|  |  |
| --- | --- |
| Import Libraries | |
| In [1]: |  |
| import numpy as np import pandas as pd  import matplotlib.pyplot as plt import seaborn as sns  **%**matplotlib inline | |
| In [2]: |  |
| import warnings warnings.filterwarnings('ignore') | |
| In [85]: |  |
| *#display image using python*  from IPython.display import Image  url**=**'https://d2jx2rerrg6sh3.cloudfront.net/image-handler/picture/2017/2/shutterstock\_57606664 Image(url,height**=**300,width**=**400) | |
| Import Dataset | |

|  |  |
| --- | --- |
| In [3]: |  |
| dataset**=**pd.read\_csv(r"C:\Users\user\Downloads\29th\29th\projects\KNN\brest cancer.txt") dataset  **1000025 5 1 1.1 1.2 2 1.3 3 1.4 1.5 2.1**  **0** 1002945 5 4 4 5 7 10 3 2 1 2  **1** 1015425 3 1 1 1 2 2 3 1 1 2  **2** 1016277 6 8 8 1 3 4 3 7 1 2  **3** 1017023 4 1 1 3 2 1 3 1 1 2  **4** 1017122 8 10 10 8 7 10 9 7 1 4  **...** ... ... ... ... ... ... ... ... ... ... ...  **693** 776715 3 1 1 1 3 2 1 1 1 2  **694** 841769 2 1 1 1 2 1 1 1 1 2  **695** 888820 5 10 10 3 7 3 8 10 2 4  **696** 897471 4 8 6 4 3 4 10 6 1 4  **697** 897471 4 8 8 5 4 5 10 4 1 4  698 rows × 11 columns | |
| Exploratory Data Analysis | |
| Now, I will explore the the to gain insight about the data. | |
| In [4]: |  |
| *# view dimensions of dataset*  dataset.shape  (698, 11) | |
| We can see that there are 699 instances and 11 attributes in the dataset. | |
| in the dataset description, it is given that there are 10 attributes and 1class which is the target varia attribiutes and 1target variable. | |

|  |  |
| --- | --- |
| In [5]: |  |
| *#View top 5rows of this dataset*  dataset.head()  **1000025 5 1 1.1 1.2 2 1.3 3 1.4 1.5 2.1**  **0** 1002945 5 4 4 5 7 10 3 2 1 2  **1** 1015425 3 1 1 1 2 2 3 1 1 2  **2** 1016277 6 8 8 1 3 4 3 7 1 2  **3** 1017023 4 1 1 3 2 1 3 1 1 2  **4** 1017122 8 10 10 8 7 10 9 7 1 4 | |
| Rename column names | |
| We can see that the dataset doesn't have proper column names.The columns are mainly labelled a should give proper names to the columns. | |
| In [6]: |  |
| col\_names **=** ['Id', 'Clump\_thickness', 'Uniformity\_Cell\_Size', 'Uniformity\_Cell\_Shape', 'Margi 'Single\_Epithelial\_Cell\_Size', 'Bare\_Nuclei', 'Bland\_Chromatin', 'Normal\_Nucleol  dataset.columns **=** col\_names dataset.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 column names are renamed. Now the columns have meaningful names. | |
| In [7]: |  |
| *# let's agian preview the dataset*  dataset.head()  **Id Clump\_thickness Uniformity\_Cell\_Size Uniformity\_Cell\_Shape Marginal\_Adhesion Single\_Epithe**  **0** 1002945 5 4 4 5 7  **1** 1015425 3 1 1 1 2  **2** 1016277 6 8 8 1 3  **3** 1017023 4 1 1 3 2  **4** 1017122 8 10 10 8 7 | |

|  |  |
| --- | --- |
| Drop redundant columns | |
| We should drop any redundant columns from the dataset which doesn't have any predictive power. column. | |
| In [8]: |  |
| *# drop Id column from dataset*  dataset.drop('Id', axis**=**1, inplace**=**True) | |
| View summary of this dataset | |
| In [9]: |  |
| *# view summary of dataset*  dataset.info()  <class 'pandas.core.frame.DataFrame'> RangeIndex: 698 entries, 0 to 697 Data columns (total 10 columns):  # Column Non-Null Count Dtype     1. Clump\_thickness 698 non-null int64 2. Uniformity\_Cell\_Size 698 non-null int64 3. Uniformity\_Cell\_Shape 698 non-null int64 4. Marginal\_Adhesion 698 non-null int64 5. Single\_Epithelial\_Cell\_Size 698 non-null int64 6. Bare\_Nuclei 698 non-null object 7. Bland\_Chromatin 698 non-null int64 8. Normal\_Nucleoli 698 non-null int64 9. Mitoses 698 non-null int64 10. Class 698 non-null int64 dtypes: int64(9), object(1)   memory usage: 54.7+ KB | |
| We can see that the Id column has been removed from this dataset. | |
| Frequency distribution of values in variables | |

|  |  |
| --- | --- |
| In [10]: |  |
| for var in dataset.columns: print(dataset[var].value\_counts()) | |

|  |  |
| --- | --- |
| 1 | 145 |
| 5 | 129 |
| 3 | 108 |
| 4 | 80 |
| 10 | 69 |
| 2 | 50 |
| 8 | 46 |
| 6 | 34 |
| 7 | 23 |
| 9 | 14 |
| Name: | Clump\_thickness, dtype: int64 |
| 1 | 383 |
| 10 | 67 |
| 3 | 52 |
| 2 | 45 |
| 4 | 40 |
| 5 | 30 |
| 8 | 29 |
| 6 | 27 |
| 7 | 19 |
| 9 | 6 |
| Name: | Uniformity\_Cell\_Size, dtype: int64 |
| 1 | 352 |
| 2 | 59 |
| 10 | 58 |
| 3 | 56 |
| 4 | 44 |
| 5 | 34 |
| 6 | 30 |
| 7 | 30 |
| 8 | 28 |
| 9 | 7 |
| Name: | Uniformity\_Cell\_Shape, dtype: int64 |
| 1 | 406 |
| 3 | 58 |
| 2 | 58 |
| 10 | 55 |
| 4 | 33 |
| 8 | 25 |
| 5 | 23 |
| 6 | 22 |
| 7 | 13 |
| 9 | 5 |
| Name: | Marginal\_Adhesion, dtype: int64 |
| 2 | 385 |
| 3 | 72 |
| 4 | 48 |
| 1 | 47 |
| 6 | 41 |
| 5 | 39 |
| 10 | 31 |
| 8 | 21 |
| 7 | 12 |
| 9 | 2 |
| Name: | Single\_Epithelial\_Cell\_Size, dtype: int64 |
| 1 | 401 |
| 10 | 132 |
| 2 | 30 |
| 5 | 30 |
| 3 | 28 |
| 8 | 21 |
| 4 | 19 |
| ? | 16 |
| 9 | 9 |

|  |  |
| --- | --- |
| 7 8  6 4  Name: Bare\_Nuclei, dtype: int64 2 166  3 164  1 152  7 73  4 40  5 34  8 28  10 20  9 11  6 10  Name: Bland\_Chromatin, dtype: int64 1 442  10 61  3 44  2 36  8 24  6 22  5 19  4 18  7 16  9 16  Name: Normal\_Nucleoli, dtype: int64 1 578  2 35  3 33  10 14  4 12  7 9  8 8  5 6  6 3  Name: Mitoses, dtype: int64 2 457  4 241  Name: Class, dtype: int64 | |
| The distribution of values shows that data type of Bare\_Nuclei is of type integer but summary of the is type object. | |
| Convert data type of Bare\_Nuclei to integer | |
| In [11]: |  |
| dataset['Bare\_Nuclei'] **=** pd.to\_numeric(dataset['Bare\_Nuclei'], errors**=**'coerce') | |
| Check data types of colmuns of the dataframe | |

|  |  |
| --- | --- |
| In [12]: |  |
| dataset.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 numeric type. | |
| Explore problems within variables | |
| Now, i will explore problems within variables. | |
| Missing values in variables | |
| In [13]: |  |
| *# check missing values in variables*  dataset.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 cloumn contains missing values. we need to dig deeper to find the values. | |

|  |  |
| --- | --- |
| In [14]: |  |
| *# check `na` values in the dataframe*  dataset.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. | |
| In [15]: |  |
| *# check frequency distribution of `Bare\_Nuclei` column*  dataset['Bare\_Nuclei'].value\_counts()  1.0 401  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: Bare\_Nuclei, dtype: int64 | |
| In [16]: |  |
| *# check unique values in `Bare\_Nuclei` column*  dataset['Bare\_Nuclei'].unique()  array([10., 2., 4., 1., 3., 9., 7., nan, 5., 8., 6.]) | |
| We can see that there are nan values in the Bare\_Nuclei column. | |
| In [17]: |  |
| *# check for nan values in `Bare\_Nuclei` column*  dataset['Bare\_Nuclei'].isna().sum()  16 | |

|  |  |
| --- | --- |
| We can see that there are 16 nan vaules in the dataset. I will impute missing values after dividing th and testset. | |
| Check frequency distribution of target variable class | |
| In [18]: |  |
| *# view frequency distribution of values in `Class` variable*  dataset['Class'].value\_counts()  2 457  4 241  Name: Class, dtype: int64 | |
| Check percentage of frequency ditribution of class | |
| In [19]: |  |
| *# view percentage of frequency distribution of values in `Class` variable*  dataset['Class'].value\_counts()**/**np.float(len(dataset))  2 0.654728  4 0.345272  Name: Class, dtype: float64 | |
| Outliers in numerical variables | |

|  |  |
| --- | --- |
| In [22]: |  |
| *# view summary statistics in numerical variables*  print(round(dataset.describe(),2))  Clump\_thickness Uniformity\_Cell\_Size Uniformity\_Cell\_Shape \ count 698.00 698.00 698.00  mean 4.42 3.14 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 698.00 698.00 682.00  mean 2.81 3.22 3.55  std 2.86 2.22 3.65  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 698.00 698.00 698.00 698.00  mean 3.44 2.87 1.59 2.69  std 2.44 3.06 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 roboost outliers. | |
| Univariate plots | |
| Check the distribution of variables | |
| Now, i will plot histogram to check variable distributions to find out if they are normal or skewed. | |

|  |  |
| --- | --- |
| In [23]: |  |
| *# plot histograms of the variables*  plt.rcParams['figure.figsize']**=**(30,25)  dataset.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 between every pa compute it using the dataset.corr() method | |
| In [24]: |  |
| correlation**=**dataset.corr() | |

|  |  |
| --- | --- |
| Our target is class. So we should check how each attribute coorelates with the class variable. | |
| In [25]: |  |
| correlation['Class'].sort\_values(ascending**=**False)  Class 1.000000  Bare\_Nuclei 0.822563  Uniformity\_Cell\_Shape 0.818794  Uniformity\_Cell\_Size 0.817772  Bland\_Chromatin 0.756732  Clump\_thickness 0.716509  Normal\_Nucleoli 0.712067  Marginal\_Adhesion 0.696605  Single\_Epithelial\_Cell\_Size 0.682618  Mitoses 0.423008  Name: Class, dtype: float64 | |
| Discover patterns and relationships | |
| An important step in EDA is to discover patterns and relationships between variables in the dataset. heatmap to explore the partterns and relationships in the dataset. | |
| Correlation Heat Map | |

|  |  |
| --- | --- |
| In [26]: |  |
| 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() | |
| Declare feature vector and target variable | |

|  |  |
| --- | --- |
| In [28]: |  |
| X **=** dataset.drop(['Class'], axis**=**1) y **=** dataset['Class'] | |
| Split data into separate training and test set | |
| In [29]: |  |
| *# 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) | |
| In [30]: |  |
| *# check the shape of X\_train and X\_test*  X\_train.shape, X\_test.shape  ((558, 9), (140, 9)) | |
| Feature Engineering | |
| Feature Engineering\*\* is the process of transforming raw data into useful features that help us to un better and increase its predictive power. I will carry out feature engineering on different types of vari | |
| In [31]: |  |
| *# 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 | |

|  |  |
| --- | --- |
| In [32]: |  |
| *# 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 15  Bland\_Chromatin 0  Normal\_Nucleoli 0  Mitoses 0  dtype: int64 | |
| In [33]: |  |
| *# 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 1  Bland\_Chromatin 0  Normal\_Nucleoli 0  Mitoses 0  dtype: int64 | |
| In [34]: |  |
| *# 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 0.0269 | |
| Assumption  I assume that the data are missing completely at random (MCAR). There are two methods which ca missing values. One is mean or median imputation and other one is random sample imputation. Wh the dataset, we should use median imputation. So, I will use median imputation because median im outliers.  I will impute missing values with the appropriate statistical measures of the data, in this case media done over the training set, and then propagated to the test set. It means that the statistical measure missing values both in train and test set, should be extracted from the train set only. This is to avoid | |

|  |  |
| --- | --- |
| In [35]: |  |
| *# 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) | |
| In [36]: |  |
| *# 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 | |
| In [37]: |  |
| *# 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. | |

|  |  |
| --- | --- |
| In [38]: |  |
| X\_train.head()  **Clump\_thickness Uniformity\_Cell\_Size Uniformity\_Cell\_Shape Marginal\_Adhesion Single\_Epithelial\_Ce**  **62** 6 3 4 1 5  **193** 3 1 1 1 2  **263** 7 9 4 10 10  **222** 7 5 6 3 3  **140** 2 1 1 1 2  In [39]:  X\_test.head()  **Clump\_thickness Uniformity\_Cell\_Size Uniformity\_Cell\_Shape Marginal\_Adhesion Single\_Epithelial\_Ce**  **603** 5 3 2 8 5  **619** 3 1 1 1 2  **452** 4 5 5 8 6  **85** 3 3 6 4 5  **416** 1 1 1 1 2  We now have training and testing set ready for model building. Before that, we should map all the fe same scale. It is called 'feature scaling'. | |
| Feature Scaling | |
| In [40]: |  |
| cols **=** X\_train.columns | |
| In [41]: |  |
| from sklearn.preprocessing import StandardScaler scaler **=** StandardScaler()  X\_train **=** scaler.fit\_transform(X\_train)  X\_test **=** scaler.transform(X\_test) | |
| In [42]: |  |
| X\_train **=** pd.DataFrame(X\_train, columns**=**[cols]) | |
| In [43]: |  |
| X\_test **=** pd.DataFrame(X\_test, columns**=**[cols]) | |

|  |  |  |
| --- | --- | --- |
|  | In [44]: |  |
| X\_train.head()  **Clump\_thickness Uniformity\_Cell\_Size Uniformity\_Cell\_Shape Marginal\_Adhesion Single\_Epithelial\_Cell\_** | |
| **0** 0.574621 -0.040143 0.277515 -0.629622 0.775913 | | |
| **1** -0.497748 -0.680143 -0.721540 -0.629622 -0.549473 | | |
| **2** 0.932077 1.879857 0.277515 2.541854 2.984890 | | |
| **3** 0.932077 0.599857 0.943552 0.075150 -0.107678 | | |
| **4** -0.855205 -0.680143 -0.721540 -0.629622 -0.549473 | | |
| We now have 'X\_train' dataset ready to be fed into the Logistic Regression classifier. | | |
| Fit K Neighbours Classifier to the training eet | | |
| In [45]: | |  |
| *# 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)  ▾ KNeighborsClassifier  KNeighborsClassifier(n\_neigh bors=3) | | |
| Predict test-set results | | |
| In [46]: | |  |
| y\_pred **=** knn.predict(X\_test) | | |
| y\_pred | | |
| array([4, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 4, | | |
| 4, 4, 2, 4, 4, 4, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 4, | | |
| 4, 4, 2, 4, 2, 4, 2, 2, 2, 4, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 2, 4, | | |
| 4, 2, 4, 4, 2, 2, 4, 2, 2, 2, 4, 2, 4, 2, 4, 2, 2, 2, 2, 2, 4, 2, | | |
| 2, 4, 4, 4, 2, 4, 2, 4, 2, 2, 2, 2, 4, 4, 4, 4, 2, 2, 4, 2, 2, 2, | | |
| 2, 4, 2, 2, 2, 2, 4, 2, 2, 4, 2, 2, 4, 4, 4, 2, 2, 4, 2, 2, 4, 4, | | |
| 2, 4, 2, 2, 2, 2, 4, 4], dtype=int64) | | |
| Predict\_proba method | | |

|  |  |  |
| --- | --- | --- |
| **predict\_proba** method gives the probabilities for the target variable(2 and 4) in this case, in array f | | |
| In [47]: |  | |
| *# probability of getting output*  knn.predict\_proba(X\_test)[:,0] | | *as 2 - benign cancer* |
| array([0. , 1. , 0. | | , 0.33333333, 1. , |
| 1. , 1. , 1. | | , 1. , 1. , |
| 1. , 1. , 1. | | , 1. , 1. , |
| 1. , 1. , 1. | | , 0. , 0. , |
| 1. , 0. , 0. | | , 0. , 1. , |
| 0. , 0. , 0. | | , 0.66666667, 1. , |
| 0. , 1. , 1. | | , 1. , 1. , |
| 1. , 1. , 0. | | , 1. , 1. , |
| 1. , 1. , 1. | | , 0. , 0. , |
| 0. , 1. , 0. | | , 1. , 0. , |
| 1. , 1. , 1. | | , 0. , 1. , |
| 1. , 1. , 1. | | , 1. , 0. , |
| 0. , 0.33333333, 0. | | , 0. , 1. , |
| 0. , 0. , 1. | | , 0. , 0. , |
| 1. , 1. , 0. | | , 1. , 1. , |
| 1. , 0.33333333, 1. | | , 0. , 1. , |
| 0. , 1. , 1. | | , 1. , 1. , |
| 1. , 0. , 1. | | , 1. , 0. , |
| 0. , 0. , 1. | | , 0.33333333, 1. , |
| 0. , 1. , 1. | | , 1. , 1. , |
| 0.33333333, 0. , 0. | | , 0. , 1. , |
| 1. , 0.33333333, 1. | | , 1. , 1. , |
| 1. , 0.33333333, 1. | | , 0.66666667, 0.66666667, |
| 1. , 0. , 1. | | , 1. , 0. , |
| 1. , 1. , 0. | | , 0.33333333, 0. , |
| 1. , 1. , 0. | | , 1. , 1. , |
| 0. , 0. , 1. | | , 0. , 1. , |
| 1. , 1. , 1. | | , 0. , 0.33333333]) |

|  |  |
| --- | --- |
| In [48]: |  |
| *# probability of getting output as 4 - malignant cancer* | |
| knn.predict\_proba(X\_test)[:,1] | |
| array([1. , 0. , 1. , 0.66666667, 0. , | |
| 0. , 0. , 0. , 0. , 0. , | |
| 0. , 0. , 0. , 0. , 0. , | |
| 0. , 0. , 0. , 1. , 1. , | |
| 0. , 1. , 1. , 1. , 0. , | |
| 1. , 1. , 1. , 0.33333333, 0. , | |
| 1. , 0. , 0. , 0. , 0. , | |
| 0. , 0. , 1. , 0. , 0. , | |
| 0. , 0. , 0. , 1. , 1. , | |
| 1. , 0. , 1. , 0. , 1. , | |
| 0. , 0. , 0. , 1. , 0. , | |
| 0. , 0. , 0. , 0. , 1. , | |
| 1. , 0.66666667, 1. , 1. , 0. , | |
| 1. , 1. , 0. , 1. , 1. , | |
| 0. , 0. , 1. , 0. , 0. , | |
| 0. , 0.66666667, 0. , 1. , 0. , | |
| 1. , 0. , 0. , 0. , 0. , | |
| 0. , 1. , 0. , 0. , 1. , | |
| 1. , 1. , 0. , 0.66666667, 0. , | |
| 1. , 0. , 0. , 0. , 0. , | |
| 0.66666667, 1. , 1. , 1. , 0. , | |
| 0. , 0.66666667, 0. , 0. , 0. , | |
| 0. , 0.66666667, 0. , 0.33333333, 0.33333333, | |
| 0. , 1. , 0. , 0. , 1. , | |
| 0. , 0. , 1. , 0.66666667, 1. , | |
| 0. , 0. , 1. , 0. , 0. , | |
| 1. , 1. , 0. , 1. , 0. , | |
| 0. , 0. , 0. , 1. , 0.66666667]) | |
| Check accuracy score | |
| In [49]: |  |
| 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 | |
| Compare the train-set and test-set accuracy Now, I will compare the train-set and test-set accuracy | |
| In [51]: |  |
| 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: 0.9803 | |
| Check for overfitting and underfitting | |

|  |  |
| --- | --- |
| In [52]: |  |
| *# 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: 0.9803 Test set score: 0.9714 | |
| The training-set accuracy score is 0.9803 while the test-set accuracy to be 0.9714. These two value 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 ab compare it with the **null accuracy**. Null accuracy is the accuracy that could be achieved by always frequent class.  So, we should first check the class distribution in the test set | |
| In [53]: |  |
| *# check class distribution in test set*  y\_test.value\_counts()  2 85  4 55  Name: Class, dtype: int64 | |
| We can see that the occurences of most frequent class is 85. So, we can calculate null accuracy by number of occurences. | |
| In [54]: |  |
| *# 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 Nearest Neighbors model is doing a very good job in predicting the class labels. | |
| Rebuild kNN Classification model using k=5 | |

|  |  |
| --- | --- |
| In [55]: |  |
| *# 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 | |
| In [56]: |  |
| *# 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 : 0.9643 | |
| Rebuild kNN Classification model using k=7 | |
| In [57]: |  |
| *# 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 : 0.9571 | |
| Rebuild kNN Classification model using k=8 | |

|  |  |
| --- | --- |
| In [58]: |  |
| *# 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 : 0.9643 | |
| Rebuild kNN Classification model using k=9 | |
| In [59]: |  |
| *# 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 : 0.9643 | |
| Interpretation  Our original model accuracy score with k=3 is 0.9714. Now, we can see that we get same accuracy 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. | |
| Confusion matrix  A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusi clear picture of classification model performance and the types of errors produced by the model. It g correct and incorrect predictions broken down by each category. The summary is represented in a t  Four types of outcomes are possible while evaluating a classification model performance. These fo described below:-  **True Positives (TP)** – True Positives occur when we predict an observation belongs to a certain cla actually belongs to that class.  **True Negatives (TN)** – True Negatives occur when we predict an observation does not belong to a observation actually does not belong to that class.  **False Positives (FP)** – False Positives occur when we predict an observation belongs to a certain 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 observation actually belongs to that class. This is a very serious error and it is called **Type II error.** | | | | | | |
| In [60]: |  | | | | | |
| *# Print the Confusion Matrix with k =3 and slice*  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]) | | | *it* | *into* | *four* | *pieces* |
| 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 predicti  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 [61]: |  |
| *# 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 [[82 3]  [ 3 52]]  True Positives(TP) = 82 True Negatives(TN) = 52 False Positives(FP) = 3 False Negatives(FN) = 3 | |
| The above confusion matrix shows '83 + 54 = 137 correct predictions' and '2 + 1 = 4 incorrect predi 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)' | |

|  |  |
| --- | --- |
| In [62]: |  |
| *# 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: > | |
| Classification Report  **Classification report** is another way to evaluate the classification model performance. It displays t and **support** scores for the model. I have described these terms in later. | |
| In [63]: |  |
| from sklearn.metrics import classification\_report print(classification\_report(y\_test, y\_pred\_7))  precision recall f1-score support 2 0.96 0.96 0.96 85  4 0.95 0.95 0.95 55  accuracy 0.96 140  macro avg 0.96 0.96 0.96 140  weighted avg 0.96 0.96 0.96 140 | |
| Classification accuracy | |

|  |  |
| --- | --- |
| In [64]: |  |
| TP **=** cm\_7[0,0] TN **=** cm\_7[1,1] FP **=** cm\_7[0,1] FN **=** cm\_7[1,0] | |
| In [65]: |  |
| *# print classification accuracy*  classification\_accuracy **=** (TP **+** TN) **/** float(TP **+** TN **+** FP **+** FN) print('Classification accuracy : {0:0.4f}'.format(classification\_accuracy))  Classification accuracy : 0.9571 | |
| Classification error | |
| In [66]: |  |
| *# print classification error*  classification\_error **=** (FP **+** FN) **/** float(TP **+** TN **+** FP **+** FN) print('Classification error : {0:0.4f}'.format(classification\_error))  Classification error : 0.0429 | |
| Precision  **Precision** can be defined as the percentage of correctly predicted positive outcomes out of all the p outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (T  So, **Precision** identifies the proportion of correctly predicted positive outcome. It is more concerned than the negative class.  Mathematically, 'precision' can be defined as the ratio of 'TP to (TP + FP)'. | |
| In [67]: |  |
| *# print precision score*  precision **=** TP **/** float(TP **+** FP) print('Precision : {0:0.4f}'.format(precision))  Precision : 0.9647 | |
| Recall | |
| In [68]: |  |
| recall **=** TP **/** float(TP **+** FN)  print('Recall or Sensitivity : {0:0.4f}'.format(recall))  Recall or Sensitivity : 0.9647 | |

|  |  |
| --- | --- |
| True Positive Rate | |
| In [69]: |  |
| true\_positive\_rate **=** TP **/** float(TP **+** FN)  print('True Positive Rate : {0:0.4f}'.format(true\_positive\_rate))  True Positive Rate : 0.9647 | |
| False Positive Rate | |
| In [70]: |  |
| false\_positive\_rate **=** FP **/** float(FP **+** TN)  print('False Positive Rate : {0:0.4f}'.format(false\_positive\_rate))  False Positive Rate : 0.0545 | |
| Specificity | |
| In [71]: |  |
| specificity **=** TN **/** (TN **+** FP)  print('Specificity : {0:0.4f}'.format(specificity))  Specificity : 0.9455 | |
| Adjusting the classification threshold level | |
| In [72]: |  |
| *# 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([[0. , 1. ],  [1. , 0. ],  [0. , 1. ],  [0.33333333, 0.66666667],  [1. , 0. ],  [1. , 0. ],  [1. , 0. ],  [1. , 0. ],  [1. , 0. ],  [1. , 0. ]]) | |

|  |  |
| --- | --- |
| In [73]: |  |
| *# store the probabilities in dataframe*  y\_pred\_prob\_df **=** pd.DataFrame(data**=**y\_pred\_prob, columns**=**['Prob of - benign cancer (2)', 'Prob y\_pred\_prob\_df  **Prob of - benign cancer (2) Prob of - malignant cancer (4)**  **0** 0.000000 1.000000  **1** 1.000000 0.000000  **2** 0.000000 1.000000  **3** 0.333333 0.666667  **4** 1.000000 0.000000  **5** 1.000000 0.000000  **6** 1.000000 0.000000  **7** 1.000000 0.000000  **8** 1.000000 0.000000  **9** 1.000000 0.000000 | |
| In [74]: |  |
| *# print the first 10 predicted probabilities for class 4 - Probability of malignant cancer*  knn.predict\_proba(X\_test)[0:10, 1]  array([1. , 0. , 1. , 0.66666667, 0. ,  0. , 0. , 0. , 0. , 0. ]) | |
| In [76]: |  |
| *# store the predicted probabilities for class 4 - Probability of malignant cancer*  y\_pred\_1 **=** knn.predict\_proba(X\_test)[:, 1] | |

|  |  |
| --- | --- |
| In [77]: |  |
| *# 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') | |
| ROC-AUC | |

|  |  |
| --- | --- |
| In [78]: |  |
| *# 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 performance. In this technique, we measure the area under the curve (AUC) . A perfect classifier wil to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. | |
| In [79]: |  |
| *# 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 : 0.9883 | |
| Interpretation | |

ROC AUC is a single number summary of classifier performance. The higher the value, the bet

ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a whether it is benign or malignant cancer.

In [86]:

*# 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').m print('Cross validated ROC AUC : {:.4f}'.format(Cross\_validated\_ROC\_AUC))

Cross validated ROC AUC : 0.9811