

1. Introduction to k Nearest Neighbours algorithm

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. Import libraries

```
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
```

3. Import dataset

```
data = '/content/Breast_cancer_Wilcoxon.csv'
```

```
df = pd.read_csv(data, header=None)
```

4. Exploratory data analysis

```
# view dimensions of dataset
```

```
df.shape
```

```
(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.

```
# preview the dataset
```

```
df.head()
```

```

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
```

Next steps:

[Generate code with df](#)
[View recommended plots](#)

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:-

```
col_names = ['Id', 'Clump_thickness', 'Uniformity_Cell_Size', 'Uniformity_Cell_Shape', 'Marginal_Adhesion',
             'Single_Epithelial_Cell_Size', 'Bare_Nuclei', 'Bland_Chromatin', 'Normal_Nucleoli', 'Mitoses', 'Class']
```

```
df.columns = col_names
```

```
df.columns
```

```
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

```
# let's again preview the dataset
```

```
df.head()
```

```

      Id  Clump_thickness  Uniformity_Cell_Size  Uniformity_Cell_Shape  Marginal_Adhesion  Single_Epithelial_Cell_Size  Bare_Nuclei
0  1000025              5              1              1              1              2
1  1002945              5              4              4              5              7
2  1015425              3              1              1              1              2
3  1016277              6              8              8              1              3
4  1017023              4              1              1              3              2

```

Next steps:

[Generate code with df](#)
[View recommended plots](#)

Drop redundant columns

```
# drop Id column from dataset
```

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

View summary of dataset

```
# 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

```

We can see that the Id column has been removed from the dataset.

We can see that there are 9 numerical variables and 1 categorical variable in the dataset. I will check the frequency distribution of values in the variables to confirm the same.

Frequency distribution of values in variables

```
for var in df.columns:
```

```

print(df[var].value_counts())
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

```

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

```
df['Bare_Nuclei'] = pd.to_numeric(df['Bare_Nuclei'], errors='coerce')
```

Check data types of columns of dataframe

```
df.dtypes
```

```

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

Missing values in variables

```
# check missing values in variables
```

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

```

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.

```
# check `na` values in the dataframe
```

```
df.isna().sum()
```

```

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 16 nan values.

```
# check frequency distribution of `Bare_Nuclei` column
```

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

```

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

```

```
# check unique values in `Bare_Nuclei` column
```

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

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

We can see that there are nan values in the Bare_Nuclei column.

```
# check for nan values in `Bare_Nuclei` column
```

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

```
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

```
# view frequency distribution of values in `Class` variable
```

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

```
Class
2    458
4    241
Name: count, dtype: int64
```

```
# view percentage of frequency distribution of values in `Class` variable
```

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

```
Class
2    0.655222
4    0.344778
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

```
# 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.

5. Data Visualization

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

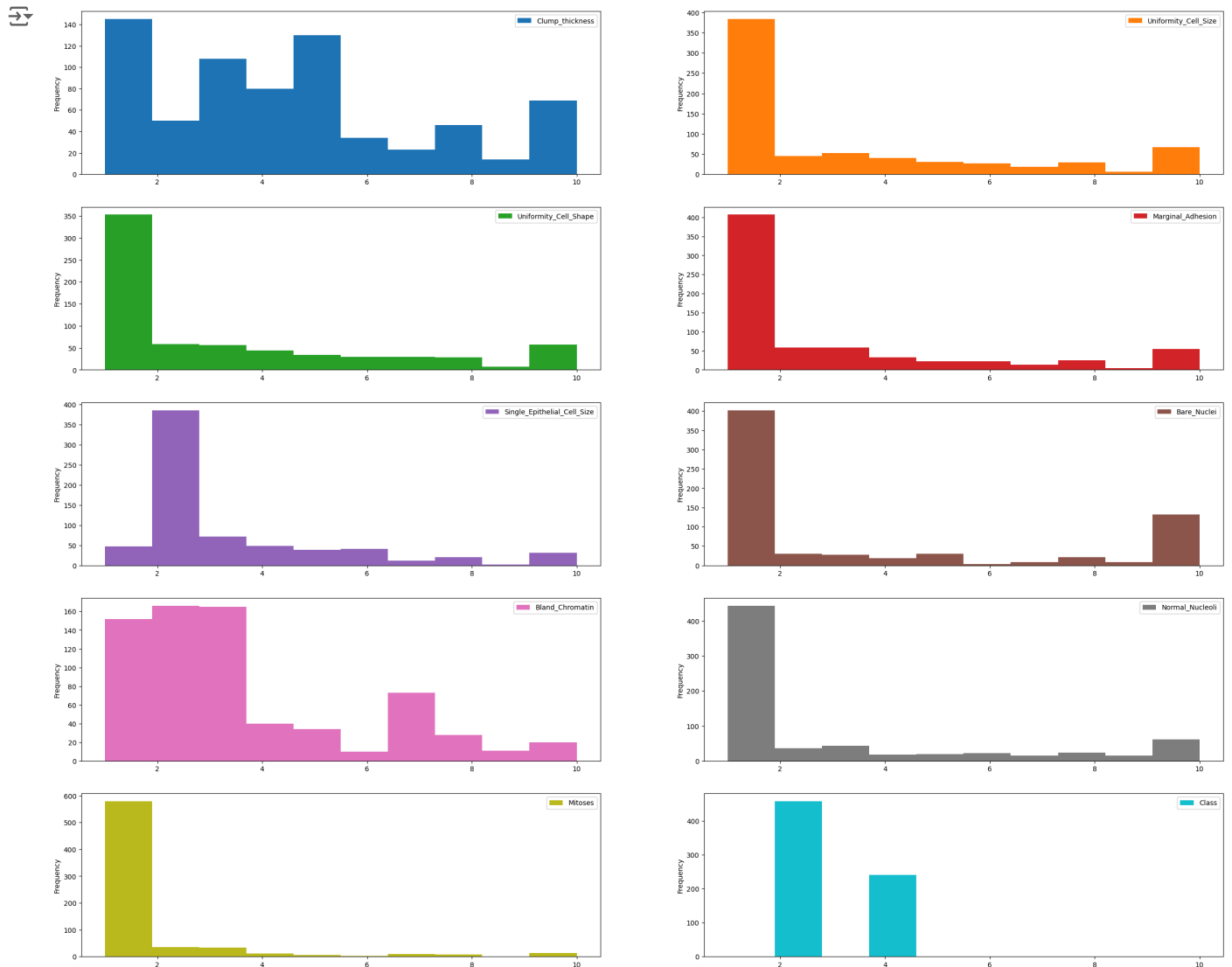
Check the distribution of variables Now, I will plot the histograms to check variable distributions to find out if they are normal or skewed

```
# 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=False)
```

```
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:-

```
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:-

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

```

↩ 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 close 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

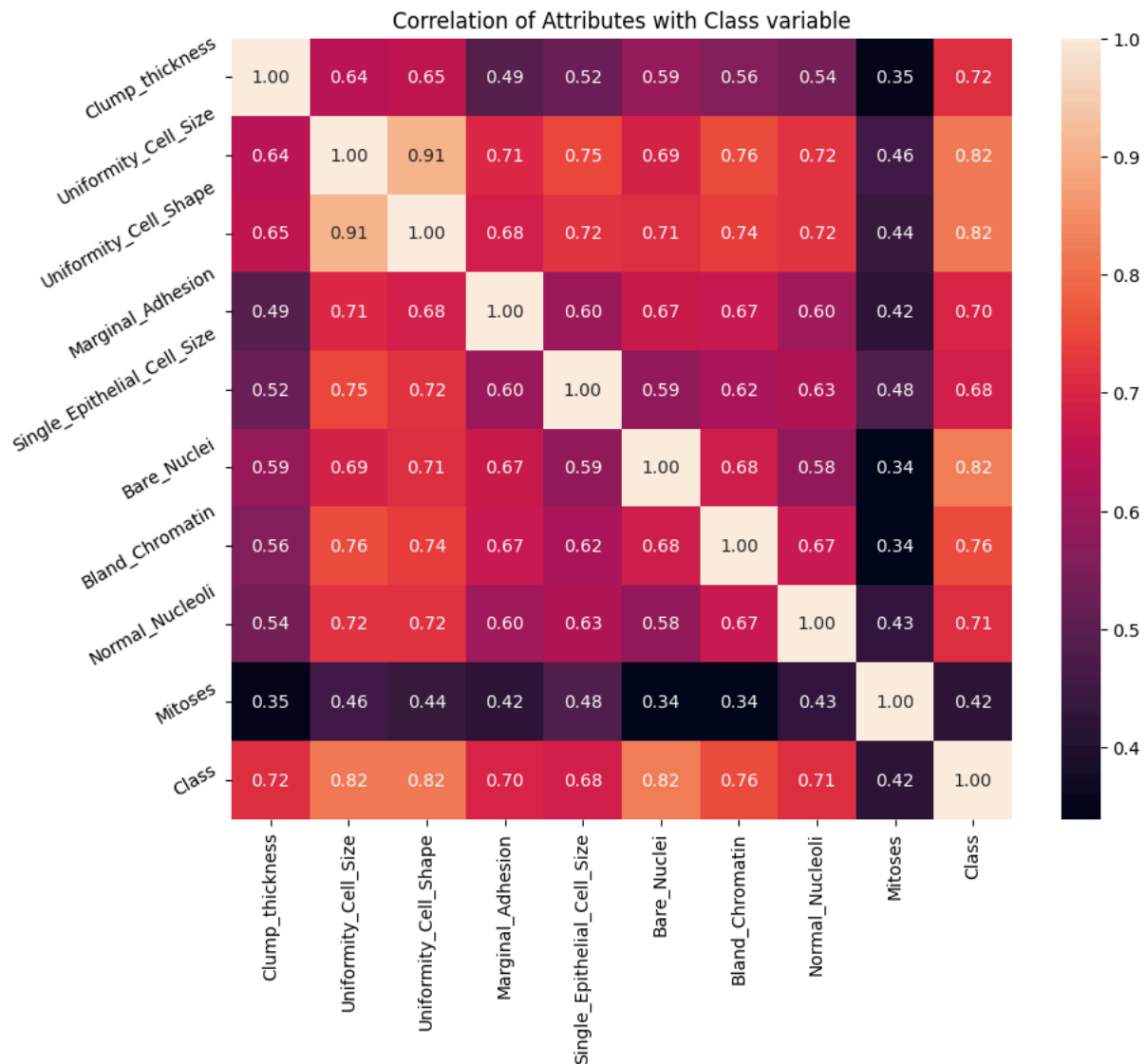
An important step in EDA is to discover patterns and relationships between variables in the dataset. I will use the seaborn heatmap to explore the patterns and relationships in the dataset.

Correlation Heat Map

```

plt.figure(figsize=(10,8))
plt.title('Correlation of Attributes with Class variable')
a = sns.heatmap(correlation, square=True, annot=True, fmt='.2f', linecolor='white')
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 :-

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

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).

Class is weakly positive correlated with Mitoses(correlation coefficient=0.42).

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

6. Declare feature vector and target variable

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

```
y = df['Class']
```

7. Split data into separate training and test set

```
# 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, random_state = 0)
```



```
# check the shape of X_train and X_test
```

```
X_train.shape, X_test.shape
```

```
((559, 9), (140, 9))
```

8. Feature Engineering

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.

```
# check data types in X_train
```

```
X_train.dtypes
```

```
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

```
# check missing values in numerical variables in X_train
```

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

```
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
```

```
# check missing values in numerical variables in X_test
```

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

```
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
```

```
# 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.

```
# impute missing values in X_train and X_test with respective column median in X_train
```

```
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)
```

```
# check again missing values in numerical variables in X_train
```

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

```
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
```

```
# check missing values in numerical variables in X_test
```

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

```
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.

```
X_train.head()
```

```
Clump_thickness  Uniformity_Cell_Size  Uniformity_Cell_Shape  Marginal_Adhesion  Single_Epithelial_Cell_Size  Bare_Nuclei  Blar
293             10                    4                    4             6                2      10.0
62              9                    10                   10             1               10       8.0
485             1                     1                     1             3                1       3.0
422             4                     3                     3             1                2       1.0
332             5                     2                     2             2                2       1.0
```

Next steps: [Generate code with X_train](#) [View recommended plots](#)

```
X_test.head()
```

```
Clump_thickness  Uniformity_Cell_Size  Uniformity_Cell_Shape  Marginal_Adhesion  Single_Epithelial_Cell_Size  Bare_Nuclei  Blar
476             4                     1                     2             1                2       1.0
531             4                     2                     2             1                2       1.0
40              6                     6                     6             9                6       1.0
432             5                     1                     1             1                2       1.0
14              8                     7                     5            10                7       9.0
```

Next steps: [Generate code with X_test](#) [View recommended plots](#)

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.

9. Feature Scaling

```
cols = X_train.columns
```



```

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.      ])

```

probability of getting output as 4 - malignant cancer

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

```

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.      ])

```

12. Check accuracy score

```
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.

```
y_pred_train = knn.predict(X_train)
```

```
print('Training-set accuracy score: {0:0.4f}'.format(accuracy_score(y_train, y_pred_train)))
```

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

Check for overfitting and underfitting

```
# 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.

```
# check class distribution in test set

y_test.value_counts()
```

↗ Class
2 85
4 55
Name: count, dtype: int64

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

```
# check null accuracy score

null_accuracy = (85/(85+55))

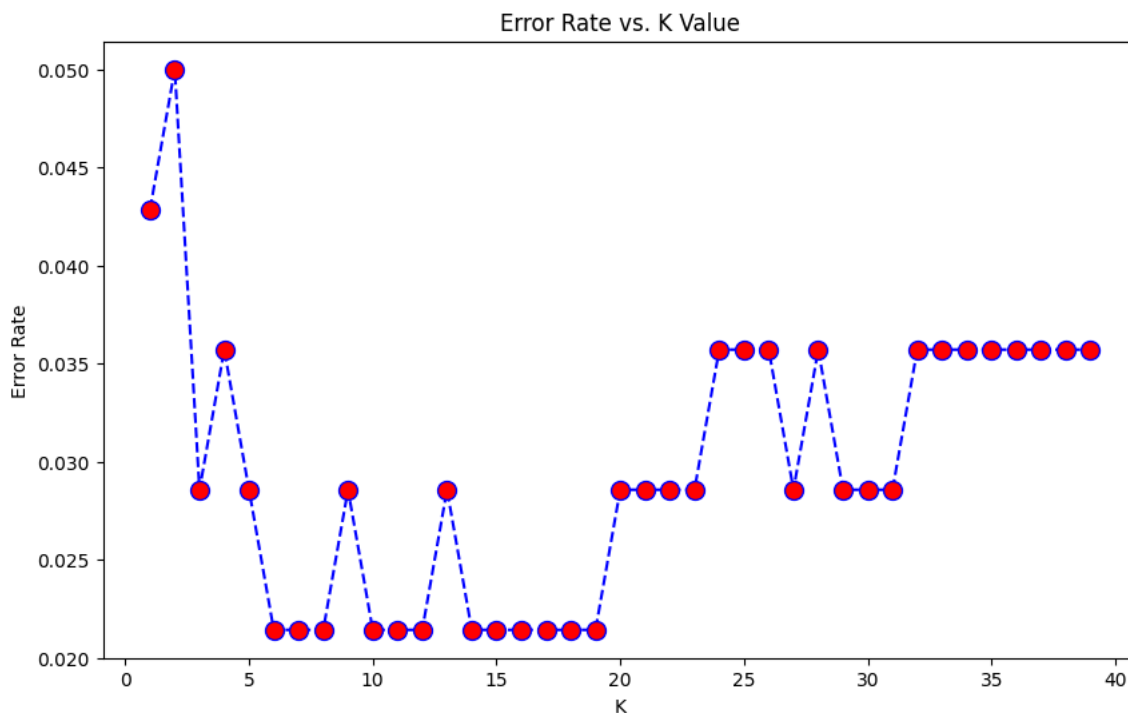
print('Null accuracy score: {0:.4f}'.format(null_accuracy))
```

↗ 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.

```
#Choosing a K Value
#Use the elbow method to pick a good K Value:
error_rate = []
# Will take some time
for i in range(1,40):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train,y_train)
    pred_i = knn.predict(X_test)
    error_rate.append(np.mean(pred_i != y_test))
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='blue', linestyle='dashed', marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

Text(0, 0.5, 'Error Rate')



We can see the graph above and can interpret that we can take values as k=5,6,7,8,9 and check the results.

13. Rebuild kNN Classification model using different values of k

Rebuild kNN Classification model using k=5

```
# 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, y_pred_5)))
```

Model accuracy score with k=5 : 0.9714

Rebuild kNN Classification model using k=6

```
# 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, y_pred_6)))
```

Model accuracy score with k=6 : 0.9786

Rebuild kNN Classification model using k=7

```
# 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, y_pred_7)))
```

↗ Model accuracy score with k=7 : 0.9786

Rebuild kNN Classification model using k=8

```
# 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, y_pred_8)))
```

↗ Model accuracy score with k=8 : 0.9786

Rebuild kNN Classification model using k=9

```
# 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, y_pred_9)))
```

↗ 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 classifier is making.

✓ 14. Confusion matrix

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.

```
# 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)

```
# 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 $2 + 1 = 3$ 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)

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.

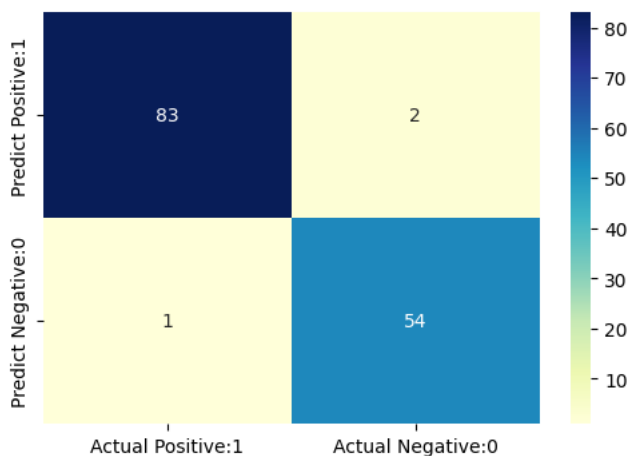
```
# visualize confusion matrix with seaborn heatmap
```

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

```
cm_matrix = pd.DataFrame(data=cm_7, columns=['Actual Positive:1', 'Actual Negative:0'],
                        index=['Predict Positive:1', 'Predict Negative:0'])
```

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

<Axes: >



15. Classification metrics

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.

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

Classification accuracy

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

```
# 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

```
# 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 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).

```
# print precision score

precision = TP / float(TP + FP)

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

→ Precision : 0.9765

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)

```
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

```
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

```
false_positive_rate = FP / float(FP + TN)

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

→ False Positive Rate : 0.0357

Specificity

```
specificity = TN / (TN + FP)

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

→ Specificity : 0.9643

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.

Adjusting the classification threshold level

```
# 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
```

```
array([[1.         , 0.         ],
       [1.         , 0.         ],
       [0.12820513, 0.87179487],
       [1.         , 0.         ],
       [0.02564103, 0.97435897],
       [1.         , 0.         ],
       [0.07692308, 0.92307692],
       [1.         , 0.         ],
       [0.05128205, 0.94871795],
       [0.53846154, 0.46153846]])
```

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.

```
# store the probabilities in dataframe
```

```
y_pred_prob_df = pd.DataFrame(data=y_pred_prob, columns=['Prob of - benign cancer (2)', 'Prob of - malignant cancer (4)'])
```

```
y_pred_prob_df
```

	Prob of - benign cancer (2)	Prob of - malignant cancer (4)
0	1.000000	0.000000
1	1.000000	0.000000
2	0.128205	0.871795
3	1.000000	0.000000
4	0.025641	0.974359
5	1.000000	0.000000
6	0.076923	0.923077
7	1.000000	0.000000
8	0.051282	0.948718
9	0.538462	0.461538

Next steps:

[Generate code with y_pred_prob_df](#)
[View recommended plots](#)

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

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

```
array([0.         , 0.         , 0.87179487, 0.         , 0.97435897,
       0.         , 0.92307692, 0.         , 0.94871795, 0.46153846])
```

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

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

```
# 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)

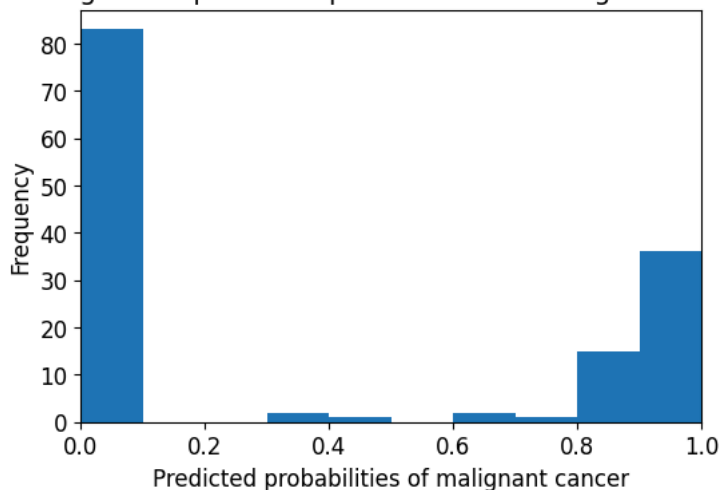
# set the title of predicted probabilities
plt.title('Histogram of predicted probabilities of malignant cancer')

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

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

↗ Text(0, 0.5, 'Frequency')

Histogram of predicted probabilities of malignant cancer



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.

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.

✓ 16. ROC-AUC

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 positive. It will increase both True Positives (TP) and False Positives (FP).

```
# 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--' )

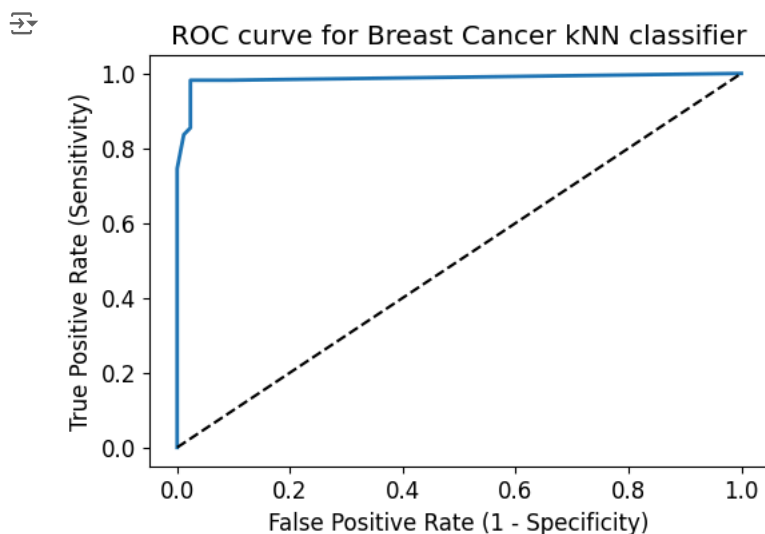
plt.rcParams['font.size'] = 12

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 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

```
# 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 : 0.9862
```

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.

```
# 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='roc_auc').mean()

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

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

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

✓ 17. k-fold Cross Validation

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.

```
# 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]
```

```
# compute Average cross-validation score
```

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

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