity_Cell_Size           0.817904
         Bland_Chromatin                0.756616
         Clump_thickness                0.716001
         Normal_Nucleoli                0.712244
         Marginal_Adhesion              0.696800
         Single_Epithelial_Cell_Size    0.682785
         Mitoses                        0.423170
         Name: Class, dtype: float64
```

## Interpretation

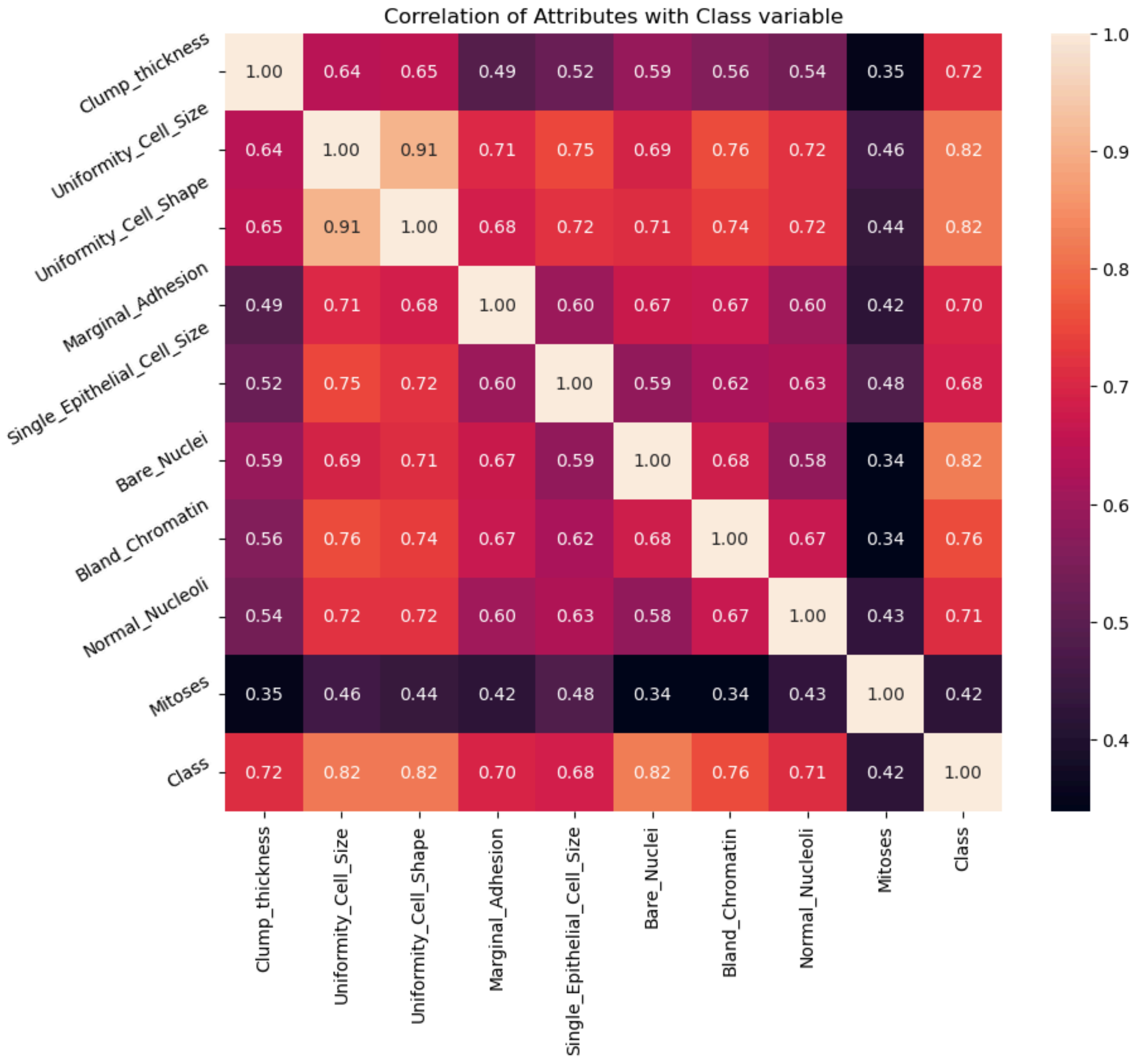
- The correlation coefficient ranges from -1 to +1.
- When it is close to +1, this signifies that there is a strong positive correlation. So, we can see that there is a strong positive correlation between `Class` and `Bare_Nuclei`, `Class` and `Uniformity_Cell_Shape`, `Class` and `Uniformity_Cell_Size`.
- When it is clsoe to -1, it means that there is a strong negative correlation. When it is close to 0, it means that there is no correlation.
- We can see that all the variables are positively correlated with `Class` variable. Some variables are strongly positive correlated while some variables are negatively correlated.

## Discover patterns and relationships

An important step in EDA is to discover patterns and relationships between variables in the dataset. I will use the seaborn heatmap to explore the patterns and relationships in the dataset.

## Correlation Heat Map

In [51]:
```
plt.figure(figsize=(10,8))
plt.title('Correlation of Attributes with Class variable')
a = sns.heatmap(correlation, square=True, annot=True, fmt='.2f', linecolor='whit
a.set_xticklabels(a.get_xticklabels(), rotation=90)
a.set_yticklabels(a.get_yticklabels(), rotation=30)
plt.show()
```



## Interpretation

From the above correlation heat map, we can conclude that :-

1. `Class` is highly positive correlated with `Uniformity_Cell_Size`, `Uniformity_Cell_Shape` and `Bare_Nuclei`. (correlation coefficient = 0.82).

2. `Class` is positively correlated with `Clump_thickness` (correlation coefficient=0.72), `Marginal_Adhesion` (correlation coefficient=0.70), `Single_Epithelial_Cell_Size)` (correlation coefficient = 0.68) and `Normal_Nucleoli` (correlation coefficient=0.71).

3. `Class` is weekly positive correlated with `Mitoses` (correlation coefficient=0.42).

4. The `Mitoses` variable is weekly positive correlated with all the other variables(correlation coefficient < 0.50).

## 9. Declare feature vector and target variable

Table of Contents

In [52]:
```
X = df.drop(['Class'], axis=1)

y = df['Class']
```

## 10. Split data into separate training and test set

Table of Contents

In [53]:
```
# split X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, randc
```

In [54]:
```
# check the shape of X_train and X_test

X_train.shape, X_test.shape
```

Out[54]:
```
((559, 9), (140, 9))
```

## 11. Feature Engineering

Table of Contents

**Feature Engineering** is the process of transforming raw data into useful features that help us to understand our model better and increase its predictive power. I will carry out feature engineering on different types of variables.

In [56]:
```
X_train.dtypes
```

Out[56]:
```
Clump_thickness                  int64
Uniformity_Cell_Size             int64
Uniformity_Cell_Shape            int64
Marginal_Adhesion                int64
Single_Epithelial_Cell_Size      int64
Bare_Nuclei                    float64
Bland_Chromatin                  int64
Normal_Nucleoli                  int64
Mitoses                          int64
dtype: object
```

## Engineering missing values in variables

In [57]:
```
# check missing values in numerical variables in X_train

X_train.isnull().sum()
```

Out[57]:
```
Clump_thickness                 0
Uniformity_Cell_Size            0
Uniformity_Cell_Shape           0
Marginal_Adhesion               0
Single_Epithelial_Cell_Size     0
Bare_Nuclei                    13
Bland_Chromatin                 0
Normal_Nucleoli                 0
Mitoses                         0
dtype: int64
```

In [58]:
```
# check missing values in numerical variables in X_test

X_test.isnull().sum()
```

Out[58]:
```
Clump_thickness                0
Uniformity_Cell_Size           0
Uniformity_Cell_Shape          0
Marginal_Adhesion              0
Single_Epithelial_Cell_Size    0
Bare_Nuclei                    3
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
dtype: int64
```

In [59]:
```
# print percentage of missing values in the numerical variables in training set

for col in X_train.columns:
    if X_train[col].isnull().mean()>0:
        print(col, round(X_train[col].isnull().mean(),4))
```

```
Bare_Nuclei 0.0233
```

## Assumption

I assume that the data are missing completely at random (MCAR). There are two methods which can be used to impute missing values. One is mean or median imputation and other one is random sample imputation. When there are outliers in the dataset, we should use median imputation. So, I will use median imputation because median imputation is robust to outliers.

I will impute missing values with the appropriate statistical measures of the data, in this case median. Imputation should be done over the training set, and then propagated to the test set. It means that the statistical measures to be used to fill missing values both in train and test set, should be extracted from the train set only. This is to avoid overfitting.

In [60]:
```
# impute missing values in X_train and X_test with respective column median in X

for df1 in [X_train, X_test]:
    for col in X_train.columns:
        col_median=X_train[col].median()
        df1[col].fillna(col_median, inplace=True)
```

In [61]:
```
# check again missing values in numerical variables in X_train

X_train.isnull().sum()
```

Out[61]:
```
Clump_thickness                0
Uniformity_Cell_Size           0
Uniformity_Cell_Shape          0
Marginal_Adhesion              0
Single_Epithelial_Cell_Size    0
Bare_Nuclei                    0
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
dtype: int64
```

In [62]:
```
# check missing values in numerical variables in X_test

X_test.isnull().sum()
```

Out[62]:
```
Clump_thickness                0
Uniformity_Cell_Size           0
Uniformity_Cell_Shape          0
Marginal_Adhesion              0
Single_Epithelial_Cell_Size    0
Bare_Nuclei                    0
Bland_Chromatin                0
Normal_Nucleoli                0
Mitoses                        0
dtype: int64
```

We can see that there are no missing values in X_train and X_test.

In [63]:
```
X_train.head()
```

Out[63]:

| | Clump_thickness | Uniformity_Cell_Size | Uniformity_Cell_Shape | Marginal_Adhesion |
|---|---|---|---|---|
| **293** | 10 | 4 | 4 | 6 |
| **62** | 9 | 10 | 10 | 1 |
| **485** | 1 | 1 | 1 | 3 |
| **422** | 4 | 3 | 3 | 1 |
| **332** | 5 | 2 | 2 | 2 |

In [64]:
```
X_test.head()
```

Out[64]:

| | Clump_thickness | Uniformity_Cell_Size | Uniformity_Cell_Shape | Marginal_Adhesion |
|---|---|---|---|---|
| **476** | 4 | 1 | 2 | 1 |
| **531** | 4 | 2 | 2 | 1 |
| **40** | 6 | 6 | 6 | 9 |
| **432** | 5 | 1 | 1 | 1 |
| **14** | 8 | 7 | 5 | 10 |

We now have training and testing set ready for model building. Before that, we should map all the feature variables onto the same scale. It is called `feature scaling`. I will do it as follows.

## 12. Feature Scaling

Table of Contents

In [65]:
```
cols = X_train.columns
```

In [66]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

In [67]:
```
X_train = pd.DataFrame(X_train, columns=[cols])
```

In [68]:
```
X_test = pd.DataFrame(X_test, columns=[cols])
```

In [69]:
```
X_train.head()
```

Out[69]:

| | Clump_thickness | Uniformity_Cell_Size | Uniformity_Cell_Shape | Marginal_Adhesion | Sin |
|---|---|---|---|---|---|
| **0** | 2.028383 | 0.299506 | 0.289573 | 1.119077 | |
| **1** | 1.669451 | 2.257680 | 2.304569 | -0.622471 | |
| **2** | -1.202005 | -0.679581 | -0.717925 | 0.074148 | |
| **3** | -0.125209 | -0.026856 | -0.046260 | -0.622471 | |
| **4** | 0.233723 | -0.353219 | -0.382092 | -0.274161 | |

We now have `X_train` dataset ready to be fed into the Logistic Regression classifier. I will do it as follows.

## 13. Fit K Neighbours Classifier to the training set

Table of Contents

In [70]:
```
# import KNeighbors ClaSSifier from sklearn
from sklearn.neighbors import KNeighborsClassifier


# instantiate the model
knn = KNeighborsClassifier(n_neighbors=3)


# fit the model to the training set
knn.fit(X_train, y_train)
```

Out[70]:
```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

## 14. Predict test-set results

Table of Contents

In [73]:
```
y_pred = knn.predict(X_test)

y_pred
```

Out[73]:
```
array([2, 2, 4, 2, 4, 2, 4, 2, 4, 2, 2, 2, 4, 4, 4, 2, 2, 4, 4, 2, 4, 4,
       2, 2, 2, 4, 2, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2,
       4, 4, 2, 4, 2, 4, 4, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 2, 4, 4, 4,
       4, 2, 2, 4, 2, 2, 4, 4, 2, 2, 2, 2, 4, 2, 2, 2, 4, 2, 2, 2, 4, 2,
       4, 4, 2, 2, 2, 4, 2, 2, 2, 4, 2, 4, 4, 2, 2, 2, 4, 2, 2, 2, 2, 2,
       4, 4, 4, 2, 2, 2, 2, 2, 4, 4, 4, 4, 2, 4, 2, 2, 4, 4, 4, 4, 4, 2,
       2, 4, 4, 2, 2, 4, 2, 2], dtype=int64)
```

## predict_proba method

**predict_proba** method gives the probabilities for the target variable(2 and 4) in this case, in array form.

`2 is for probability of benign cancer` and `4 is for probability of malignant cancer.`

In [74]:
```
# probability of getting output as 2 - benign cancer

knn.predict_proba(X_test)[:,0]
```

Out[74]:
```
array([1.        , 1.        , 0.33333333, 1.        , 0.        ,
       1.        , 0.        , 1.        , 0.        , 0.66666667,
       1.        , 1.        , 0.        , 0.33333333, 0.        ,
       1.        , 1.        , 0.        , 0.        , 1.        ,
       0.        , 0.        , 1.        , 1.        , 1.        ,
       0.        , 1.        , 1.        , 0.        , 0.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       0.66666667, 1.        , 0.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 0.        ,
       0.        , 1.        , 0.        , 1.        , 0.        ,
       0.        , 1.        , 1.        , 0.        , 1.        ,
       1.        , 1.        , 1.        , 0.66666667, 1.        ,
       0.        , 1.        , 1.        , 0.        , 0.        ,
       0.33333333, 0.        , 1.        , 1.        , 0.        ,
       1.        , 1.        , 0.        , 0.        , 1.        ,
       1.        , 1.        , 1.        , 0.        , 1.        ,
       1.        , 1.        , 0.        , 1.        , 1.        ,
       1.        , 0.        , 1.        , 0.        , 0.        ,
       1.        , 1.        , 0.66666667, 0.        , 1.        ,
       1.        , 1.        , 0.        , 1.        , 0.        ,
       0.        , 1.        , 1.        , 1.        , 0.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       0.        , 0.33333333, 0.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 0.        , 0.        ,
       0.        , 0.33333333, 1.        , 0.        , 1.        ,
       1.        , 0.33333333, 0.33333333, 0.        , 0.        ,
       0.        , 1.        , 1.        , 0.33333333, 0.        ,
       1.        , 1.        , 0.        , 1.        , 1.        ])
```

In [75]:
```
# probability of getting output as 4 - malignant cancer

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

```
Out[75]: array([0.        , 0.        , 0.66666667, 0.        , 1.        ,
                0.        , 1.        , 0.        , 1.        , 0.33333333,
                0.        , 0.        , 1.        , 0.66666667, 1.        ,
                0.        , 0.        , 1.        , 1.        , 0.        ,
                1.        , 1.        , 0.        , 0.        , 0.        ,
                1.        , 0.        , 0.        , 1.        , 1.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                0.33333333, 0.        , 1.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 1.        ,
                1.        , 0.        , 1.        , 0.        , 1.        ,
                1.        , 0.        , 0.        , 1.        , 0.        ,
                0.        , 0.        , 0.        , 0.33333333, 0.        ,
                1.        , 0.        , 0.        , 1.        , 1.        ,
                0.66666667, 1.        , 0.        , 0.        , 1.        ,
                0.        , 0.        , 1.        , 1.        , 0.        ,
                0.        , 0.        , 0.        , 1.        , 0.        ,
                0.        , 0.        , 1.        , 0.        , 0.        ,
                0.        , 1.        , 0.        , 1.        , 1.        ,
                0.        , 0.        , 0.33333333, 1.        , 0.        ,
                0.        , 0.        , 1.        , 0.        , 1.        ,
                1.        , 0.        , 0.        , 0.        , 1.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                1.        , 0.66666667, 1.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 1.        , 1.        ,
                1.        , 0.66666667, 0.        , 1.        , 0.        ,
                0.        , 0.66666667, 0.66666667, 1.        , 1.        ,
                1.        , 0.        , 0.        , 0.66666667, 1.        ,
                0.        , 0.        , 1.        , 0.        , 0.        ])
```

## 15. Check accuracy score

Table of Contents

```
In [77]: from sklearn.metrics import accuracy_score

         print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

```
Model accuracy score: 0.9714
```

Here, **y_test** are the true class labels and **y_pred** are the predicted class labels in the test-set.

### Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting.

```
In [78]: y_pred_train = knn.predict(X_train)
```

```
In [79]: print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_
```

```
Training-set accuracy score: 0.9821
```

### Check for overfitting and underfitting

In [80]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(knn.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(knn.score(X_test, y_test)))
```

```
Training set score: 0.9821
Test set score: 0.9714
```

The training-set accuracy score is 0.9821 while the test-set accuracy to be 0.9714. These two values are quite comparable. So, there is no question of overfitting.

### Compare model accuracy with null accuracy

So, the model accuracy is 0.9714. But, we cannot say that our model is very good based on the above accuracy. We must compare it with the **null accuracy**. Null accuracy is the accuracy that could be achieved by always predicting the most frequent class.

So, we should first check the class distribution in the test set.

In [81]:
```
# check class distribution in test set

y_test.value_counts()
```

Out[81]:
```
Class
2    85
4    55
Name: count, dtype: int64
```

We can see that the occurences of most frequent class is 85. So, we can calculate null accuracy by dividing 85 by total number of occurences.

In [82]:
```
# check null accuracy score

null_accuracy = (85/(85+55))

print('Null accuracy score: {0:0.4f}'. format(null_accuracy))
```

```
Null accuracy score: 0.6071
```

We can see that our model accuracy score is 0.9714 but null accuracy score is 0.6071. So, we can conclude that our K Nearest Neighbors model is doing a very good job in predicting the class labels.

# 16. Rebuild kNN Classification model using different values of k

Table of Contents

I have build the kNN classification model using k=3. Now, I will increase the value of k and see its effect on accuracy.

## Rebuild kNN Classification model using k=5

In [83]:
```
# instantiate the model with k=5
knn_5 = KNeighborsClassifier(n_neighbors=5)


# fit the model to the training set
knn_5.fit(X_train, y_train)


# predict on the test-set
y_pred_5 = knn_5.predict(X_test)


print('Model accuracy score with k=5 : {0:0.4f}'. format(accuracy_score(y_test,
```

```
Model accuracy score with k=5 : 0.9714
```

## Rebuild kNN Classification model using k=6

In [84]:
```
# instantiate the model with k=6
knn_6 = KNeighborsClassifier(n_neighbors=6)


# fit the model to the training set
knn_6.fit(X_train, y_train)


# predict on the test-set
y_pred_6 = knn_6.predict(X_test)


print('Model accuracy score with k=6 : {0:0.4f}'. format(accuracy_score(y_test,
```

```
Model accuracy score with k=6 : 0.9786
```

## Rebuild kNN Classification model using k=7

In [85]:
```
# instantiate the model with k=7
knn_7 = KNeighborsClassifier(n_neighbors=7)


# fit the model to the training set
knn_7.fit(X_train, y_train)


# predict on the test-set
y_pred_7 = knn_7.predict(X_test)


print('Model accuracy score with k=7 : {0:0.4f}'. format(accuracy_score(y_test,
```

```
Model accuracy score with k=7 : 0.9786
```

## Rebuild kNN Classification model using k=8

In [86]:
```
# instantiate the model with k=8
knn_8 = KNeighborsClassifier(n_neighbors=8)


# fit the model to the training set
knn_8.fit(X_train, y_train)


# predict on the test-set
y_pred_8 = knn_8.predict(X_test)


print('Model accuracy score with k=8 : {0:0.4f}'. format(accuracy_score(y_test,
```

```
Model accuracy score with k=8 : 0.9786
```

## Rebuild kNN Classification model using k=9

In [87]:
```
# instantiate the model with k=9
knn_9 = KNeighborsClassifier(n_neighbors=9)


# fit the model to the training set
knn_9.fit(X_train, y_train)


# predict on the test-set
y_pred_9 = knn_9.predict(X_test)


print('Model accuracy score with k=9 : {0:0.4f}'. format(accuracy_score(y_test,
```

```
Model accuracy score with k=9 : 0.9714
```

## Interpretation

Our original model accuracy score with k=3 is 0.9714. Now, we can see that we get same accuracy score of 0.9714 with k=5. But, if we increase the value of k further, this would result in enhanced accuracy.

With k=6,7,8 we get accuracy score of 0.9786. So, it results in performance improvement.

If we increase k to 9, then accuracy decreases again to 0.9714.

Now, based on the above analysis we can conclude that our classification model accuracy is very good. Our model is doing a very good job in terms of predicting the class labels.

But, it does not give the underlying distribution of values. Also, it does not tell anything about the type of errors our classifer is making.

We have another tool called `Confusion matrix` that comes to our rescue.

# 17. Confusion matrix

Table of Contents

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

**True Positives (TP)** – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

**True Negatives (TN)** – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

**False Positives (FP)** – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called **Type I error.**

**False Negatives (FN)** – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called **Type II error.**

These four outcomes are summarized in a confusion matrix given below.

In [88]:
```
# Print the Confusion Matrix with k =3 and slice it into four pieces

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

```
Confusion matrix

 [[83  2]
 [ 2 53]]

True Positives(TP) =  83

True Negatives(TN) =  53

False Positives(FP) =  2

False Negatives(FN) =  2
```

The confusion matrix shows `83 + 53 = 136 correct predictions` and `2 + 2 = 4 incorrect predictions`.

In this case, we have

- `True Positives` (Actual Positive:1 and Predict Positive:1) - 83
- `True Negatives` (Actual Negative:0 and Predict Negative:0) - 53
- `False Positives` (Actual Negative:0 but Predict Positive:1) - 2 `(Type I error)`
- `False Negatives` (Actual Positive:1 but Predict Negative:0) - 2 `(Type II error)`

In [89]:
```python
# Print the Confusion Matrix with k =7 and slice it into four pieces

cm_7 = confusion_matrix(y_test, y_pred_7)

print('Confusion matrix\n\n', cm_7)

print('\nTrue Positives(TP) = ', cm_7[0,0])

print('\nTrue Negatives(TN) = ', cm_7[1,1])

print('\nFalse Positives(FP) = ', cm_7[0,1])

print('\nFalse Negatives(FN) = ', cm_7[1,0])
```

```
Confusion matrix

 [[83  2]
 [ 1 54]]

True Positives(TP) =  83

True Negatives(TN) =  54

False Positives(FP) =  2

False Negatives(FN) =  1
```

The above confusion matrix shows `83 + 54 = 137 correct predictions` and `2 + 1 = 4 incorrect predictions`.

In this case, we have

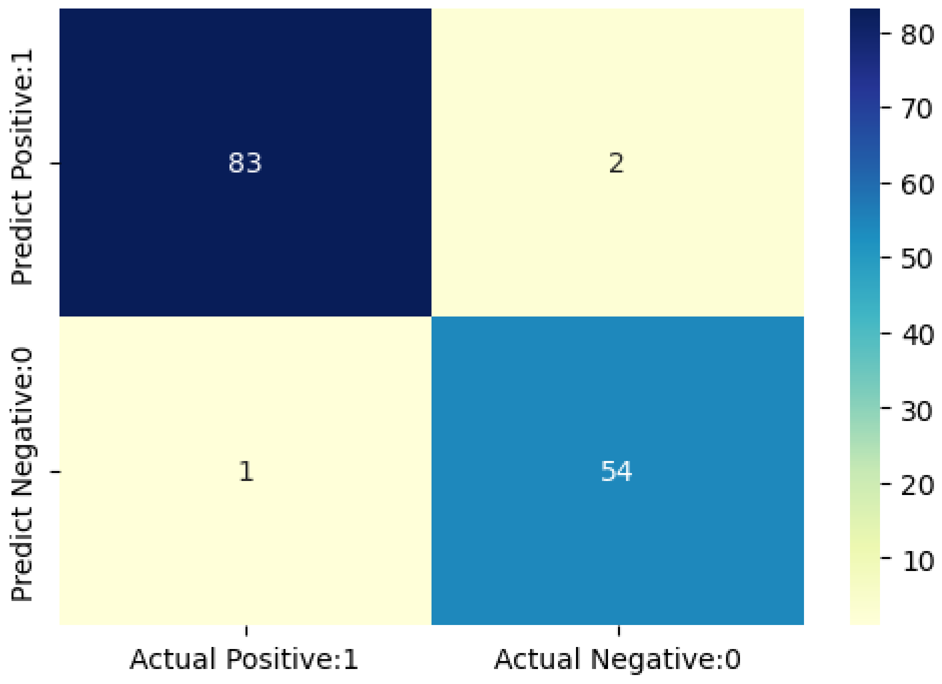
- `True Positives` (Actual Positive:1 and Predict Positive:1) - 83
- `True Negatives` (Actual Negative:0 and Predict Negative:0) - 54
- `False Positives` (Actual Negative:0 but Predict Positive:1) - 2 `(Type I error)`
- `False Negatives` (Actual Positive:1 but Predict Negative:0) - 1 `(Type II error)`

## Comment

So, kNN Classification model with k=7 shows more accurate predictions and less number of errors than k=3 model. Hence, we got performance improvement with k=7.

In [90]:
```
# visualize confusion matrix with seaborn heatmap

plt.figure(figsize=(6,4))

cm_matrix = pd.DataFrame(data=cm_7, columns=['Actual Positive:1', 'Actual Negati
                                 index=['Predict Positive:1', 'Predict Negative:

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
```

Out[90]:
```
<Axes: >
```



# 18. Classification metrices

Table of Contents

## Classification Report

**Classification report** is another way to evaluate the classification model performance. It displays the **precision**, **recall**, **f1** and **support** scores for the model. I have described these terms in later.

We can print a classification report as follows:-

In [92]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_7))
```

```
              precision    recall  f1-score   support

           2       0.99      0.98      0.98        85
           4       0.96      0.98      0.97        55

    accuracy                           0.98       140
   macro avg       0.98      0.98      0.98       140
weighted avg       0.98      0.98      0.98       140
```

## Classification accuracy

In [93]:
```
TP = cm_7[0,0]
TN = cm_7[1,1]
FP = cm_7[0,1]
FN = cm_7[1,0]
```

In [94]:
```
# print classification accuracy

classification_accuracy = (TP + TN) / float(TP + TN + FP + FN)

print('Classification accuracy : {0:0.4f}'.format(classification_accuracy))
```

```
Classification accuracy : 0.9786
```

## Classification error

In [95]:
```
# print classification error

classification_error = (FP + FN) / float(TP + TN + FP + FN)

print('Classification error : {0:0.4f}'.format(classification_error))
```

```
Classification error : 0.0214
```

## Precision

**Precision** can be defined as the percentage of correctly predicted positive outcomes out of all the predicted positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (TP + FP).

So, **Precision** identifies the proportion of correctly predicted positive outcome. It is more concerned with the positive class than the negative class.

Mathematically, `precision` can be defined as the ratio of `TP to (TP + FP)`.

In [96]:
```
# print precision score

precision = TP / float(TP + FP)


print('Precision : {0:0.4f}'.format(precision))
```

```
Precision : 0.9765
```

## Recall

Recall can be defined as the percentage of correctly predicted positive outcomes out of all the actual positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true positives and false negatives (TP + FN). **Recall** is also called **Sensitivity**.

**Recall** identifies the proportion of correctly predicted actual positives.

Mathematically, `recall` can be given as the ratio of `TP to (TP + FN)`.

In [97]:
```
recall = TP / float(TP + FN)

print('Recall or Sensitivity : {0:0.4f}'.format(recall))
```
```
Recall or Sensitivity : 0.9881
```

## True Positive Rate

**True Positive Rate** is synonymous with **Recall**.

In [98]:
```
true_positive_rate = TP / float(TP + FN)


print('True Positive Rate : {0:0.4f}'.format(true_positive_rate))
```
```
True Positive Rate : 0.9881
```

## False Positive Rate

In [99]:
```
false_positive_rate = FP / float(FP + TN)


print('False Positive Rate : {0:0.4f}'.format(false_positive_rate))
```
```
False Positive Rate : 0.0357
```

## Specificity

In [100…
```
specificity = TN / (TN + FP)

print('Specificity : {0:0.4f}'.format(specificity))
```
```
Specificity : 0.9643
```

## f1-score

**f1-score** is the weighted harmonic mean of precision and recall. The best possible **f1-score** would be 1.0 and the worst would be 0.0. **f1-score** is the harmonic mean of precision and recall. So, **f1-score** is always lower than accuracy measures as they embed precision and recall into their computation. The weighted average of `f1-score` should be used to compare classifier models, not global accuracy.

## Support

**Support** is the actual number of occurrences of the class in our dataset.

## Adjusting the classification threshold level

In [101…

```
# print the first 10 predicted probabilities of two classes- 2 and 4

y_pred_prob = knn.predict_proba(X_test)[0:10]

y_pred_prob
```

Out[101…

```
array([[1.        , 0.        ],
       [1.        , 0.        ],
       [0.33333333, 0.66666667],
       [1.        , 0.        ],
       [0.        , 1.        ],
       [1.        , 0.        ],
       [0.        , 1.        ],
       [1.        , 0.        ],
       [0.        , 1.        ],
       [0.66666667, 0.33333333]])
```

## Observations

- In each row, the numbers sum to 1.
- There are 2 columns which correspond to 2 classes - 2 and 4.
  - Class 2 - predicted probability that there is benign cancer.
  - Class 4 - predicted probability that there is malignant cancer.
- Importance of predicted probabilities
  - We can rank the observations by probability of benign or malignant cancer.
- predict_proba process
  - Predicts the probabilities
  - Choose the class with the highest probability
- Classification threshold level
  - There is a classification threshold level of 0.5.
  - Class 4 - probability of malignant cancer is predicted if probability > 0.5.
  - Class 2 - probability of benign cancer is predicted if probability < 0.5.

In [102…

```
# store the probabilities in dataframe

y_pred_prob_df = pd.DataFrame(data=y_pred_prob, columns=['Prob of - benign cance

y_pred_prob_df
```

Out[102…

| | Prob of - benign cancer (2) | Prob of - malignant cancer (4) |
|---|---|---|
| **0** | 1.000000 | 0.000000 |
| **1** | 1.000000 | 0.000000 |
| **2** | 0.333333 | 0.666667 |
| **3** | 1.000000 | 0.000000 |
| **4** | 0.000000 | 1.000000 |
| **5** | 1.000000 | 0.000000 |
| **6** | 0.000000 | 1.000000 |
| **7** | 1.000000 | 0.000000 |
| **8** | 0.000000 | 1.000000 |
| **9** | 0.666667 | 0.333333 |

In [103…

```
# print the first 10 predicted probabilities for class 4 - Probability of malign

knn.predict_proba(X_test)[0:10, 1]
```

Out[103…

```
array([0.        , 0.        , 0.66666667, 0.        , 1.        ,
       0.        , 1.        , 0.        , 1.        , 0.33333333])
```

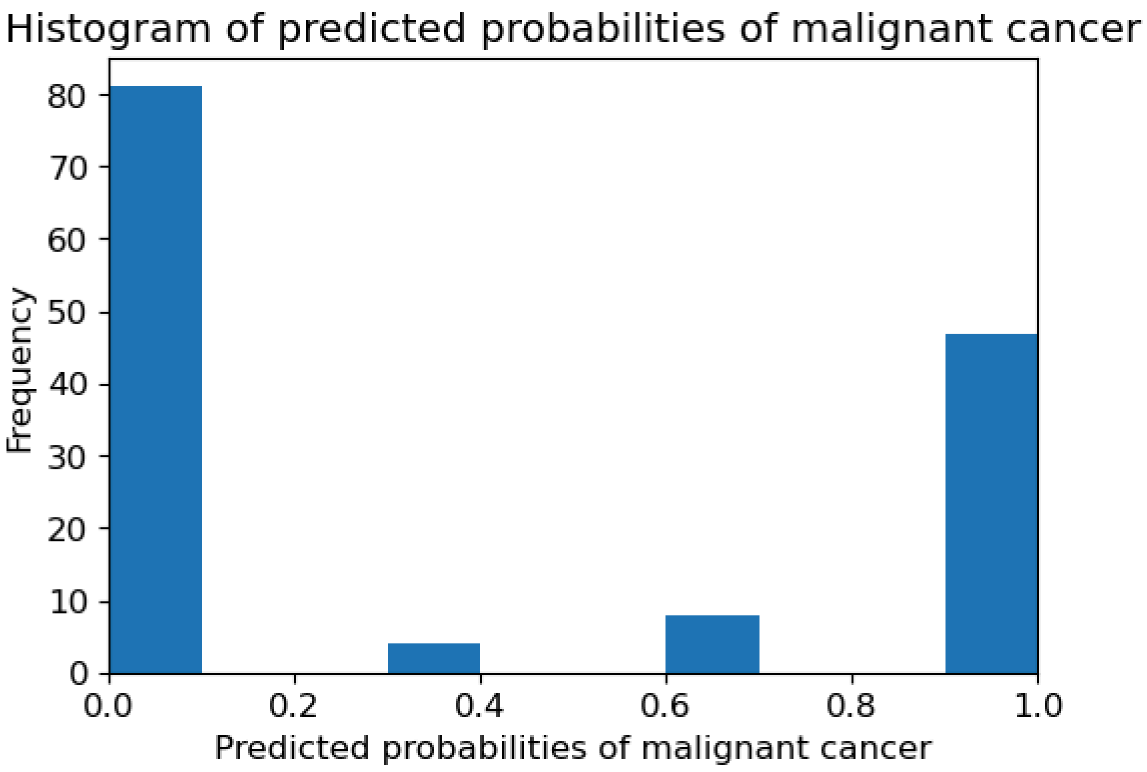
In [104…

```
# store the predicted probabilities for class 4 - Probability of malignant cance

y_pred_1 = knn.predict_proba(X_test)[:, 1]
```

In [105…

```
# plot histogram of predicted probabilities


# adjust figure size
plt.figure(figsize=(6,4))


# adjust the font size
plt.rcParams['font.size'] = 12


# plot histogram with 10 bins
plt.hist(y_pred_1, bins = 10)


# set the title of predicted probabilities
plt.title('Histogram of predicted probabilities of malignant cancer')


# set the x-axis limit
plt.xlim(0,1)


# set the title
plt.xlabel('Predicted probabilities of malignant cancer')
plt.ylabel('Frequency')
```

```
Out[105…   Text(0, 0.5, 'Frequency')
```



## Observations

- We can see that the above histogram is positively skewed.
- The first column tell us that there are approximately 80 observations with 0 probability of malignant cancer.
- There are few observations with probability > 0.5.
- So, these few observations predict that there will be malignant cancer.

## Comments

- In binary problems, the threshold of 0.5 is used by default to convert predicted probabilities into class predictions.
- Threshold can be adjusted to increase sensitivity or specificity.
- Sensitivity and specificity have an inverse relationship. Increasing one would always decrease the other and vice versa.
- Adjusting the threshold level should be one of the last step you do in the model-building process.

# 19. ROC-AUC

Table of Contents

## ROC Curve

Another tool to measure the classification model performance visually is **ROC Curve**. ROC Curve stands for **Receiver Operating Characteristic Curve**. An **ROC Curve** is a plot which shows the performance of a classification model at various classification threshold levels.

The **ROC Curve** plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** at various threshold levels.

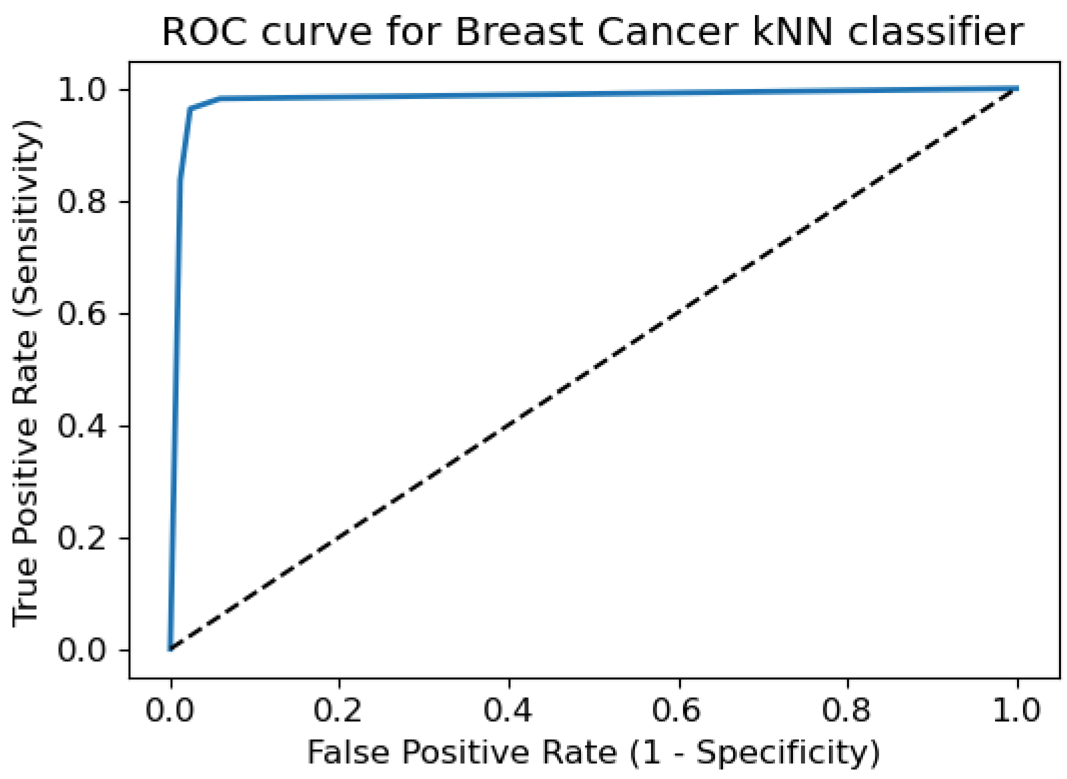
**True Positive Rate (TPR)** is also called **Recall**. It is defined as the ratio of **TP to (TP + FN)**.

**False Positive Rate (FPR)** is defined as the ratio of **FP to (FP + TN)**.

In the ROC Curve, we will focus on the TPR (True Positive Rate) and FPR (False Positive Rate) of a single point. This will give us the general performance of the ROC curve which consists of the TPR and FPR at various threshold levels. So, an ROC Curve plots TPR vs FPR at different classification threshold levels. If we lower the threshold levels, it may result in more items being classified as positve. It will increase both True Positives (TP) and False Positives (FP).

In [106…

```
# plot ROC Curve

from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_pred_1, pos_label=4)

plt.figure(figsize=(6,4))

plt.plot(fpr, tpr, linewidth=2)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12

plt.title('ROC curve for Breast Cancer kNN classifier')

plt.xlabel('False Positive Rate (1 - Specificity)')

plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```



## ROC AUC

**ROC AUC** stands for **Receiver Operating Characteristic - Area Under Curve**. It is a technique to compare classifier performance. In this technique, we measure the `area under the curve (AUC)`. A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

So, **ROC AUC** is the percentage of the ROC plot that is underneath the curve

In [107…

```
# compute ROC AUC

from sklearn.metrics import roc_auc_score

ROC_AUC = roc_auc_score(y_test, y_pred_1)

print('ROC AUC : {:.4f}'.format(ROC_AUC))
```

```
ROC AUC : 0.9825
```

## Interpretation

- ROC AUC is a single number summary of classifier performance. The higher the value, the better the classifier.
- ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a good job in predicting whether it is benign or malignant cancer.

In [108…

```
# calculate cross-validated ROC AUC

from sklearn.model_selection import cross_val_score
```

```
Cross_validated_ROC_AUC = cross_val_score(knn_7, X_train, y_train, cv=5, scoring

print('Cross validated ROC AUC : {:.4f}'.format(Cross_validated_ROC_AUC))
```

```
Cross validated ROC AUC : 0.9910
```

## Interpretation

Our Cross Validated ROC AUC is very close to 1. So, we can conclude that, the KNN classifier is indeed a very good model.

## 20. k-fold Cross Validation

Table of Contents

In this section, I will apply k-fold Cross Validation technique to improve the model performance. Cross-validation is a statistical method of evaluating generalization performance It is more stable and thorough than using a train-test split to evaluate model performance.

In [109…

```
# Applying 10-Fold Cross Validation

from sklearn.model_selection import cross_val_score

scores = cross_val_score(knn_7, X_train, y_train, cv = 10, scoring='accuracy')

print('Cross-validation scores:{}'.format(scores))
```

```
Cross-validation scores:[0.875      0.96428571 0.94642857 0.98214286 0.96428571
0.96428571
 0.98214286 0.98214286 1.         0.98181818]
```

We can summarize the cross-validation accuracy by calculating its mean.

In [110…

```
# compute Average cross-validation score

print('Average cross-validation score: {:.4f}'.format(scores.mean()))
```

```
Average cross-validation score: 0.9643
```

## Interpretation

- Using the mean cross-validation, we can conclude that we expect the model to be around 96.46 % accurate on average.
- If we look at all the 10 scores produced by the 10-fold cross-validation, we can also conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 87.72% accuracy. So, we can conclude that the model is very dependent on the particular folds used for training, but it also be the consequence of the small size of the dataset.
- We can see that 10-fold cross-validation accuracy does not result in performance improvement for this model.

# 21. Results and Conclusion

Table of Contents

1. In this project, I build a kNN classifier model to classify the patients suffering from breast cancer. The model yields very good performance as indicated by the model accuracy which was found to be 0.9786 with k=7.
2. With k=3, the training-set accuracy score is 0.9821 while the test-set accuracy to be 0.9714. These two values are quite comparable. So, there is no question of overfitting.
3. I have compared the model accuracy score which is 0.9714 with null accuracy score which is 0.6071. So, we can conclude that our K Nearest Neighbors model is doing a very good job in predicting the class labels.
4. Our original model accuracy score with k=3 is 0.9714. Now, we can see that we get same accuracy score of 0.9714 with k=5. But, if we increase the value of k further, this would result in enhanced accuracy. With k=6,7,8 we get accuracy score of 0.9786. So, it results in performance improvement. If we increase k to 9, then accuracy decreases again to 0.9714. So, we can conclude that our optimal value of k is 7.
5. kNN Classification model with k=7 shows more accurate predictions and less number of errors than k=3 model. Hence, we got performance improvement with k=7.
6. ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a good job in predicting whether it is benign or malignant cancer.
7. Using the mean cross-validation, we can conclude that we expect the model to be around 96.46 % accurate on average.
8. If we look at all the 10 scores produced by the 10-fold cross-validation, we can also conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 87.72% accuracy. So, we can conclude that the model is very dependent on the particular folds used for training, but it also be the consequence of the small size of the dataset.

```
In [ ]:
```