November 2020

CLASSIFICATION OF BENIGN AND MALIGNANT TUMORS BASED ON BREAST CYTOLOGY DATA

Prepared by:
Vipul Choudhary (18UCS030)
Deepanshu Somani (18UCS105)
Amey Prakash Dalal (18UCS189)
Tanvin Kalra (18UCS196)

Aim

Our aim is to apply machine learning classification algorithms on the data set to predict whether a tumor is benign or malignant based on several biological factors, visualize these attributes and analyze their relationship among themselves.

About our data-set

This breast cancer database was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg.
Wisconsin Breast Cancer Database consists of 10 attributes plus the class attribute consisting of various biological factors which are specified below along with their domains:

#	Attribute	Domain
1.	Sample Code Number	ID Number
2.	Clump Thickness	1 – 10
3.	Uniformity of Cell Size	1 – 10
4.	Uniformity of Cell Shape	1 – 10
5.	Marginal Adhesion	1 – 10
6.	Single Epithelial Cell Size	1 – 10
7.	Bare Nuclei	1 – 10
8.	Bland Chromatin	1 – 10
9.	Normal Nucleoli	1 – 10
10.	Mitoses	1 – 10
11.	Class	2 for benign, 4 for malignant

The hyperlink to our data-set is the following:
Wisconsin Breast Cancer Database

Procedure:

1. First, we'll provide a brief data description of our data – set.
2. Next, we'll import all the modules required for doing our tasks.
3. Then, we'll read our data into a pandas Data Frame for initial pre – processing
4. Next, we'll deal with missing values in our data – set by applying a suitable technique.
5. Then we'll analyze each and every dependent attribute and decide whether to consider it for classification or drop it.
6. Next, we'll visualize attributes by plotting them and gain some important inferences.
7. Next, we will prepare the data – set for classification i.e. dividing into training and test set, whether to apply standardization or normalization.
8. Then, we'll decide which ML classification algorithms to use based on our inferences.
9. Next, we'll dig deep to find ways to avoid overfitting/underfitting the model.
10. Next, we'll train our models and test it on our test set.
11. Finally, we will take a peek at how our models have performed under various parameters.

GitHub Repository Link

Data Description

The following data-set can be described by the following parameters:

- Data Set Characteristics: Multivariate
- Number of Instances: 699
- Area: Life
- Attribute Characteristics: Integer
- Number of Attributes: 10
- Associated Tasks: Classification
- Missing Values: Yes

Exploring our Dataset

```
# Importing the required classes and libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

At the beginning of our program, we imported the required classes and libraries for carrying out various tasks with our data.

NumPy has functions for working in domain of linear algebra, Fourier transform, and matrices but in here we only use NumPy for basic mathematical operations.

Pandas is used for reading in the data in a tabular form. Pandas also allows various data manipulation operations such as merging, slicing, selecting etc.

Matplotlib and Seaborn are Python data visualization libraries used for making informative statistical graphics.

Creating a Data frame

```
# create a dataframe
df = pd.read_csv('breast-cancer-wisconsin.data' ,header = None, delimiter=',')
df.columns=['Sample code no.','Clump thickness','Uniformity-cell size',
            'Uniformity-cell shape','Marginal Adhesion',
            'Single Epithelial cell size','Bare Nuclei','Bland Chromatin',
            'Normal Nucleoli','Mitoses','Class']
```

We created a data frame and gave the columns their respective headings.

```
df.head()
```

	Sample code no.	Clump thickness	Uniformity-cell size	Uniformity-cell shape	Marginal Adhesion	Single Epithelial cell size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
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

Here, are the first five rows of our dataset.

```
df.shape
```

```
(699, 11)
```

The df.shape returns a tuple representing the dimensionality of the Data Frame, hence signifying there are 698 rows and 11 columns present in our data – set.

Handling Missing Values

As given in our data description, our data – set has some missing values. There are 16 instances in Groups 1 to 6 that contain a single missing (i.e., unavailable) attribute value, now denoted by "?" (This data was supplied with the data-set we were given to classify).

Here is an instance of the data – set that has a missing (unavailable) attribute value corresponding to the 'Bare Nuclei' attribute:

```
# An instance of data - set that has a missing(unavailable) value
print(df.loc[23,:])
```

```
Sample code no.                1057013
Clump thickness                      8
Uniformity-cell size                 4
Uniformity-cell shape                5
Marginal Adhesion                    1
Single Epithelial cell size          2
Bare Nuclei                          ?
Bland Chromatin                      7
Normal Nucleoli                      3
Mitoses                              1
Class                                4
Name: 23, dtype: object
```

We decided to replace all the missing attribute values with NaN (acronym for Not a Number) instances with the help of NumPy.

```
df.replace('?',np.nan, inplace=True)
```

Now, we have replaced all the missing values from our data – set with np.nan to make our task easier for further data manipulation.

```
print(df.loc[23,:])
```

```
Sample code no.                1057013
Clump thickness                      8
Uniformity-cell size                 4
Uniformity-cell shape                5
Marginal Adhesion                    1
Single Epithelial cell size          2
Bare Nuclei                        NaN
Bland Chromatin                      7
Normal Nucleoli                      3
Mitoses                              1
Class                                4
Name: 23, dtype: object
```

Question – Which technique to use to handle the missing data?

To handle these missing values, we applied the technique of **Imputation Using Most Frequent Values** which works by by replacing missing data with the most frequent values within each column as well as works well with numerical data.

To accomplish this task, we imported the SimpleImputer class from sklearn.impute to transform the Data Frame.

```
from sklearn.impute import SimpleImputer
si = SimpleImputer(missing_values = np.nan , strategy="most_frequent")
df = pd.DataFrame(si.fit_transform(df))
df.columns=['Sample code no.', 'Clump thickness','Uniformity-cell size',
            'Uniformity-cell shape', 'Marginal Adhesion',
            'Single Epithelial cell size','Bare Nuclei','Bland Chromatin',
            'Normal Nucleoli','Mitoses','Class']
```

By applying the above transformation, the missing values will be replaced by the most frequent values.

```
print(df.loc[23,:])
```

```
Sample code no.                1057013
Clump thickness                      8
Uniformity-cell size                 4
Uniformity-cell shape                5
Marginal Adhesion                    1
Single Epithelial cell size          2
Bare Nuclei                          1
Bland Chromatin                      7
Normal Nucleoli                      3
Mitoses                              1
Class                                4
Name: 23, dtype: object
```

Analyzing the 'Sample code no.' column

We counted the number of unique values present in the Sample code no. column as show below:

```
df['Sample code no.'].nunique(dropna = True)
```

```
645
```

Question – Should removing this column help in better classification?

As we can see out of 698 data points in our data – set, there are 644 unique values of the Sample code number. Therefore, we can see the data samples are nearly **independent and identically distributed**, and this attribute doesn't help in identifying the sample class, so we can drop this column.

```
df.drop('Sample code no.' , inplace=True , axis=1)
```

```
df.head()
```

	Clump thickness	Uniformity-cell size	Uniformity-cell shape	Marginal Adhesion	Single Epithelial cell size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
0	5	1	1	1	2	1	3	1	1	2
1	5	4	4	5	7	10	3	2	1	2
2	3	1	1	1	2	2	3	1	1	2
3	6	8	8	1	3	4	3	7	1	2
4	4	1	1	3	2	1	3	1	1	2

Analyzing other attributes

```
df.describe()
```

	Clump thickness	Uniformity-cell size	Uniformity-cell shape	Marginal Adhesion	Single Epithelial cell size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
count	699	699	699	699	699	699	699	699	699	699
unique	10	10	10	10	10	10	10	10	9	2
top	1	1	1	1	2	1	2	1	1	2
freq	145	384	353	407	386	418	166	443	579	458

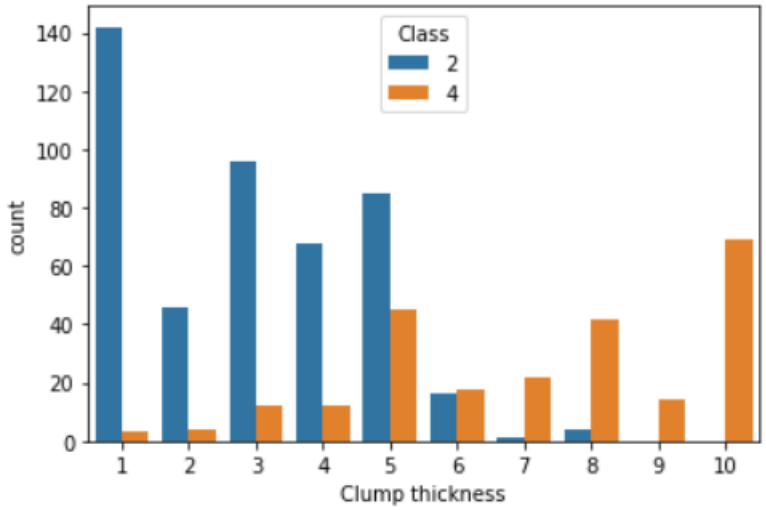
The domain of the dependent variables is from 1 – 10 which we can clearly verify by the above description of the data. But we can see that there are only 9 unique values for the ‘Mitoses’ attribute which tells us that this attribute only takes 9 values.
In the visualization part, we can see that the Mitoses attribute doesn’t take the value ‘9’. The other parameters we can observe from this description is the most occurring value of each attribute and how many times it occurs. By observing this we can say that each attribute’s most frequent value is either 1 or 2.

Visualizing Data:

Clump Thickness Vs Count:

```
sns.countplot(x='Clump thickness', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb470ec470>
```



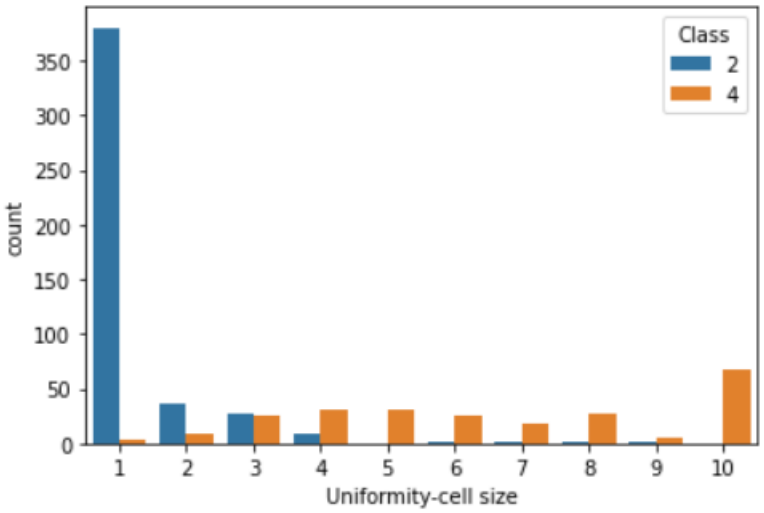
The above plot shows the relationship between Clump Thickness and the count of Benign and Malignant Tumor cases.

Observation:
We can observe from the above plot that Benign tumors are more frequent in regions where there is low Clump - Thickness while Malignant Tumors are more frequent in regions of high clump thickness.

Uniformity-Cell-Size Vs Count:

```
sns.countplot(x='Uniformity-cell size', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb41b31ef0>
```



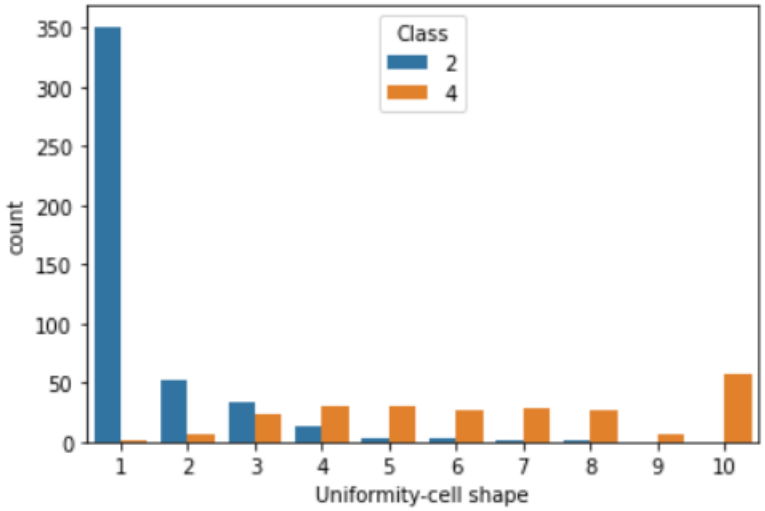
The above plot shows the relationship between Uniformity-cell size and the count of Benign and Malignant Tumor cases.

Observation:
We can observe from the above plot that Benign tumors are more frequent in the lower Uniformity-cell sizes while Malignant Tumors are more frequent in the higher uniformity cell sizes.

Uniformity-Cell-Shape Vs Count:

```
sns.countplot(x='Uniformity-cell shape', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb41687668>
```



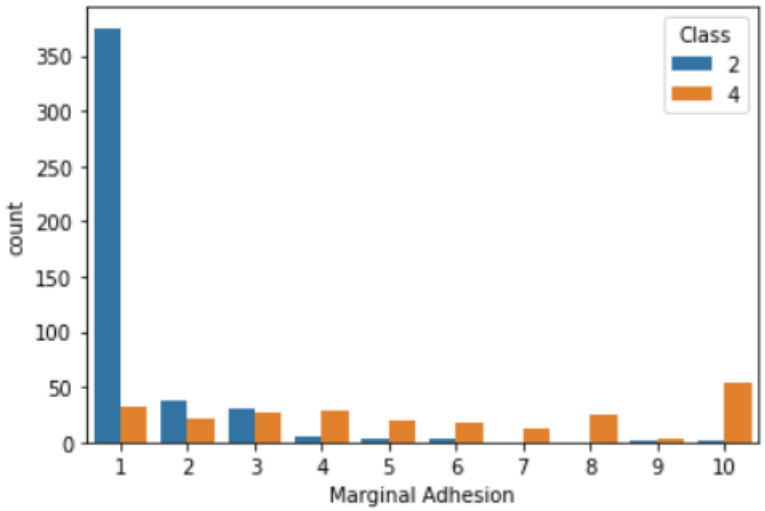
The above plot shows the relationship between the uniformity cell shape and count of Benign and Malignant Tumor cases.

Observation:
We can see from the plot that Benign Tumors are present mostly in cell shapes 1-4 and Malignant Tumors are present mostly in cell shapes 4-10.

Marginal Adhesion Vs Count:

```
sns.countplot(x='Marginal Adhesion', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb415bc320>
```



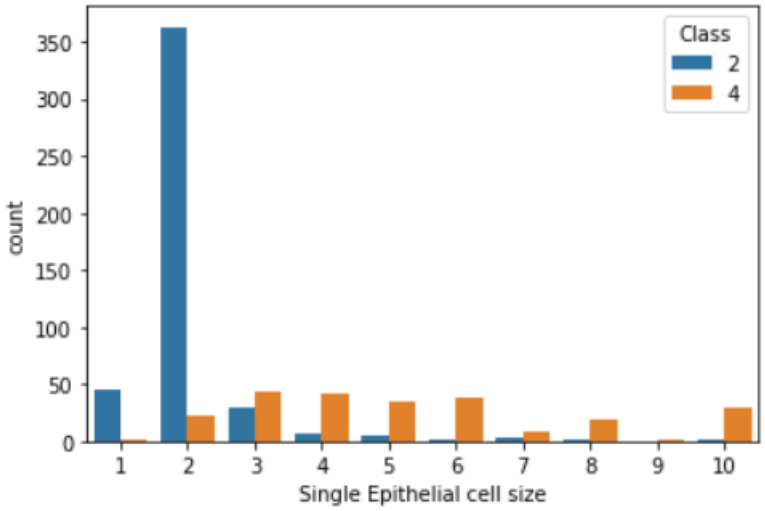
The above plot shows the relationship between Marginal Adhesion and the count of Benign and Malignant Tumor cases.

Observation:
We can see from the above plot that Benign Tumors are most frequent with Marginal Adhesion of 1 and Malignant Tumors are more distributed across the range and the most frequent is the one with Marginal Adhesion of 10.

Single Epithelial cell size Vs Count:

```
sns.countplot(x='Single Epithelial cell size', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb414dae80>
```



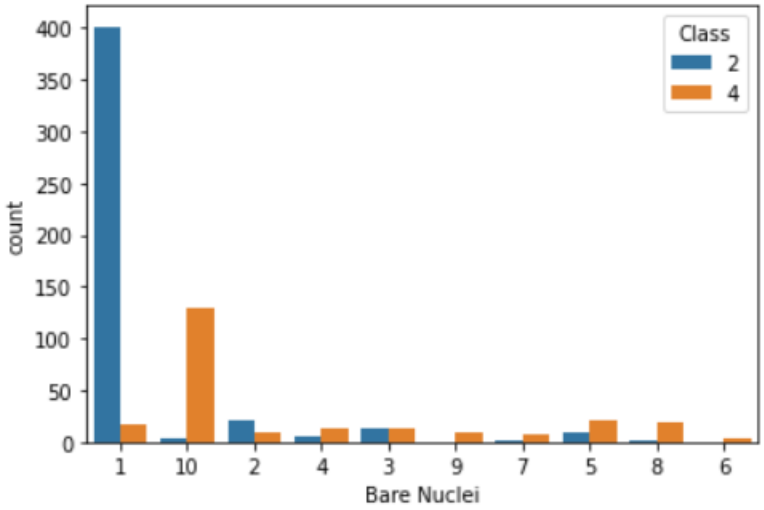
The above plot shows the relationship between Single Epithelial cell size and the count of Benign and Malignant Tumor cases.

Observation:
We can see from the above plot that Benign tumors are most frequent with a Single Epithelial size of 2 and Malignant Tumors follow a more distributed pattern and are most frequent with Single Epithelial cell sizes 3, 4 and 6.

Bare Nuclei Vs Count:

```
sns.countplot(x='Bare Nuclei', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb4145d1d0>
```



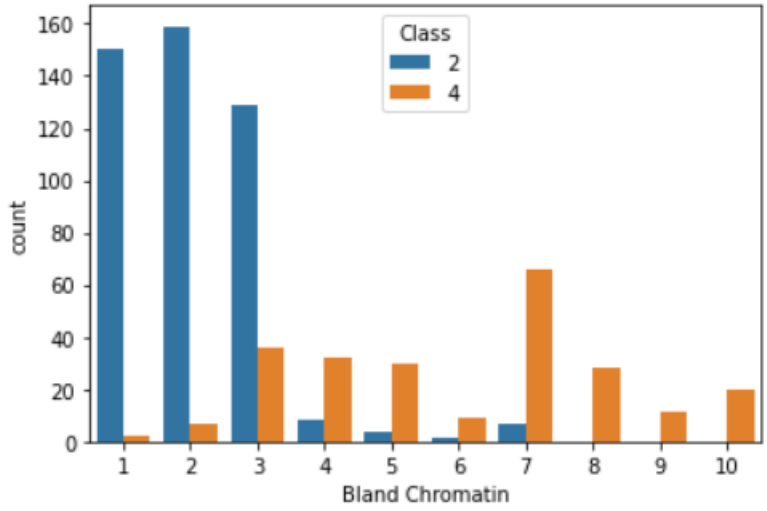
The above plot shows the relationship between Bare nuclei and the count of Benign and Malignant Tumor cases.

Observation:
We can see from the above plot that benign tumors are most frequent with Bare Nuclei 1 and malignant tumors are most frequent with Bare Nuclei 10.

Bland Chromatin Vs Count:

```
sns.countplot(x='Bland Chromatin', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb4146eb00>
```



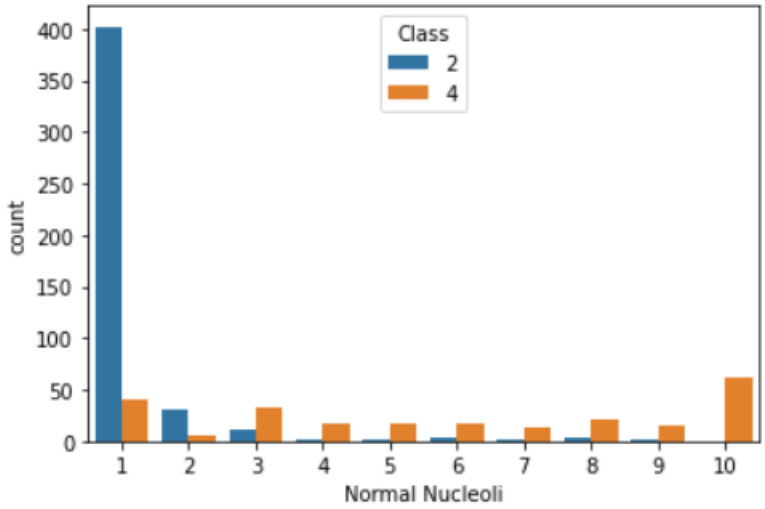
The above plot shows the relationship between Bland Chromatin and the count of Benign and Malignant Tumor cases.

Observation:
We can see that the number of cases for Benign Tumors is more prevalent for Bland chromatin 1, 2 and 3 while Malignant tumors are more frequent in rest of the cases and there are almost no cases of Benign tumors for high values (>=8) of Bland Chromatin.

Normal Nucleoli Vs Count:

```
sns.countplot(x='Normal Nucleoli', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb41338b38>
```



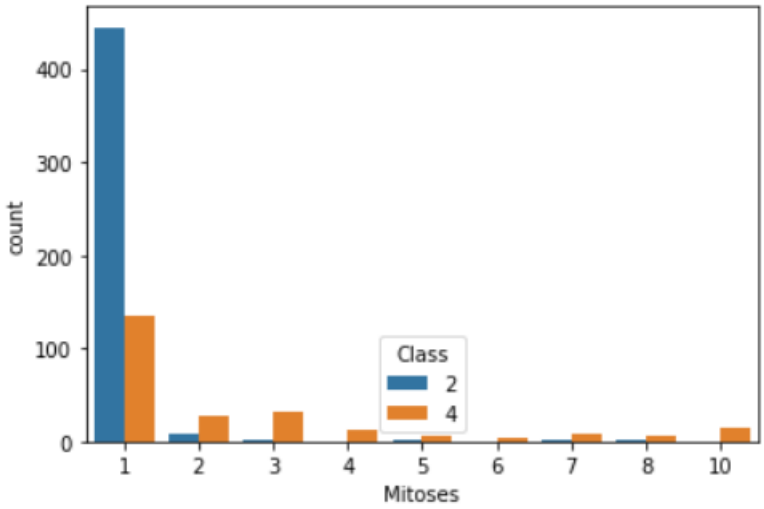
The above plot shows the relationship between the Normal nucleoli and the count of Benign and Malignant Tumor cases.

Observation:
We can see that Benign tumors are mostly present for normal nucleoli values 1, 2 and 3 while malignant tumors are present in almost all values of normal nucleoli (except 2) in some decent amount with a normal nucleoli value of 10 being almost exclusively for malignant tumors.

Mitoses Vs Count:

```
sns.countplot(x='Mitoses', hue='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb41287a58>
```



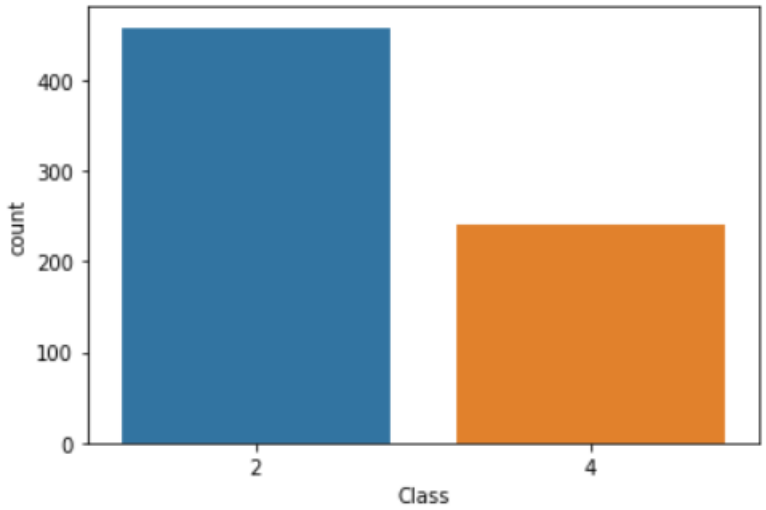
The above plot shows the relationship between Mitoses and the count of Benign and Malignant Tumor cases.

Observation:
We can observe that both benign and malignant tumors are most frequent in Mitoses value 1, and get less frequent as the value increases.

Class Vs Count:

```
sns.countplot(x='Class', data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb4120a4a8>
```



The above plot shows the number of cases for each class of tumors.

Observation:
We can see that Benign tumors are more frequent than Malignant tumors.

```
# Calculating the percentage of each class
counts = df['Class'].value_counts()
print('Benign: ' + str(counts[2]) + ' (' + str(round(counts[2]/counts.sum(), 2)*100) + '%)' )
print('Malignant: ' + str(counts[4]) + ' (' + str(round(counts[4]/counts.sum(), 2)*100) + '%)' )
```

```
Benign: 458 (66.0%)
Malignant: 241 (34.0%)
```

As you can see from the above code snippet, the Class distribution is as follows:

Benign: 458 (66%)
Malignant: 241 (34%)

Preparing data for classification:

Dividing the data into dependent and independent attributes:

```
#Dividing the data into dependent and independent attributes
x = df.iloc[:,:-1].values
y = df.iloc[:,-1].values
y = y.astype('int')
```

```
print(x)
```

```
[[5 1 1 ... 3 1 1]
 [5 4 4 ... 3 2 1]
 [3 1 1 ... 3 1 1]
 ...
 [5 10 10 ... 8 10 2]
 [4 8 6 ... 10 6 1]
 [4 8 8 ... 10 4 1]]
```

```
print(y)
```

```
[2 2 2 2 2 4 2 2 2 2 2 2 4 2 4 4 2 2 4 2 4 4 2 4 2 4 2 2 2 2 2 2 4 2 2 2 4
 2 4 4 2 4 4 4 4 2 4 2 2 4 4 4 4 4 4 4 4 4 4 4 2 4 4 2 4 2 4 4 2 2 4 2 4
 4 2 2 2 2 2 2 2 2 2 4 4 4 4 2 2 2 2 2 2 2 2 2 2 4 4 4 4 2 4 4 4 4 4 2 4 2
 4 4 4 2 2 2 4 2 2 2 2 4 4 4 2 4 2 4 2 2 2 4 2 2 2 2 2 2 2 2 2 4 2 2 2 4 2
 2 4 2 4 4 2 2 4 2 2 2 4 4 2 2 2 2 2 4 4 2 2 2 2 2 4 4 4 2 4 2 4 2 2 2 4 4
 2 4 4 4 2 4 4 2 2 2 2 2 2 2 2 4 4 2 2 2 4 4 2 2 2 4 4 2 4 4 4 2 2 4 2 2 4
 4 4 4 2 4 4 2 4 4 4 2 4 2 2 4 4 4 4 2 2 2 2 2 2 4 4 2 2 2 4 2 4 4 4 2 2 2
 2 4 4 4 4 4 2 4 4 4 2 4 2 4 4 2 2 2 2 2 4 2 2 4 4 4 4 4 2 4 4 2 2 4 4 2 4
 2 2 2 4 4 2 4 2 4 4 2 2 4 2 2 2 4 2 2 2 4 4 2 2 4 2 2 4 2 2 4 2 4 4 4 2 2
 4 4 2 4 2 2 4 4 2 2 2 4 2 2 2 4 4 2 2 2 4 2 2 4 4 4 4 4 4 2 2 2 2 4 4 2 2
 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 4 2 2 2 2 4 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2
 2 2 2 2 2 4 2 4 2 4 2 2 2 2 4 2 2 2 4 2 4 2 2 2 2 2 2 2 4 4 2 2 2 4 2 2 2
 2 2 2 2 2 4 2 2 2 4 2 4 4 4 2 2 2 2 2 2 2 4 4 4 2 2 2 2 2 2 2 2 2 2 2 4 2
 2 4 4 2 2 2 4 4 4 2 4 2 4 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 4 4 2 2
 2 4 2 2 4 4 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2 4 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 4 2 2 4 4 4 4 2 2 4 2 2 2 2 2 2 4 4 2 2 2 4 2 4 2 4 4
 4 2 4 2 2 2 2 2 2 2 2 4 4 4 2 2 4 2 4 4 4 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2
 2 2 2 2 4 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2
 2 2 4 4 4 2 2 2 2 2 2 2 2 2 2 4 4 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 4 4 4]
```

Scaling the data:

Question – Should we Standardize the data?

Standardization (μ=0, σ=1) is applied to attributes of the data - set as variables that are measured at different scales do not contribute equally to the model fitting and might dominate the objective function and make the estimator unable to learn from other features correctly as expected.
But in our data – set the domain of each attribute is exactly same with each other And distribution is also very similar as seen in the above visualizations.
Therefore, Standardization is not required in our case.

Dividing the data – set 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=0.25, random_state=0)
```

We decided to split our data – set in a 75% to 25% ratio corresponding to training size and testing size respectively.

```
print(x_train)
```

```
[[8 10 4 ... 8 2 1]
 [3 1 2 ... 2 1 1]
 [8 10 10 ... 4 8 7]
 ...
 [4 1 1 ... 1 1 1]
 [5 1 1 ... 2 1 1]
 [1 1 1 ... 1 1 1]]
```

```
print(y_train)
```

```
[4 2 4 4 2 4 2 4 2 2 4 4 2 2 2 4 2 4 2 2 2 2 2 2 2 2 2 4 2 2 4 2 4 2 4 4
 4 2 4 4 4 2 4 2 2 4 2 2 2 2 2 2 2 2 4 2 2 4 2 4 2 4 2 4 2 2 4 2 2 2 2 2
 2 2 2 2 2 4 4 4 2 4 2 4 2 4 2 2 2 2 4 2 4 2 2 2 4 2 4 4 2 2 2 4 4 4 4 2 4
 2 2 2 4 4 4 2 2 4 2 2 2 4 4 2 4 2 4 2 2 2 2 2 4 4 2 2 4 2 4 2 4 4 2 4 2 2
 4 2 2 4 2 2 2 2 2 2 2 4 2 2 4 2 2 2 4 4 4 4 4 2 4 2 2 2 2 2 4 4 2 4 4 4 2
 2 2 4 2 4 4 2 4 2 2 2 2 2 4 2 4 2 2 4 4 2 2 2 4 4 4 2 2 2 2 4 2 4 2 2 2 2
 2 2 2 2 2 4 4 4 2 2 2 4 4 2 2 2 2 2 2 2 4 2 4 2 2 2 4 4 2 4 2 2 2 2 4 2 4
 2 4 2 2 4 2 2 2 2 2 2 2 2 4 4 2 2 2 4 2 4 2 2 2 2 2 4 4 2 2 4 2 2 4 4 2 4
 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4 4 2 2 2 2 2 2 4 2 2 2 2 2 2 4 4 2 2 2 2 2
 2 4 4 2 2 4 2 2 2 2 4 2 2 4 4 2 2 4 2 2 2 4 2 2 4 2 4 2 2 4 4 4 2 2 2 4 2
 2 2 2 4 2 2 2 2 4 2 4 2 4 4 2 2 2 4 4 4 2 2 2 2 4 2 2 4 2 4 2 4 2 2 4 2 2
 2 2 2 2 4 2 2 2 4 4 4 4 4 2 2 2 2 4 4 4 4 2 2 4 2 2 2 2 2 4 2 2 4 2 4 2 2
 2 2 2 2 4 4 2 2 4 2 2 2 2 2 2 4 4 4 4 2 2 2 2 2 2 4 2 2 2 2 4 2 2 2 4 4 2
 4 4 4 2 4 2 2 4 2 2 2 2 2 4 2 2 2 4 4 2 2 2 2 2 2 2 4 4 2 2 2 2 2 2 2 2 2
 2 4 2 2 2 2]
```

```
print(x_test)
```

```
[[4 1 2 ... 1 1 1]
 [4 2 2 ... 2 1 1]
 [6 6 6 ... 7 8 1]
 ...
 [5 3 2 ... 1 1 1]
 [1 1 1 ... 3 1 1]
 [4 1 1 ... 3 2 1]]
```

```
print(y_test)
```

```
[2 2 2 2 4 2 4 2 4 4 2 2 4 4 4 2 2 4 4 2 4 4 2 2 2 4 2 2 4 2 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 4 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 4 4 2 2 2 2 2 4
 2 2 2 4 4 2 2 2 2 2 2 2 2 4 2 4 2 2 2 2 2 2 2 2 4 2 2 2 2]
```

Applying classification algorithms:

Question – Which Algorithm to use for classification?

K – Nearest – Neighbor: KNN classifies the data based on the similarity measure of the earlier stored data – points. It works well if there are properly labeled data points, data is noise – free and small in size. Our data – set checks all the above points so we'll be applying this algorithm for classifying our data.

Support Vector Machines: SVM is mostly used to solve classification problems. The algorithm classifies by creating a hyperplane which separates data into classes. SVM tries to finds the "best" margin (distance between the line and the support vectors) that separates the classes and this reduces the risk of error on the data. The chances of overfitting are quite low in SVM and it is quite robust to noise points as well. Therefore, we chose this algorithm for our data – set.

```
# Importing the required models
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

Question – Which things can be done to avoid model to be overfitting/underfitting?

We can apply K – Fold cross validation. This technique ensures that every data – point has the chance of appearing in the training and test set. In this method, the data – set is divided into K parts (folds). Then the model is trained on K – 1 folds and validated on Kth fold. This is quite similar to solving home – work problems before appearing for the final exam. As the model is trained again and again, it can learn from its mistakes and in the end provide an efficient model.

```
# Importing module for K - Fold cross validation
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
k_fold = KFold(n_splits=10, shuffle=True, random_state=0)
```

We have used K – Fold technique by dividing it into 10 parts.

Using K – Fold Cross Validation for our training data – set:

Using SVM:

```
classifier = SVC()
accuracies = cross_val_score(classifier, x_train, y_train, cv=k_fold, scoring='accuracy')
print(accuracies.mean())
```

```
0.97144412191582
```

Using KNN:

Question – How many neighbors to consider for classification?

This is a problem of **Hyperparameter Tuning.** We can use an instance of GridSearchCV class to identify the most optimal value of neighbors to be considered. It will do exhaustive search over specified parameter values for an estimator and then we can choose which value to take.

```
classifier_knn = KNeighborsClassifier()
```

Making a KNN classifier for hyperparameter testing.

```
# Hyperparameter Tuning
from sklearn.model_selection import GridSearchCV
parameters = [{ 'n_neighbors': [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]}]
grid_search = GridSearchCV(estimator=classifier_knn,
                           param_grid=parameters,
                           scoring='accuracy',
                           cv=10,
                           n_jobs= -1)
grid_search.fit(x_train,y_train)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
print("Best Accuracy: {:.2f} %".format(best_accuracy*100))
print("Best Parameters:", best_parameters)
```

```
Best Accuracy: 96.77 %
Best Parameters: {'n_neighbors': 5}
```

By the above code snippet, we can take the value of neighbors to be **5**.

Applying K – Fold cross validation for KNN, we get:

```
accuracies = cross_val_score(classifier_knn, x_train, y_train, cv=k_fold, scoring='accuracy')
print(accuracies.mean())
```

```
0.9655660377358488
```

Using Training and Test set to train the model:

Using SVM:

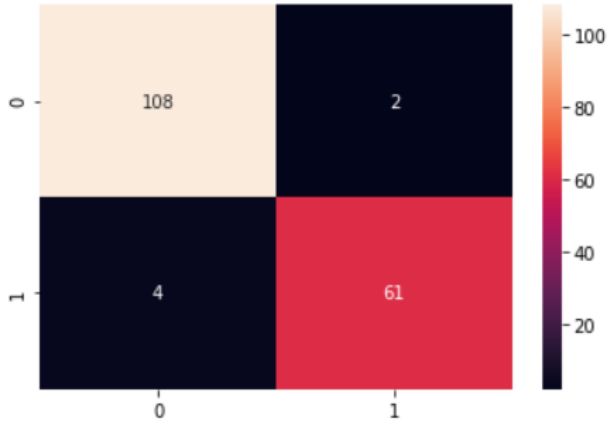
```
# Testing the classifier for test set
classifier_1 = SVC()
classifier_1.fit(x_train, y_train)
y_pred_1 = classifier_1.predict(x_test)
print(y_pred_1)
```

```
[2 2 4 2 4 2 4 2 4 4 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 4 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 4 4 2 2 2 2 2 2
 2 2 2 4 4 4 2 2 2 2 2 2 4 2 4 2 2 2 2 2 2 2 4 2 2 2 2]
```

Now, plotting Confusion Matrix for SVM model:

```
#Generating Confusion mattrix
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
cm_1 = confusion_matrix(y_pred_1, y_test)
print("confusion matrix :")
sns.heatmap(cm_1, annot=True, fmt='d' )
```

```
confusion matrix :
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb35f182e8>
```



Here are the Accuracy Score and Classification Report:

```
print("Classification report: ")
print(str(classification_report(y_pred_1, y_test)))
print("Accuracy Score: " + str(accuracy_score(y_pred_1, y_test)))
```

```
Classification report:
              precision    recall  f1-score   support

           2       0.96      0.98      0.97       110
           4       0.97      0.94      0.95        65

    accuracy                           0.97       175
   macro avg       0.97      0.96      0.96       175
weighted avg       0.97      0.97      0.97       175

Accuracy Score: 0.9657142857142857
```

Using KNN:

For the hyperparameters of the KNN algorithm, we have chosen the minkowski metric as distance metric to use for the tree with p (Power parameter for the Minkowski metric) = 2 which is equivalent to the standard Euclidean metric and the value of neighbors to be 5 as we have found by tuning of the model.

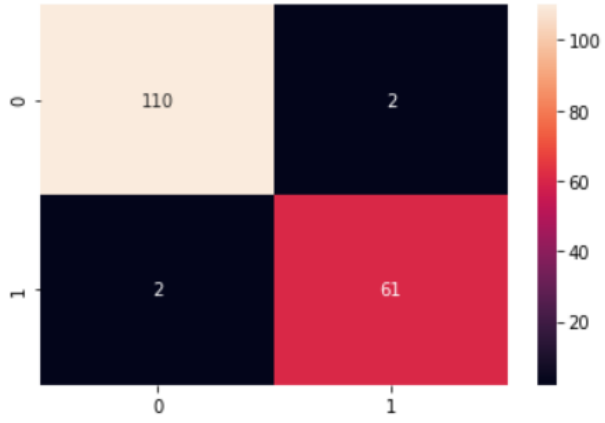
```
classifier_2 = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
classifier_2.fit(x_train, y_train)
y_pred_2 = classifier_2.predict(x_test)
print(y_pred_2)
```

```
[2 2 4 2 4 2 4 2 4 4 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 4 4 2 2 2 2 2 2
 2 2 2 4 4 2 2 2 2 2 2 2 4 2 4 2 2 2 2 2 2 2 4 2 2 2 2]
```

Now, plotting Confusion Matrix for KNN model:

```
#Generating Confusion mattrix
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
cm_2 = confusion_matrix(y_pred_2, y_test)
print("confusion matrix :")
sns.heatmap(cm_2, annot=True, fmt='d' )
```

```
confusion matrix :
<matplotlib.axes._subplots.AxesSubplot at 0x7fbb335cc6a0>
```



Here are the Accuracy Score and Classification Report:

```
print("Classification report: ")
print(str(classification_report(y_pred_2, y_test)))
print("Accuracy Score: " + str(accuracy_score(y_pred_2, y_test)))
```

```
Classification report:
              precision    recall  f1-score   support

           2       0.98      0.98      0.98       112
           4       0.97      0.97      0.97        63

    accuracy                           0.98       175
   macro avg       0.98      0.98      0.98       175
weighted avg       0.98      0.98      0.98       175

Accuracy Score: 0.9771428571428571
```

References

1. O. L. Mangasarian and W. H. Wolberg: "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, September 1990, pp 1 & 18.

2. William H. Wolberg and O.L. Mangasarian: "Multisurface method of pattern separation for medical diagnosis applied to breast cytology", Proceedings of the National Academy of Sciences, U.S.A., Volume 87, December 1990, pp 9193-9196.

3. O. L. Mangasarian, R. Setiono, and W.H. Wolberg: "Pattern recognition via linear programming: Theory and application to medical diagnosis", in: "Large-scale numerical optimization", Thomas F. Coleman and Yuying Li, editors, SIAM Publications, Philadelphia 1990, pp 22-30.

4. K. P. Bennett & O. L. Mangasarian: "Robust linear programming discrimination of two linearly inseparable sets", Optimization Methods and Software 1, 1992, 23-34 (Gordon & Breach Science Publishers).

5. https://stackoverflow.com/