Early Detection of Breast Cancer

SANIYA ANKLESARIA

Technologies Used: Python,

DATASET INFO:

Dataset imported from http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+%28diagnostic%29

Dataset Description 1) ID number 2) Diagnosis (M = malignant, B = benign) 3) 32 features

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter) b) texture (standard deviation of gray-scale values) c) perimeter d) area e) smoothness (local variation in radius lengths) f) compactness (perimeter^2 / area - 1.0) g) concavity (severity of concave portions of the contour) h) concave points (number of concave portions of the contour) i) symmetry j) fractal dimension ("coastline approximation" - 1)

The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features.

Importing Libraries

In [29]:
```
#importing the libraries
import pandas as pd
import numpy as np
import os
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import seaborn as sns
import warnings #to remove warning from the notebook
warnings.filterwarnings(action='ignore')
```

In [30]:
```
dataset=pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data")
```

In [31]:
```
dataset.head()
```

Out[31]:

	842302	M	17.99	10.38	122.8	1001	0.1184	0.2776	0.3001	0.1471	...	25.38	17.33	184.6	2019	0.1622	0.6656	0.7119	0.2654	0.4601	0.1189
0	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902
1	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758
2	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300
3	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678
4	843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000	0.1578	0.08089	...	15.47	23.75	103.40	741.6	0.1791	0.5249	0.5355	0.1741	0.3985	0.12440

5 rows × 32 columns

In [32]:
```
dataset=dataset.shift(1)
dataset.iloc[0]=dataset.columns
```

In [33]:
```
X = dataset.iloc[:, 1:31].values
Y = dataset.iloc[:, 31].values
```

In [34]:
```
print("Cancer data set dimensions : {}".format(dataset.shape))
```

```
Cancer data set dimensions : (568, 32)
```

In [35]:
```
headerList = ['id', 'diagnosis','mean_radius','mean_texture','mean_perimeter','mean_area','mean_smoothness','mean_compactness','mean_concavity','mean_concave points','mean_symmetry
              'SE_radius','SE_texture','SE_perimeter','SE_area','SE_smoothness','SE_compactness','SE_concavity','SE_concave points','SE_symmetry','SE_fractal dimension',
              'worst_radius','worst_texture','worst_perimeter','worst_area','worst_smoothness','worst_compactness','worst_concavity','worst_concave points','worst_symmetry','worst_
# converting data frame to csv
dataset.to_csv("cancer.csv", header=headerList, index=False)
```

In [36]:
```
dataset = pd.read_csv('cancer.csv')
X = dataset.iloc[:, 1:31].values
Y = dataset.iloc[:, 31].values
```

In [37]:
```
dataset.head()
```

Out[37]:

	id	diagnosis	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	mean_compactness	mean_concavity	mean_concave points	...	worst_radius	worst_texture	worst_perimeter	worst_area
0	842302.0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.60	2019.0
1	842517.0	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0
2	84300903.0	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0
3	84348301.0	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7
4	84358402.0	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0

5 rows × 32 columns

In [38]:
```
dataset.shape
```

Out[38]:
```
(568, 32)
```

Finding the diagnosis cell related info

In [39]:
```
dataset['diagnosis'].describe()
```

Out[39]:
```
count     568
unique      2
```

```
unique      2
top         B
freq      356
Name: diagnosis, dtype: object
```

Thus benign types entries are 357 and malign are hence 568-357=211

Visualizing the data

In [40]:
```
lis1=headerList[2:]
lis1
```

Out[40]:
```
['mean_radius',
 'mean_texture',
 'mean_perimeter',
 'mean_area',
 'mean_smoothness',
 'mean_compactness',
 'mean_concavity',
 'mean_concave points',
 'mean_symmetry',
 'mean_fractal dimension',
 'SE_radius',
 'SE_texture',
 'SE_perimeter',
 'SE_area',
 'SE_smoothness',
 'SE_compactness',
 'SE_concavity',
 'SE_concave points',
 'SE_symmetry',
 'SE_fractal dimension',
 'worst_radius',
 'worst_texture',
 'worst_perimeter',
 'worst_area',
 'worst_smoothness',
 'worst_compactness',
 'worst_concavity',
 'worst_concave points',
 'worst_symmetry',
 'worst_fractal dimension']
```

Finding Correlation

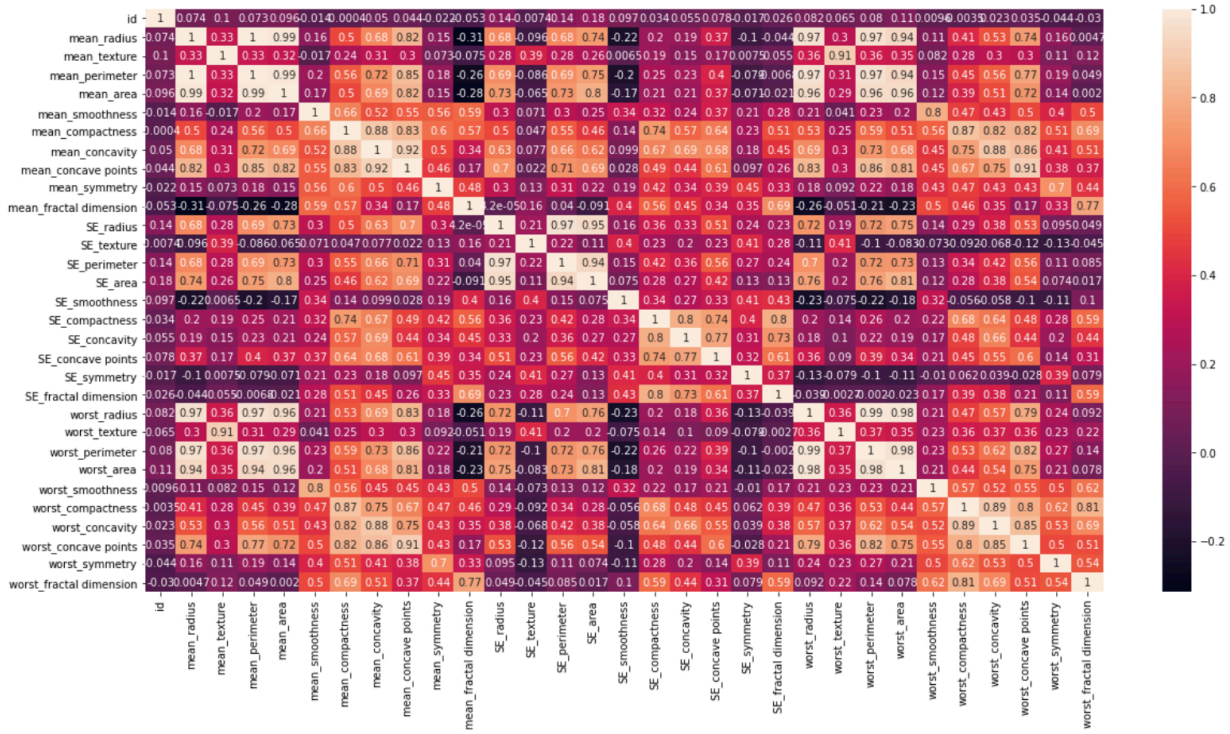
In [41]:
```
cor=dataset.corr()
cor
```

Out[41]:

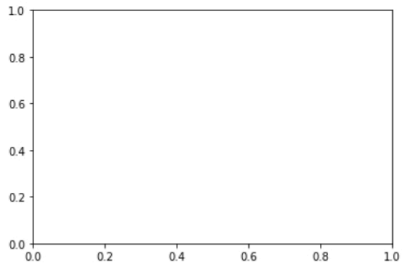
	id	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	mean_compactness	mean_concavity	mean_concave points	mean_symmetry	...	worst_radius	worst_texture	worst
id	1.000000	0.074072	0.100429	0.072600	0.096477	-0.014421	-0.000396	0.049661	0.043683	-0.022479	...	0.081951	0.065083	
mean_radius	0.074072	1.000000	0.329052	0.997843	0.987489	0.162529	0.504501	0.675900	0.822031	0.145636	...	0.969538	0.300461	
mean_texture	0.100429	0.329052	1.000000	0.334847	0.324931	-0.016861	0.239772	0.305552	0.296985	0.073305	...	0.356702	0.912068	
mean_perimeter	0.072600	0.997843	0.334847	1.000000	0.986639	0.199621	0.555519	0.715432	0.850604	0.181040	...	0.969478	0.306526	
mean_area	0.096477	0.987489	0.324931	0.986639	1.000000	0.171383	0.497146	0.685183	0.822744	0.149675	...	0.962624	0.289915	
mean_smoothness	-0.014421	0.162529	-0.016861	0.199621	0.171383	1.000000	0.659225	0.520901	0.552278	0.558391	...	0.207528	0.040627	
mean_compactness	-0.000396	0.504501	0.239772	0.555519	0.497146	0.659225	1.000000	0.882857	0.830711	0.602038	...	0.534015	0.250101	
mean_concavity	0.049661	0.675900	0.305552	0.715432	0.685183	0.520901	0.882857	1.000000	0.921213	0.499900	...	0.687424	0.301864	
mean_concave points	0.043683	0.822031	0.296985	0.850604	0.822744	0.552278	0.830711	0.921213	1.000000	0.461596	...	0.829800	0.295015	
mean_symmetry	-0.022479	0.145636	0.073305	0.181040	0.149675	0.558391	0.602038	0.499900	0.461596	1.000000	...	0.184122	0.091856	
mean_fractal dimension	-0.052768	-0.314410	-0.075350	-0.264108	-0.284976	0.586905	0.565047	0.336143	0.165950	0.479528	...	-0.255604	-0.050554	
SE_radius	0.143026	0.680832	0.276384	0.693556	0.733573	0.303684	0.497914	0.632482	0.698877	0.303459	...	0.716150	0.194994	
SE_texture	-0.007364	-0.096388	0.386091	-0.085797	-0.065484	0.071124	0.047042	0.077065	0.022365	0.128699	...	-0.110948	0.408750	
SE_perimeter	0.137274	0.675633	0.282389	0.694663	0.727435	0.297785	0.549239	0.660820	0.711313	0.313858	...	0.698049	0.200690	
SE_area	0.177586	0.736644	0.261241	0.745800	0.800414	0.246153	0.455327	0.617306	0.690363	0.223472	...	0.757689	0.197263	
SE_smoothness	0.096807	-0.223087	0.006516	-0.203126	-0.166927	0.335519	0.135558	0.098769	0.027802	0.187504	...	-0.230974	-0.074848	
SE_compactness	0.033506	0.203123	0.194966	0.248042	0.210409	0.315644	0.738105	0.669525	0.489108	0.420730	...	0.202313	0.144815	
SE_concavity	0.054845	0.191577	0.145903	0.225581	0.205682	0.244938	0.569599	0.690630	0.437863	0.341639	...	0.184782	0.101816	
SE_concave points	0.078208	0.372349	0.168733	0.403575	0.369562	0.374598	0.641201	0.682452	0.614225	0.392032	...	0.355141	0.089651	
SE_symmetry	-0.016994	-0.102270	0.007521	-0.079495	-0.070859	0.206772	0.231888	0.179773	0.097208	0.450715	...	-0.126536	-0.078566	
SE_fractal dimension	0.025567	-0.043992	0.055361	-0.006767	-0.020831	0.283977	0.507200	0.449099	0.257127	0.331473	...	-0.038511	-0.002682	
worst_radius	0.081951	0.969538	0.356702	0.969478	0.962624	0.207528	0.534015	0.687424	0.829800	0.184122	...	1.000000	0.362641	
worst_texture	0.065083	0.300461	0.912068	0.306526	0.289915	0.040627	0.250101	0.301864	0.295015	0.091856	...	0.362641	1.000000	
worst_perimeter	0.079522	0.965098	0.362252	0.970376	0.958986	0.233420	0.589059	0.728867	0.855485	0.217622	...	0.993686	0.367880	
worst_area	0.106842	0.941328	0.346669	0.941804	0.959172	0.202765	0.508535	0.675303	0.809198	0.175924	...	0.984070	0.347823	
worst_smoothness	0.009574	0.114375	0.081859	0.145475	0.119667	0.804457	0.564150	0.447023	0.450626	0.425539	...	0.212978	0.228745	
worst_compactness	-0.003488	0.411331	0.281157	0.453815	0.388665	0.470480	0.865482	0.754392	0.666561	0.472343	...	0.474260	0.363110	
worst_concavity	0.022681	0.525054	0.304692	0.562182	0.511092	0.432114	0.815805	0.883833	0.751680	0.432742	...	0.572594	0.370870	
worst_concave points	0.034523	0.742788	0.300269	0.769962	0.720962	0.499094	0.815174	0.861150	0.909994	0.429183	...	0.786576	0.363264	
worst_symmetry	-0.044247	0.164274	0.105250	0.189512	0.143686	0.397450	0.510723	0.409819	0.376166	0.700172	...	0.243838	0.233213	
worst_fractal dimension	-0.030203	0.004690	0.121040	0.048785	0.001959	0.499725	0.687003	0.514274	0.367690	0.437806	...	0.091833	0.220356	

31 rows × 31 columns

In [42]:
```
fig=plt.figure(figsize=(20,10))
b=sns.heatmap(cor,annot=True)
```



In [79]:
```
for name in lis1:
    plotdata=dataset[name]
    a=plotdata.plot(kind="hist");
    a.figure.savefig(r'C:\Users\saniy\Downloads\mini\graph_images\ '+name)
    a.clear()
```

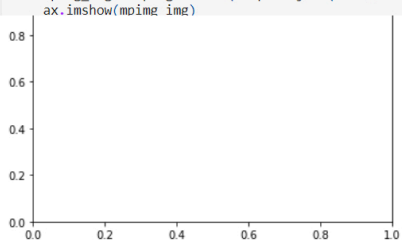


In [80]:
```
images=['mean_radius','mean_texture','mean_perimeter','mean_area','mean_smoothness','mean_compactness','mean_concavity','mean_concave points','mean_symmetry','mean_fractal dimensio
```

Displaying the graphs thus generated

MEAN VALUE GRAPHS

In [81]:
```
i=0
fig = plt.figure(figsize=(25, 4),dpi=400)
for idx in np.arange(10):
    ax = fig.add_subplot(2, 10/2, idx+1, xticks=[], yticks=[])
    mpimg_img = mpimg.imread(os.path.join(r'C:\Users\saniy\Downloads\mini\graph_images', images[i]+'.png'))
    ax.imshow(mpimg_img)
```

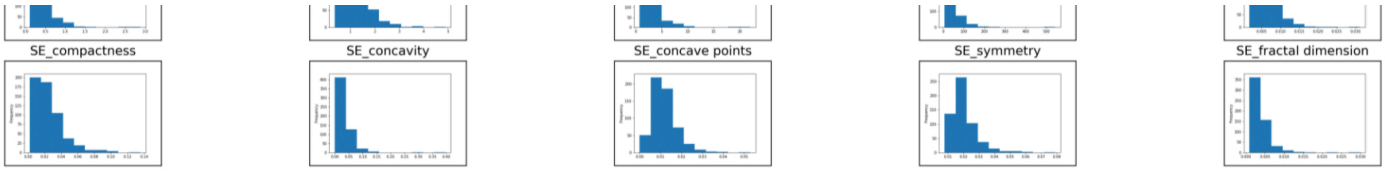


In [80]:
```
images=['mean_radius','mean_texture','mean_perimeter','mean_area','mean_smoothness','mean_compactness','mean_concavity','mean_concave points','mean_symmetry','mean_fractal dimensio
```

Displaying the graphs thus generated

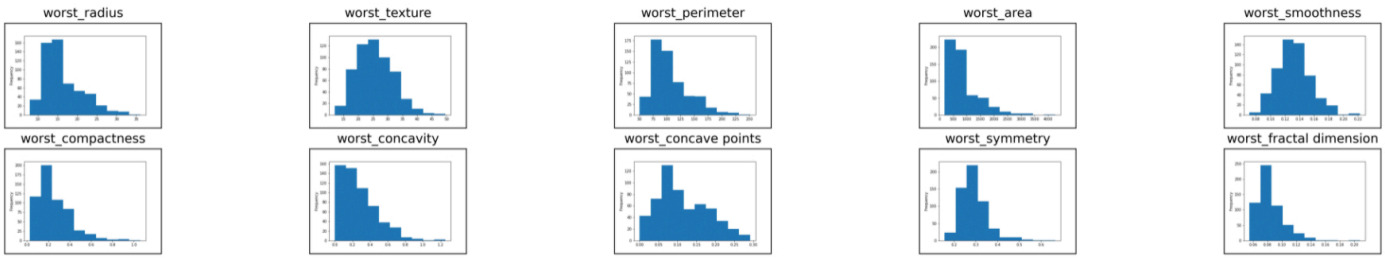
MEAN VALUE GRAPHS

In [81]:
```
i=0
fig = plt.figure(figsize=(25, 4),dpi=400)
for idx in np.arange(10):
    ax = fig.add_subplot(2, 10/2, idx+1, xticks=[], yticks=[])
    mpimg_img = mpimg.imread(os.path.join(r'C:\Users\saniy\Downloads\mini\graph_images', images[i]+'.png'))
    ax.imshow(mpimg_img)
    ax.set_title(images[i])
    i+=1
```



WORST VALUE GRAPHS

In [83]:
```
i=20
fig = plt.figure(figsize=(25, 4),dpi=400)
for idx in np.arange(10):
    ax = fig.add_subplot(2, 10/2, idx+1, xticks=[], yticks=[])
    mpimg_img = mpimg.imread(os.path.join(r'C:\Users\saniy\Downloads\mini\graph_images', images[i]+'.png'))
    ax.imshow(mpimg_img)
    ax.set_title(images[i])
    i+=1
```



Inference: The dataset contains features highly varying in magnitudes, units and range and thus needs to be scaled accordingly. Also the data is skewed at times to the left or right and hence needs to be normalised too.

Missing or Null Data points

We can find any missing or null data points of the data set (if there is any) using the following pandas function.

In [43]:
```
dataset.isnull().sum()
dataset.isna().sum()
```

Out[43]:
```
id                         0
diagnosis                  0
mean_radius                0
mean_texture               0
mean_perimeter             0
mean_area                  0
mean_smoothness            0
mean_compactness           0
mean_concavity             0
mean_concave points        0
mean_symmetry              0
mean_fractal dimension     0
SE_radius                  0
SE_texture                 0
SE_perimeter               0
SE_area                    0
SE_smoothness              0
SE_compactness             0
SE_concavity               0
SE_concave points          0
SE_symmetry                0
SE_fractal dimension       0
worst_radius               0
worst_texture              0
worst_perimeter            0
worst_area                 0
worst_smoothness           0
worst_compactness          0
worst_concavity            0
worst_concave points       0
worst_symmetry             0
worst_fractal dimension    0
dtype: int64
```

Since there aren't any null values present in the dataset we don't need to unecessarily delete any rows.

Categorical Data to Numerical Data thus easing the ML process Label Encoder to label the categorical data. Label Encoder is the part of SciKit Learn library in Python and used to convert categorical data, or text data, into numbers, which our predictive models can better understand.

In [44]:
```
from sklearn.preprocessing import LabelEncoder
labelencoder_Y = LabelEncoder()
dataset['diagnosis'] = labelencoder_Y.fit_transform(dataset['diagnosis'])
```

In [45]:
```
dataset.tail()
```

Out[45]:

	id	diagnosis	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	mean_compactness	mean_concavity	mean_concave points	...	worst_radius	worst_texture	worst_perimeter	worst_area
563	926125.0	1	20.92	25.09	143.0	1347.0	0.10990	0.2236	0.31740	0.14740	...	24.29	29.41	179.1	1819.0
564	926424.0	1	21.56	22.39	142.0	1479.0	0.11100	0.1159	0.24390	0.13890	...	25.45	26.40	166.1	2027.0
565	926682.0	1	20.13	28.25	131.2	1261.0	0.09780	0.1034	0.14400	0.09791	...	23.69	38.25	155.0	1731.0
566	926954.0	1	16.60	28.08	108.3	858.1	0.08455	0.1023	0.09251	0.05302	...	18.98	34.12	126.7	1124.0
567	927241.0	1	20.60	29.33	140.1	1265.0	0.11780	0.2770	0.35140	0.15200	...	25.74	39.42	184.6	1821.0

5 rows × 32 columns

Thus we have converted the categorical diagnosis to numerical with malign being 1

Undersampling

In [46]:
```
minority_len=len(dataset.loc[dataset['diagnosis']==1])
```

```
print(minority_len)
```

```
212
```

In [47]:
```
majority_indices=dataset[dataset['diagnosis']==0].index
print(majority_indices)
```

```
Int64Index([ 19,  20,  21,  37,  46,  48,  49,  50,  51,  52,
            ...
            552, 553, 554, 555, 556, 557, 558, 559, 560, 561],
           dtype='int64', length=356)
```

In [48]:
```
rand_majority_indices=np.random.choice(majority_indices, minority_len, replace=False)
print(len(rand_majority_indices))
```

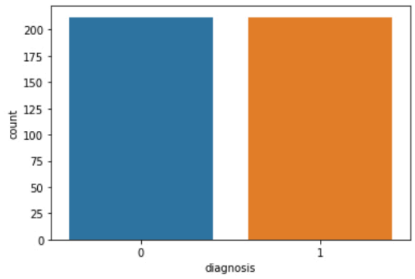
```
212
```

In [49]:
```
minority_indices= dataset[dataset['diagnosis']==1].index
print(minority_indices)
```

```
Int64Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
            ...
            521, 533, 535, 536, 562, 563, 564, 565, 566, 567],
           dtype='int64', length=212)
```

In [50]:
```
underSample_indices=np.concatenate([minority_indices, rand_majority_indices])
dataset=dataset.loc[underSample_indices]
sns.countplot(x=dataset['diagnosis'], data=dataset)
#X=underSample.loc[:, df.columns!=]
#Y=underSample.loc[:, df.columns==]
```

Out[50]:
```
<AxesSubplot:xlabel='diagnosis', ylabel='count'>
```



In [51]:
```
dataset.shape
```

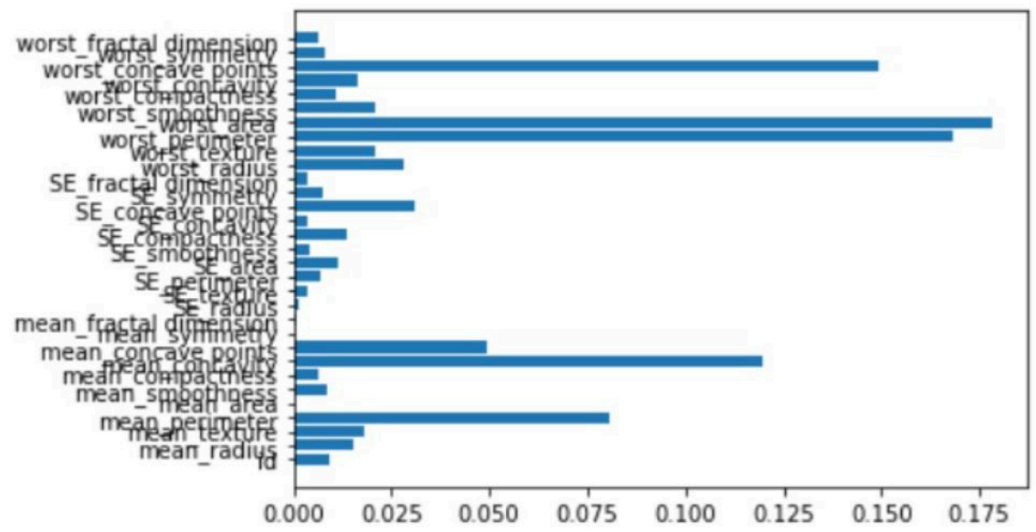
Out[51]:
```
(424, 32)
```

Feature Selection

In [5]:
```
#classifier.feature_importances_
#Top 8 most important features
#n=8
#a=[np.argsort( classifier.feature_importances_)[-n:]]
#a
#X.iloc[:,[21, 18,  8,  3,  7, 28, 23, 24]]
#plt.barh(X.columns, classifier.feature_importances_)
```

In [4]:
```
from IPython.display import Image
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\Features.jpg')
```

Out[4]:



In [52]:
```
dataset=dataset.iloc[:,[1,22,  9,  4,  8, 29, 24, 25]]
```

In [53]:
```
dataset.head()
```

Out[53]:

diagnosis	worst_radius	mean_concave_points	mean_perimeter	mean_concavity	worst_concave_points	worst_perimeter	worst_area

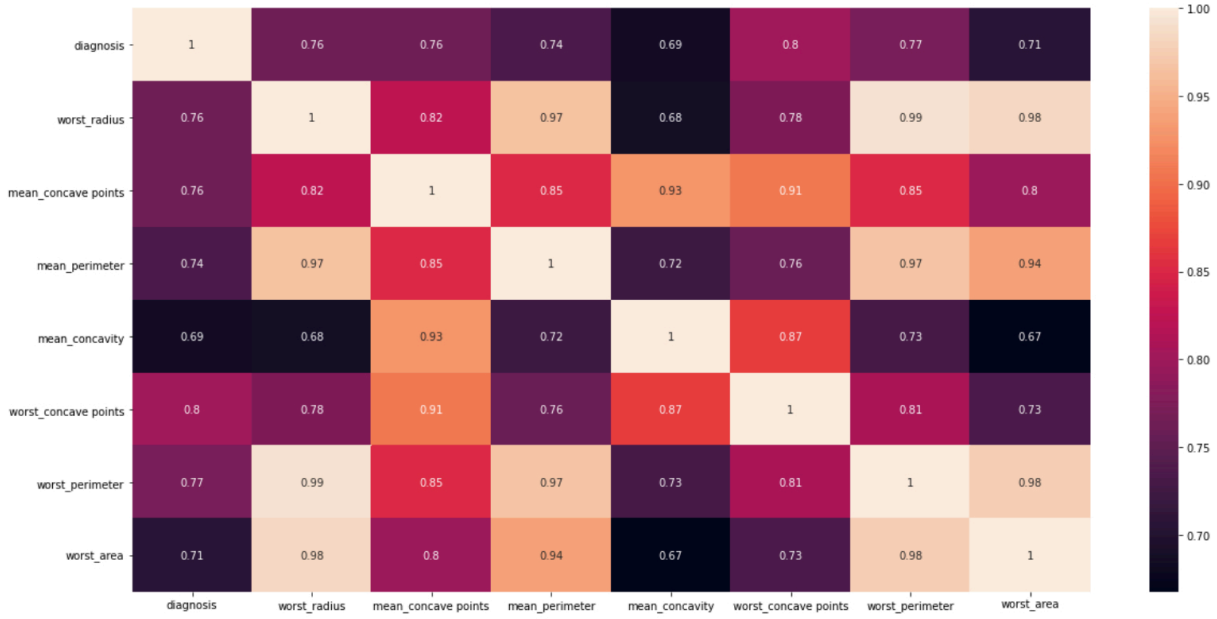
0	1	25.38	0.14710	122.80	0.3001	0.2654	184.60	2019.0
1	1	24.99	0.07017	132.90	0.0869	0.1860	158.80	1956.0
2	1	23.57	0.12790	130.00	0.1974	0.2430	152.50	1709.0
3	1	14.91	0.10520	77.58	0.2414	0.2575	98.87	567.7
4	1	22.54	0.10430	135.10	0.1980	0.1625	152.20	1575.0

In [54]:
```
cor=dataset.corr()
cor
```

Out[54]:

	diagnosis	worst_radius	mean_concave points	mean_perimeter	mean_concavity	worst_concave points	worst_perimeter	worst_area
diagnosis	1.000000	0.763053	0.763772	0.735453	0.686499	0.801884	0.770451	0.705661
worst_radius	0.763053	1.000000	0.823464	0.966047	0.684385	0.775959	0.993260	0.984404
mean_concave points	0.763772	0.823464	1.000000	0.851333	0.927385	0.907900	0.852504	0.798763
mean_perimeter	0.735453	0.966047	0.851333	1.000000	0.722473	0.758851	0.967712	0.939801
mean_concavity	0.686499	0.684385	0.927385	0.722473	1.000000	0.865752	0.730062	0.667475
worst_concave points	0.801884	0.775959	0.907900	0.758851	0.865752	1.000000	0.807322	0.733862
worst_perimeter	0.770451	0.993260	0.852504	0.967712	0.730062	0.807322	1.000000	0.977505
worst_area	0.705661	0.984404	0.798763	0.939801	0.667475	0.733862	0.977505	1.000000

In [55]:
```
fig=plt.figure(figsize=(20,10))
b=sns.heatmap(cor,annot=True)
```



Splitting the dataset

In [56]:
```
X=dataset.drop(['diagnosis'],axis=1)
X.head()
```

Out[56]:

	worst_radius	mean_concave points	mean_perimeter	mean_concavity	worst_concave points	worst_perimeter	worst_area
0	25.38	0.14710	122.80	0.3001	0.2654	184.60	2019.0
1	24.99	0.07017	132.90	0.0869	0.1860	158.80	1956.0
2	23.57	0.12790	130.00	0.1974	0.2430	152.50	1709.0
3	14.91	0.10520	77.58	0.2414	0.2575	98.87	567.7
4	22.54	0.10430	135.10	0.1980	0.1625	152.20	1575.0

In [57]:
```
Y=dataset['diagnosis']
Y.head()
```

Out[57]:
```
0    1
1    1
2    1
3    1
4    1
Name: diagnosis, dtype: int32
```

Selecting independent variables according to random forest feature importance

In [60]:
```
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)
```

Feature Scaling

We need to bring all features to the same level of magnitudes. This can be achieved by scaling. This means that transforming the data so that it fits within a specific scale, like 0–100 or 0–1. We will use StandardScaler method from SciKit-Learn library.

In [61]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

```
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Example of a single entry in the data thus produced

In [62]:
```
X_test[0]
```

Out[62]:
```
array([ 0.06517251,  1.64354009,  0.43231344,  2.22587597,  1.30022925,
        0.19263339, -0.08378639])
```

False Negative (FN): It refers to the number of predictions where the classifier incorrectly predicts the positive class as negative.

Thus FN is of much importance to us because if a breast cancer patient is shown as not having breast cancer it would be a potential threat to her life as compared to if a non breast cancer patient is shown as having breast cancer in which case she could be sent for further testing

In [2]:
```
from IPython.display import Image
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\Confusion_matrix.png', width=300)
```

Out[2]:

Precision: It tells you what fraction of predictions as a positive class were actually positive. To calculate precision, use the following formula: TP/(TP+FP).

Recall: It tells you what fraction of all positive samples were correctly predicted as positive by the classifier. It is also known as True Positive Rate (TPR), Sensitivity, Probability of Detection. To calculate Recall, use the following formula: TP/(TP+FN).

F1-score: It combines precision and recall into a single measure. Mathematically it's the harmonic mean of precision and recall. It can be calculated as follows:

Model Selection

Using Logistic Regression Algorithm to the Training Set

In [67]:
```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, Y_train)
```

Out[67]:
```
LogisticRegression(random_state=0)
```

Predicting the results on the test dataset

In [68]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [69]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [70]:
```
cm
```

Out[70]:
```
array([[40,  4],
       [ 2, 60]], dtype=int64)
```

In [71]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.94
Precision: 0.94
Recall: 0.97
F1-score: 0.95
Sensitivity:0.91
Specificity: 0.97
```

Using GaussianNB method of naïve_bayes class to use Naïve Bayes Algorithm

In [72]:
```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, Y_train)
```

Out[72]:
```
GaussianNB()
```

Predicting the results on the test dataset

In [73]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [74]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [75]:
```
cm
```

Out[75]:
```
array([[40,  4],
       [ 5, 57]], dtype=int64)
```

```
[ 5, 57]], dtype=int64)
```

In [76]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.92
Precision: 0.93
Recall: 0.92
F1-score: 0.93
Sensitivity:0.91
Specificity: 0.92
```

Using DecisionTreeClassifier of tree class to use Decision Tree Algorithm

In [77]:
```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
classifier.fit(X_train, Y_train)
```

Out[77]:
```
DecisionTreeClassifier(criterion='entropy', random_state=0)
```

Predicting the results on the test dataset

In [78]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [79]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [80]:
```
cm
```

Out[80]:
```
array([[39,  5],
       [ 5, 57]], dtype=int64)
```

In [81]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.91
Precision: 0.92
Recall: 0.92
F1-score: 0.92
Sensitivity:0.89
Specificity: 0.92
```

Using Random Forest Classifier method of ensemble class to use Random Forest Classification algorithm

In [82]:
```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(X_train, Y_train)
```

Out[82]:
```
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=0)
```

Predicting the results on the test dataset

In [83]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [84]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [85]:
```
cm
```

Out[85]:
```
array([[42,  2],
       [ 2, 60]], dtype=int64)
```

In [86]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.96
Precision: 0.97
Recall: 0.97
F1-score: 0.97
Sensitivity:0.95
Specificity: 0.97
```

Using KNeighbors Classifier Method of neighbors class to use Nearest Neighbor algorithm

In [92]:
```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 3, metric = 'minkowski', p = 2)
#The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric.
classifier.fit(X_train, Y_train)
```

Out[92]:
```
KNeighborsClassifier(n_neighbors=3)
```

Predicting the results on the test dataset

In [93]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [94]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [95]:
```
cm
```

Out[95]:
```
array([[41,  3],
       [ 3, 59]], dtype=int64)
```

In [96]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.94
Precision: 0.95
Recall: 0.95
F1-score: 0.95
Sensitivity:0.93
Specificity: 0.95
```

Using SVC method of svm class to use Support Vector Machine Algorithm

In [97]:
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(X_train, Y_train)
```

Out[97]:
```
SVC(kernel='linear', random_state=0)
```

Predicting the results on the test dataset

In [98]:
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [99]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [100…
```
cm
```

Out[100…
```
array([[40,  4],
       [ 2, 60]], dtype=int64)
```

In [101…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.94
Precision: 0.94
Recall: 0.97
F1-score: 0.95
Sensitivity:0.91
Specificity: 0.97
```

Using SVC method of svm class to use Kernel SVM Algorithm

In [102…
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', random_state = 0, gamma='auto')
classifier.fit(X_train, Y_train)
```

Out[102…
```
SVC(gamma='auto', random_state=0)
```

Predicting the results on the test dataset

In [103…
```
Y_pred = classifier.predict(X_test)
```

Testing via the confusion matrix

In [104…
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
```

In [105…
```
cm
```

Out[105…
```
array([[41,  3],
       [ 2, 60]], dtype=int64)
```

In [106…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.95
Precision: 0.95
Recall: 0.97
F1-score: 0.96
Sensitivity:0.93
Specificity: 0.97
```

Using ANN

In [112…

```
#importing keras
import keras
#importing sequential module
from keras.models import Sequential
# import dense module for hidden layers
from keras.layers import Dense
#importing activation functions
from keras.layers import LeakyReLU,PReLU,ELU
from keras.layers import Dropout
```

In [113…

```
#creating model
classifier = Sequential()
```

In [114…

```
#first hidden layer
classifier.add(Dense(units=9,kernel_initializer='he_uniform',activation='relu',input_dim=7))
#second hidden layer
classifier.add(Dense(units=9,kernel_initializer='he_uniform',activation='relu'))
classifier.add(Dense(units=9,kernel_initializer='he_uniform',activation='relu'))
# last layer or output layer
classifier.add(Dense(units=1,kernel_initializer='glorot_uniform',activation='sigmoid'))
```

In [115…

```
#compiling the ANN
classifier.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
```

In [116…

```
#fitting the ANN to the training set
model = classifier.fit(X_train, Y_train, batch_size=10, epochs=100)
```

```
Epoch 1/100
32/32 [==============================] - 0s 1ms/step - loss: 0.4514 - accuracy: 0.8711
Epoch 2/100
32/32 [==============================] - 0s 1ms/step - loss: 0.2902 - accuracy: 0.9277
Epoch 3/100
32/32 [==============================] - 0s 978us/step - loss: 0.2363 - accuracy: 0.9497
Epoch 4/100
32/32 [==============================] - 0s 1ms/step - loss: 0.2054 - accuracy: 0.9528
Epoch 5/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1855 - accuracy: 0.9497
Epoch 6/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1708 - accuracy: 0.9560
Epoch 7/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1606 - accuracy: 0.9591
Epoch 8/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1529 - accuracy: 0.9591
Epoch 9/100
32/32 [==============================] - 0s 902us/step - loss: 0.1474 - accuracy: 0.9591
Epoch 10/100
32/32 [==============================] - 0s 804us/step - loss: 0.1429 - accuracy: 0.9591
Epoch 11/100
32/32 [==============================] - 0s 804us/step - loss: 0.1392 - accuracy: 0.9591
Epoch 12/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1362 - accuracy: 0.9591
Epoch 13/100
32/32 [==============================] - 0s 805us/step - loss: 0.1342 - accuracy: 0.9623
Epoch 14/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1318 - accuracy: 0.9623
Epoch 15/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1305 - accuracy: 0.9591
Epoch 16/100
32/32 [==============================] - 0s 965us/step - loss: 0.1292 - accuracy: 0.9560
Epoch 17/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1266 - accuracy: 0.9623
Epoch 18/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1249 - accuracy: 0.9591
Epoch 19/100
32/32 [==============================] - 0s 933us/step - loss: 0.1232 - accuracy: 0.9623
Epoch 20/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1226 - accuracy: 0.9654
Epoch 21/100
32/32 [==============================] - 0s 934us/step - loss: 0.1210 - accuracy: 0.9654
Epoch 22/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1195 - accuracy: 0.9623
Epoch 23/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1186 - accuracy: 0.9623
Epoch 24/100
32/32 [==============================] - 0s 772us/step - loss: 0.1173 - accuracy: 0.9654
Epoch 25/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1160 - accuracy: 0.9654
Epoch 26/100
32/32 [==============================] - 0s 999us/step - loss: 0.1143 - accuracy: 0.9654
Epoch 27/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1132 - accuracy: 0.9654
Epoch 28/100
32/32 [==============================] - 0s 965us/step - loss: 0.1122 - accuracy: 0.9654
Epoch 29/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1108 - accuracy: 0.9654
Epoch 30/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1111 - accuracy: 0.9686
Epoch 31/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1086 - accuracy: 0.9654
Epoch 32/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1078 - accuracy: 0.9654
Epoch 33/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1065 - accuracy: 0.9686
Epoch 34/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1069 - accuracy: 0.9654
Epoch 35/100
32/32 [==============================] - 0s 997us/step - loss: 0.1057 - accuracy: 0.9686
Epoch 36/100
32/32 [==============================] - 0s 937us/step - loss: 0.1044 - accuracy: 0.9717
Epoch 37/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1028 - accuracy: 0.9717
Epoch 38/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1032 - accuracy: 0.9717
Epoch 39/100
32/32 [==============================] - 0s 1ms/step - loss: 0.1011 - accuracy: 0.9748
Epoch 40/100
```

```
32/32 [==============================] - 0s 967us/step - loss: 0.1007 - accuracy: 0.9717
Epoch 41/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0997 - accuracy: 0.9717
Epoch 42/100
32/32 [==============================] - 0s 999us/step - loss: 0.0989 - accuracy: 0.9717
Epoch 43/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0984 - accuracy: 0.9717
Epoch 44/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0976 - accuracy: 0.9717
Epoch 45/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0965 - accuracy: 0.9748
Epoch 46/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0963 - accuracy: 0.9748
Epoch 47/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0960 - accuracy: 0.9717
Epoch 48/100
32/32 [==============================] - 0s 869us/step - loss: 0.0944 - accuracy: 0.9748
Epoch 49/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0938 - accuracy: 0.9748
Epoch 50/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0933 - accuracy: 0.9748
Epoch 51/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0926 - accuracy: 0.9748
Epoch 52/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0922 - accuracy: 0.9748
Epoch 53/100
32/32 [==============================] - 0s 869us/step - loss: 0.0923 - accuracy: 0.9748
Epoch 54/100
32/32 [==============================] - 0s 965us/step - loss: 0.0922 - accuracy: 0.9748
Epoch 55/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0909 - accuracy: 0.9748
Epoch 56/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0896 - accuracy: 0.9748
Epoch 57/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0896 - accuracy: 0.9748
Epoch 58/100
32/32 [==============================] - 0s 997us/step - loss: 0.0884 - accuracy: 0.9748
Epoch 59/100
32/32 [==============================] - 0s 998us/step - loss: 0.0893 - accuracy: 0.9748
Epoch 60/100
32/32 [==============================] - 0s 836us/step - loss: 0.0897 - accuracy: 0.9748
Epoch 61/100
32/32 [==============================] - 0s 933us/step - loss: 0.0886 - accuracy: 0.9748
Epoch 62/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0863 - accuracy: 0.9748
Epoch 63/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0858 - accuracy: 0.9748
Epoch 64/100
32/32 [==============================] - 0s 965us/step - loss: 0.0858 - accuracy: 0.9748
Epoch 65/100
32/32 [==============================] - 0s 952us/step - loss: 0.0850 - accuracy: 0.9780
Epoch 66/100
32/32 [==============================] - 0s 901us/step - loss: 0.0844 - accuracy: 0.9717
Epoch 67/100
32/32 [==============================] - 0s 872us/step - loss: 0.0836 - accuracy: 0.9748
Epoch 68/100
32/32 [==============================] - 0s 835us/step - loss: 0.0834 - accuracy: 0.9780
Epoch 69/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0822 - accuracy: 0.9748
Epoch 70/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0824 - accuracy: 0.9748
Epoch 71/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0823 - accuracy: 0.9748
Epoch 72/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0829 - accuracy: 0.9780
Epoch 73/100
32/32 [==============================] - 0s 865us/step - loss: 0.0805 - accuracy: 0.9748
Epoch 74/100
32/32 [==============================] - 0s 933us/step - loss: 0.0803 - accuracy: 0.9780
Epoch 75/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0802 - accuracy: 0.9780
Epoch 76/100
32/32 [==============================] - 0s 772us/step - loss: 0.0790 - accuracy: 0.9748
Epoch 77/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0789 - accuracy: 0.9780
Epoch 78/100
32/32 [==============================] - 0s 985us/step - loss: 0.0784 - accuracy: 0.9780
Epoch 79/100
32/32 [==============================] - 0s 901us/step - loss: 0.0789 - accuracy: 0.9780
Epoch 80/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0775 - accuracy: 0.9780
Epoch 81/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0786 - accuracy: 0.9780
Epoch 82/100
32/32 [==============================] - 0s 772us/step - loss: 0.0769 - accuracy: 0.9748
Epoch 83/100
32/32 [==============================] - 0s 965us/step - loss: 0.0772 - accuracy: 0.9748
Epoch 84/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0760 - accuracy: 0.9780
Epoch 85/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0753 - accuracy: 0.9780
Epoch 86/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0748 - accuracy: 0.9780
Epoch 87/100
32/32 [==============================] - 0s 901us/step - loss: 0.0741 - accuracy: 0.9780
Epoch 88/100
32/32 [==============================] - 0s 2ms/step - loss: 0.0738 - accuracy: 0.9780
Epoch 89/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0735 - accuracy: 0.9780
Epoch 90/100
32/32 [==============================] - 0s 869us/step - loss: 0.0740 - accuracy: 0.9780
Epoch 91/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0731 - accuracy: 0.9780
Epoch 92/100
32/32 [==============================] - 0s 963us/step - loss: 0.0735 - accuracy: 0.9780
Epoch 93/100
32/32 [==============================] - 0s 915us/step - loss: 0.0721 - accuracy: 0.9780
Epoch 94/100
32/32 [==============================] - 0s 770us/step - loss: 0.0727 - accuracy: 0.9780
Epoch 95/100
32/32 [==============================] - 0s 942us/step - loss: 0.0723 - accuracy: 0.9780
Epoch 96/100
32/32 [==============================] - 0s 902us/step - loss: 0.0702 - accuracy: 0.9780
Epoch 97/100
32/32 [==============================] - 0s 868us/step - loss: 0.0707 - accuracy: 0.9780
Epoch 98/100
32/32 [==============================] - 0s 933us/step - loss: 0.0711 - accuracy: 0.9780
Epoch 99/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0712 - accuracy: 0.9780
Epoch 100/100
32/32 [==============================] - 0s 1ms/step - loss: 0.0703 - accuracy: 0.9780
```

In [117…

```
Y_pred=classifier.predict(X_test)
Y_pred=(Y_pred>0.5)
Y_pred
```

Out[117…

```
array([[ True],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [False],
       [ True],
       [ True],
       [False],
       [False],
       [ True],
       [False],
       [False],
       [ True],
       [ True],
       [ True],
       [False],
       [False],
       [False],
       [False],
       [ True],
       [ True],
       [False],
       [False],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [ True],
       [ True],
       [False],
       [False],
       [ True],
       [False],
       [ True],
       [False],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [ True],
       [False],
       [ True],
       [False],
       [False],
       [False],
       [False],
       [ True],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [ True],
       [False],
       [False],
       [ True],
       [False],
       [False],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [ True],
       [ True],
       [False],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True],
       [False],
       [ True],
       [ True],
       [False],
       [ True]])
```

In [118…

```
cm=confusion_matrix(Y_test, Y_pred)
print(cm)
```

```
[[41  3]
 [ 2 60]]
```

In [119…

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

```
Accuracy: 0.95
Precision: 0.95
Recall: 0.97
F1-score: 0.96
Sensitivity:0.93
Specificity: 0.97
```

Using XGBOOST

In [120…
```
from sklearn.ensemble import GradientBoostingClassifier
```

In [121…
```
from xgboost import XGBClassifier
```

In [122…
```
xgb_clf = XGBClassifier()
xgb_clf.fit(X_train, Y_train)
```

```
[14:16:50] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/learner.cc:1095: Starting in XGBoost 1.3.0, the default evaluation metric used with the objectiv
e 'binary:logistic' was changed from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the old behavior.
```

Out[122…
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [123…
```
score = xgb_clf.score(X_test, Y_test)
print(score)
```

```
0.9339622641509434
```

In [124…
```
Y_pred = xgb_clf.predict(X_test)
Y_pred = [round(value) for value in Y_pred]
```

In [125…
```
cm=confusion_matrix(Y_test, Y_pred)
print(cm)
```

```
[[40  4]
 [ 3 59]]
```

In [126…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
sensitivity = cm[0,0]/(cm[0,0]+cm[0,1])
specificity = cm[1,1]/(cm[1,0]+cm[1,1])
print('\nAccuracy: {:.2f}'.format(accuracy_score(Y_test, Y_pred)))
print('Precision: {:.2f}'.format(precision_score(Y_test, Y_pred)))
print('Recall: {:.2f}'.format(recall_score(Y_test, Y_pred)))
print('F1-score: {:.2f}'.format(f1_score(Y_test, Y_pred)))
print('Sensitivity:{:.2f}'.format(sensitivity))
print('Specificity: {:.2f}'.format(specificity))
```

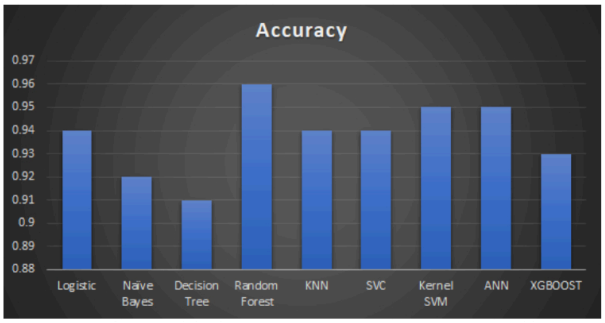
```
Accuracy: 0.93
Precision: 0.94
Recall: 0.95
F1-score: 0.94
Sensitivity:0.91
Specificity: 0.95
```

Conclusion

Accuracy

In [129…
```
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\Accuracy.png')
```

Out[129…

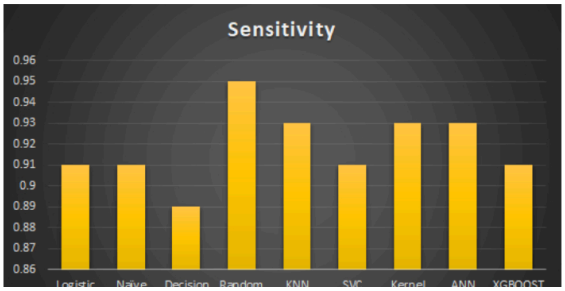


Random Forest has the highest accuracy of 96% while Decision tree showed the lowest accuracy of 91%

Sensitivity

In [130…
```
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\Sensitivity.png')
```

Out[130…

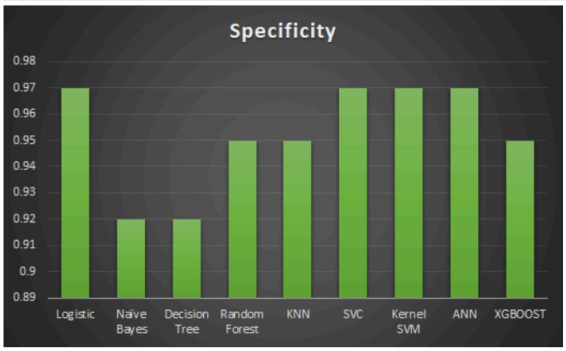


Random Forest has the highest sensitivity of 95% while Decision tree showed the lowest sensitivity of 89%

Specificity

In [131…]
```
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\Specificity.png')
```

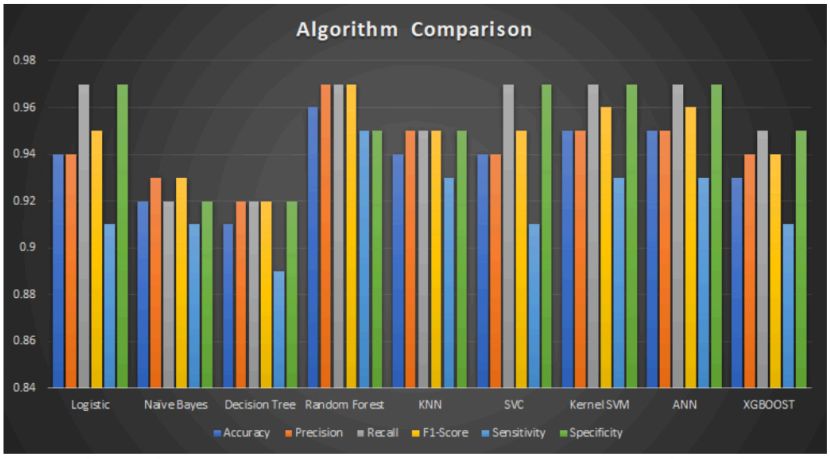
Out[131…]



Random Forest has an average specificity of 95% which was expected.

In [3]:
```
Image(filename=r'C:\Users\saniy\Downloads\ml_proj\allPara.png')
```

Out[3]:



We observed that Random Forest Algorithm proved to be the best model which fit our dataset with an Accuracy of 96% and Sensitivity of 95% followed by Kernel SVM and ANN with an accuracy of 95%. The higher a test's specificity, the less often it will incorrectly find a result it is not supposed to.

Since the goal of our project is to identify everyone who has breast cancer, the number of false negatives should be low and hence we have a high sensitivity. This is especially important in our case since the consequence of failing to treat the condition are serious and very effective.