Breast Cancer Wisconsin (Diagnostic)

Whether the cancer is benign or malignant?

1. Data Introduction

Identify the problem

Excluding skin cancer, breast cancer is the most common type of cancer in women in the United States, accounting for one of every three-cancer diagnosis. It ranks second among cancer deaths in women. Breast Cancer occurs as a result of abnormal growth of cells in the breast tissue, commonly referred to as a Tumor. A tumor does not mean cancer - tumors can be benign (not cancerous), pre-malignant (pre-cancerous), or malignant (cancerous).

Objectives

The goal of this project is the application of several data mining and machine learning techniques to classify whether the tumor mass is benign or malignant in women residing in the state of Wisconsin, USA. This will help in understanding the important underlaying importance of attributes thereby helping in predicting the stage of breast cancer depending on the values of these attributes. Through the understanding of nature of attributes in cancer prediction, the healthcare community can perform additional research corresponding to these attributes to help prevent pervasion of breast cancer into the population of USA.

Data Sources

The features in the dataset are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes. The dataset contains information of 569 women

Attribute Information:

1. ID number 2) Diagnosis (M = malignant, B = benign) 3-32)

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 variables are divided into three parts first is Mean (3-13), Stranded Error (13-23) and Worst (23-32) and each contain 10 parameters (radius, texture, area, perimeter, smoothness, compactness, concavity, concave points, symmetry and fractal dimension). Mean is the means of the all cells, standard Error of all cell and worst means the worst cell.

2. Data Wrangling

As we see below, the dataset contains 33 columns but there are one missing values column and one redundant column. So, we will drop these two columns “Unnamed:32” and “id” from our dataset.

```
: data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   id                       569 non-null    int64  
 1   diagnosis                569 non-null    object 
 2   radius_mean              569 non-null    float64
 3   texture_mean             569 non-null    float64
 4   perimeter_mean           569 non-null    float64
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst             569 non-null    float64
 23  texture_worst            569 non-null    float64
 24  perimeter_worst          569 non-null    float64
 25  area_worst               569 non-null    float64
 26  smoothness_worst         569 non-null    float64
 27  compactness_worst        569 non-null    float64
 28  concavity_worst          569 non-null    float64
 29  concave points_worst     569 non-null    float64
 30  symmetry_worst           569 non-null    float64
 31  fractal_dimension_worst  569 non-null    float64
 32  Unnamed: 32              0 non-null      float64
dtypes: float64(31), int64(1), object(1)
memory usage: 146.8+ KB
```

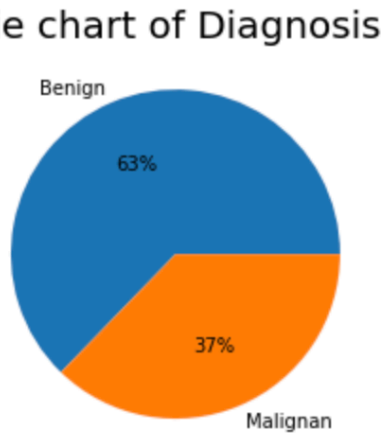
Checking and confirming there are non-null values in each feature and their corresponding data type. As we see below, all features even the label are numerical, therefore processing of categorical variables is something that will not be done in this project, we will only explore the distribution and meaning of each numerical.

```
: data = data.drop(labels= ["Unnamed: 32", "id"], axis=1)

: data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   diagnosis                569 non-null    object 
 1   radius_mean              569 non-null    float64
 2   texture_mean             569 non-null    float64
 3   perimeter_mean           569 non-null    float64
 4   area_mean                569 non-null    float64
 5   smoothness_mean          569 non-null    float64
 6   compactness_mean         569 non-null    float64
 7   concavity_mean           569 non-null    float64
 8   concave points_mean      569 non-null    float64
 9   symmetry_mean            569 non-null    float64
 10  fractal_dimension_mean   569 non-null    float64
 11  radius_se                569 non-null    float64
 12  texture_se               569 non-null    float64
 13  perimeter_se             569 non-null    float64
 14  area_se                  569 non-null    float64
 15  smoothness_se            569 non-null    float64
 16  compactness_se           569 non-null    float64
 17  concavity_se             569 non-null    float64
 18  concave points_se        569 non-null    float64
 19  symmetry_se              569 non-null    float64
 20  fractal_dimension_se     569 non-null    float64
 21  radius_worst             569 non-null    float64
 22  texture_worst            569 non-null    float64
 23  perimeter_worst          569 non-null    float64
 24  area_worst               569 non-null    float64
 25  smoothness_worst         569 non-null    float64
 26  compactness_worst        569 non-null    float64
 27  concavity_worst          569 non-null    float64
 28  concave points_worst     569 non-null    float64
 29  symmetry_worst           569 non-null    float64
 30  fractal_dimension_worst  569 non-null    float64
dtypes: float64(30), object(1)
memory usage: 137.9+ KB
```

3. Exploratory Data Analysis

One of the main goals of visualizing the data here is to observe which features are most helpful in predicting malignant or benign cancer. The other is to see general trends that may aid us in model selection and hyper parameter selection.

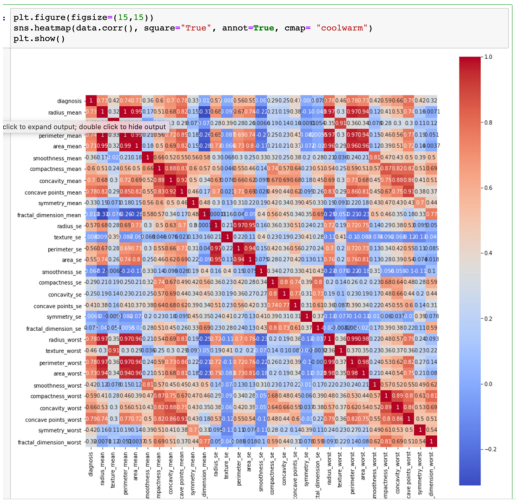
The pie chart below shows the frequency of cancer diagnosis:



The dataset contains a class imbalance between the malignant and benign. 357 observations which account for 63% of all observations indicating the absence of cancer cells, 212 which accounts for 37 % of all observations shows the presence of cancerous cell. The percent is unusually large; the dataset does not represent in this case a typical medical analysis distribution. Typically, we will have a considerable large number of cases that represents negative vs. a small number of cases that represents positives (malignant) tumor.

Visualize Data

An important step in exploratory data analysis step is to identify if at all there is any correlation between variables. By using Pearson's correlation, the below plot was created. The positive correlation between two variables is demonstrated through the red color. The reddest is the strongest positive correlation between respective variables. Similarly, negative correlation between two variables is demonstrated through the blue color. The darkest of blue is the strongest is the negative correlation between respective variables.



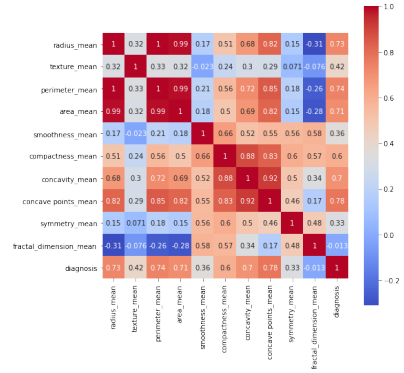
There are quite a few variables that are correlated. Often, we have features that are highly correlated and those provide redundant information. By eliminating highly correlated features we can avoid a predictive bias for the information contained in these features. This also shows us, that when we want to make statements about the biological/ medical importance of specific features, we need to keep in mind that just because they are suitable to predicting an outcome, they are not necessarily causal - they could simply be correlated with causal factors.

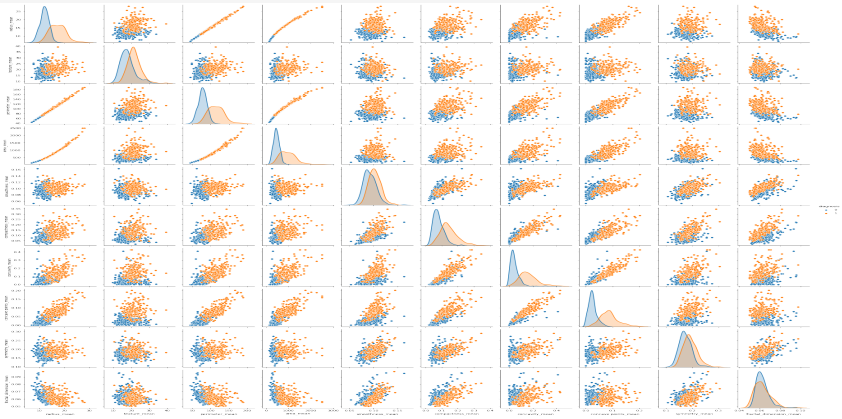
In order to have a better understand between these variables mentioned above. Scatterplots is plotted the correlation between the pairs of two variables.

Mean

```
data_mean = data[['radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean','diagnosis']]
```

```
plt.figure(figsize=(8,8))
sns.heatmap(data_mean.corr(), square="True", annot=True, cmap= "coolwarm")
```



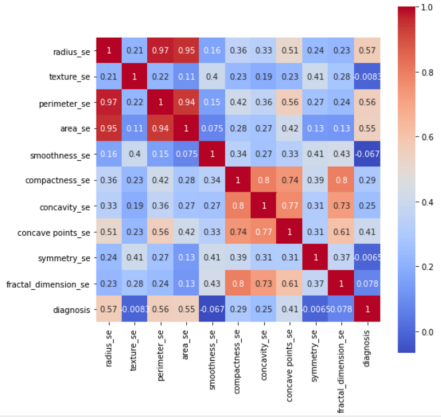


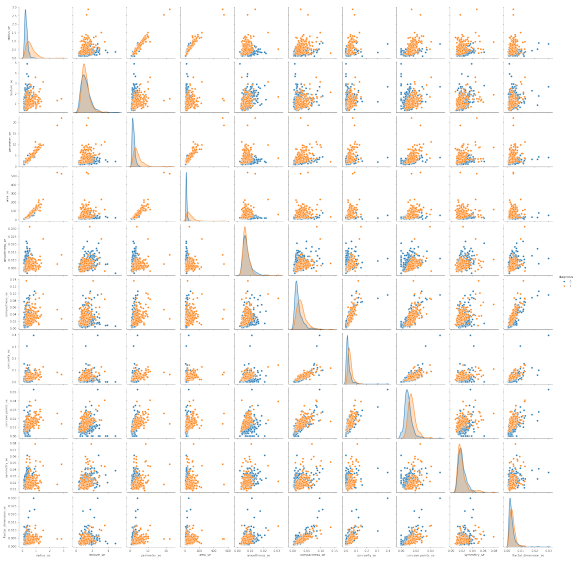
SE

```
data_se = data[['radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se','diagnosis']]
```

```
plt.figure(figsize=(8,8))
sns.heatmap(data_se.corr(), square="True", annot=True, cmap= "coolwarm")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa3a42ea250>
```

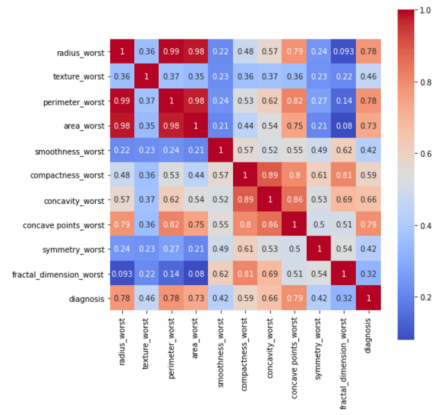


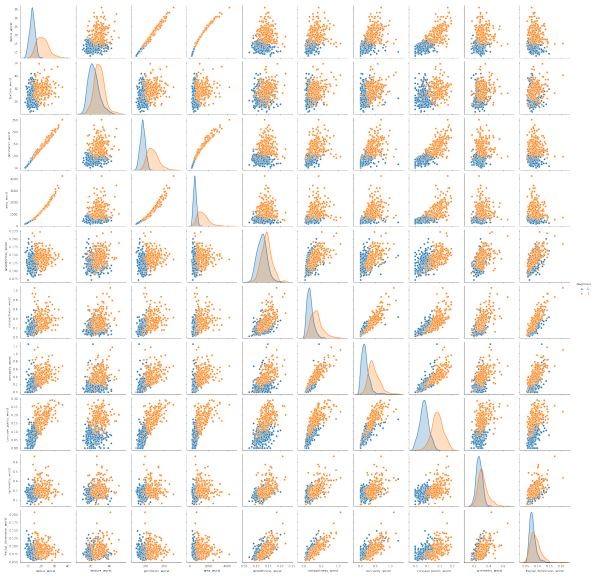


```
data_worst = data[['radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst','diagnosis']]
```

```
plt.figure(figsize=(8,8))
sns.heatmap(data_worst.corr(), square="True", annot=True, cmap= "coolwarm")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa39f649280>
```





Most of the features are normally distributed. Comparison of radius distribution by malignancy shows that there is no perfect separation between any of the features; we do have fairly good separations for concave.points_worst, concavity_worst, perimeter_worst, area_mean, perimeter_mean. We do have as well tight superposition for some of the values, like symmetry_se, smoothness_se .

4. Pre-Processing Model

As the important features in a binary classification model can be selected using a technique called information value (IV), this method is employed in the selection of features for the prediction of the cancer. Only the features exhibiting IV statistics within a range of 0.1-0.8 are selected for building a model. While the IV statistics within a range of 01.-0.8 are selected for building a model. While the IV statistics less than 0.1 are not useful for modeling, a value greater than 0.8 could lead biased and suspicious relationship with a dependent variable. To reduce the degree of multicollinearity between the identified features, a technique called variance inflation factor (VIF) is further used in feature selection. Features exhibiting VIF greater than 5 exhibit extreme multicollinearity and are avoided. This helps to make features independent and ensure that the model can easily predict the dependent variable. These techniques reduce the number of features to 7. ('compactness_se', 'fractal_dimension_se', 'fractal_dimension_worst', 'smoothness_mean', 'smoothness_se', 'symmetry_mean', 'symmetry_se')

```
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
max_bin = 20
force_bin = 3

# define a binning function
def mono_bin(Y, X, n = max_bin):
    df1 = pd.DataFrame({"X": X, "Y": Y})
    justmiss = df1[['X','Y']][df1.X.isnull()]
    notmiss = df1[['X','Y']][df1.X.notnull()]
    #print("justmiss", justmiss)
    #print("notmiss", notmiss)
    r = 0
    while np.abs(r) < 1:
        try:
            d1 = pd.DataFrame({"X": notmiss.X, "Y": notmiss.Y, "Bucket": pd.qcut(notmiss.X, n)})
            d2 = d1.groupby('Bucket', as_index=True)
            r, p = stats.spearmanr(d2.mean().X, d2.mean().Y)
            #print("I am here 1",r, n,len(d2))
            n = n - 1

        except Exception as e:
            n = n - 1
            #print("I am here e",n)

    if len(d2) == 1:
        #print("I am second step ",r, n)
        n = force_bin
        bins = algos.quantile(notmiss.X, np.linspace(0, 1, n))
        if len(np.unique(bins)) == 2:
            bins = np.insert(bins, 0, 1)
            bins[1] = bins[1]-(bins[1]/2)
        d1 = pd.DataFrame({"X": notmiss.X, "Y": notmiss.Y, "Bucket": pd.cut(notmiss.X, np.unique(bins),include_lowest=1
        d2 = d1.groupby('Bucket', as_index=True)

    d3 = pd.DataFrame({},index=[])
    d3["MIN_VALUE"] = d2.min().X
    d3["MAX_VALUE"] = d2.max().X
    d3["COUNT"] = d2.count().Y
    d3["EVENT"] = d2.sum().Y
    d3["NONEVENT"] = d2.count().Y - d2.sum().Y
    d3=d3.reset_index(drop=True)

    if len(justmiss.index) > 0:
        d4 = pd.DataFrame({'MIN_VALUE':np.nan},index=[0])
        d4["MAX_VALUE"] = np.nan
        #print(justmiss.count().Y)
        d4["COUNT"] = justmiss.count().Y
        d4["EVENT"] = justmiss.sum().Y
        d4["NONEVENT"] = justmiss.count().Y - justmiss.sum().Y
        d3 = d3.append(d4,ignore_index=True)

    d3["EVENT_RATE"] = d3.EVENT/d3.COUNT
    d3["NON_EVENT_RATE"] = d3.NONEVENT/d3.COUNT
    d3["DIST_EVENT"] = d3.EVENT/d3.sum().EVENT
    d3["DIST_NON_EVENT"] = d3.NONEVENT/d3.sum().NONEVENT
    print(np.log(d3.DIST_EVENT/d3.DIST_NON_EVENT))
    d3["WOE"] = np.log(d3.DIST_EVENT/d3.DIST_NON_EVENT)
    d3["IV"] = (d3.DIST_EVENT-d3.DIST_NON_EVENT)*np.log(d3.DIST_EVENT/d3.DIST_NON_EVENT)
    d3["VAR_NAME"] = "VAR"
    d3 = d3[['VAR_NAME','MIN_VALUE', 'MAX_VALUE', 'COUNT', 'EVENT', 'EVENT_RATE', 'NONEVENT', 'NON_EVENT_RATE', 'DIST_E
    d3 = d3.replace([np.inf, -np.inf], 0)
    d3.IV = d3.IV.sum()

    return(d3)
```

```
def char_bin(Y, X):

    df1 = pd.DataFrame({"X": X, "Y": Y})
    justmiss = df1[['X','Y']][df1.X.isnull()]
    notmiss = df1[['X','Y']][df1.X.notnull()]
    df2 = notmiss.groupby('X',as_index=True)
    d3 = pd.DataFrame({},index=[])
    d3["COUNT"] = df2.count().Y
    d3["MIN_VALUE"] = df2.sum().Y.index
    d3["MAX_VALUE"] = d3["MIN_VALUE"]
    d3["EVENT"] = df2.sum().Y
    d3["NONEVENT"] = df2.count().Y - df2.sum().Y

    if len(justmiss.index) > 0:
        d4 = pd.DataFrame({'MIN_VALUE':np.nan},index=[0])
        d4["MAX_VALUE"] = np.nan
        d4["COUNT"] = justmiss.count().Y
        d4["EVENT"] = justmiss.sum().Y
        d4["NONEVENT"] = justmiss.count().Y - justmiss.sum().Y
        d3 = d3.append(d4,ignore_index=True)

    d3["EVENT_RATE"] = d3.EVENT/d3.COUNT
    d3["NON_EVENT_RATE"] = d3.NONEVENT/d3.COUNT
    d3["DIST_EVENT"] = d3.EVENT/d3.sum().EVENT
    d3["DIST_NON_EVENT"] = d3.NONEVENT/d3.sum().NONEVENT
    d3["WOE"] = np.log(d3.DIST_EVENT/d3.DIST_NON_EVENT)
    d3["IV"] = (d3.DIST_EVENT-d3.DIST_NON_EVENT)*np.log(d3.DIST_EVENT/d3.DIST_NON_EVENT)
    d3["VAR_NAME"] = "VAR"
    d3 = d3[['VAR_NAME','MIN_VALUE', 'MAX_VALUE', 'COUNT', 'EVENT', 'EVENT_RATE', 'NONEVENT', 'NON_EVENT_RATE', 'DIST
    d3 = d3.replace([np.inf, -np.inf], 0)
    d3.IV = d3.IV.sum()
    #print("hi",d3.IV )
    d3 = d3.reset_index(drop=True)

    return(d3)

def data_vars(df1, target):

    stack = traceback.extract_stack()
    filename, lineno, function_name, code = stack[-2]
    vars_name = re.compile(r'\((.*?)\).*$').search(code).groups()[0]
    final = (re.findall(r"[\w']+", vars_name))[-1]

    x = df1.dtypes.index
    count = -1
    for i in x:
        print(i)
        if i.upper() not in (final.upper()):
            if np.issubdtype(df1[i], np.number) and len(Series.unique(df1[i])) > 2:
                #print("Number and unique value greater than 2")
                conv = mono_bin(target, df1[i])
                conv["VAR_NAME"] = i
                count = count + 1
            else:
                #print("I am here 2")
                conv = char_bin(target, df1[i])
                conv["VAR_NAME"] = i
                count = count + 1

            if count == 0:
                iv_df = conv
            else:
                iv_df = iv_df.append(conv,ignore_index=True)

    iv = pd.DataFrame({'IV':iv_df.groupby('VAR_NAME').IV.max()})
    iv = iv.reset_index()
    return(iv_df,iv)
```

```
)]: import traceback
    import sys
    final_iv, IV = data_vars(X_train, y_train)
```

```
    radius_mean
    0        -inf
    1   -3.378314
    2   -2.643494
    3   -1.478924
    4   -0.458354
    5    1.114280
    6    2.710731
    7         inf
    dtype: float64
    texture_mean
    0   -2.238029
    1   -0.681733
    2    0.614603
    3    1.267278
    dtype: float64
    perimeter_mean
    0        -inf
    1   -3.378314
    2   -3.357694
```

```
#features = list(IV[(IV['IV'] >= 0.01) & (IV['IV'] <= 0.8)]['VAR_NAME'])
features = list(IV[(IV['IV'] >= 0.01) & (IV['IV'] <= 0.8)]['VAR_NAME'])
X2 = X_train[features]
display(X2.shape)
X2.head()
```

```
(398, 7)
```

	compactness_se	fractal_dimension_se	fractal_dimension_worst	smoothness_mean	smoothness_se	symmetry_mean	symmetry_se
249	-0.737638	-0.382376	-0.324505	0.351537	-0.126331	0.260609	-0.456744
58	-0.943703	-0.304456	-1.166825	-1.121587	0.150013	0.026949	0.164388
476	0.413687	-0.397506	-0.297351	-0.501736	-0.978033	-1.115796	-0.537867
529	-0.779555	-0.590414	-0.189844	0.970677	0.095344	-0.564504	-0.111671
422	-0.303376	-0.685733	-0.536193	0.885278	-0.385675	0.271562	-0.249700

```
display(X2.shape[1])
for i in range(X2.shape[1]):
    print((i, variance_inflation_factor(X2.values, i) ))
    #print(variance_inflation_factor(X2.values, i) )
```

```
7

(0, 3.0868696340477566)
(1, 3.4735372472130357)
(2, 2.3391861484700094)
(3, 1.9170592081300493)
(4, 1.6671758939247243)
(5, 1.9261698275387005)
(6, 1.7812083391918967)
```

```
def iterate_vif(df, vif_threshold=5, max_vif=6):
    count = 0
    while max_vif > vif_threshold:
        count += 1
        print("Iteration # "+str(count))
        vif = pd.DataFrame()
        vif["VIFactor"] = [variance_inflation_factor(df.values, i) for i in range(df.shape[1])]
        vif["features"] = df.columns

        if vif['VIFactor'].max() > vif_threshold:
            print('Removing %s with VIF of %f' % (vif[vif['VIFactor'] == vif['VIFactor'].max()]['features'].values[0],
            df = df.drop(vif[vif['VIFactor'] == vif['VIFactor'].max()]['features'].values[0], axis=1)
            max_vif = vif['VIFactor'].max()
        else:
            print('Complete')
            return df, vif.sort_values('VIFactor')

X1 = X2._get_numeric_data()
final_df, final_vif = iterate_vif(X1)
```

```
Iteration # 1
Complete
```

```
X_train=final_df
display(len(X_train.columns))
display(X_train.columns)
X_train.head()
```

```
7

Index(['compactness_se', 'fractal_dimension_se', 'fractal_dimension_worst',
       'smoothness_mean', 'smoothness_se', 'symmetry_mean', 'symmetry_se'],
      dtype='object')
```

	compactness_se	fractal_dimension_se	fractal_dimension_worst	smoothness_mean	smoothness_se	symmetry_mean	symmetry_se
249	-0.737638	-0.382376	-0.324505	0.351537	-0.126331	0.260609	-0.456744
58	-0.943703	-0.304456	-1.166825	-1.121587	0.150013	0.026949	0.164388
476	0.413687	-0.397506	-0.297351	-0.501736	-0.978033	-1.115796	-0.537867
529	-0.779555	-0.590414	-0.189844	0.970677	0.095344	-0.564504	-0.111671
422	-0.303376	-0.685733	-0.536193	0.885278	-0.385675	0.271562	-0.249700

5. Model

I use Scikit-Learn for my prediction models. Data are partitioned into train and test sets with a size ratio of 7/3. The following models will be built and compared:

A. Random Forest Classifier

Like its name implies consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes become our model's prediction. As we know in tree-based models do not need to scale the features nor encoding, but these are already engineered and in order to compare the performance of models under certain context the random forest will be trained in the same way as the prior models.
It is common practice to use a grid search method for all adjustable parameters in any type of machine learning algorithm. Grid search is used to find the best hyperparameters for the model. Grid search is defined with 6 random of trees from 100 to 500. Max levels in each decision tree are randomly range from 4 to 16, and auto, square root and log2 are features to consider at every split. Gini and entropy criteria also included. The best model parameters are selected with accuracy score of 79. The model again is trained with selected parameters.

Find the best paramaters using GridSearchCV

```
#Grid Search for RainForest Classifier
parameters = {
    'n_estimators' : [100,150,200,300,400,500],
    'max_features' : ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,6,8,12,14,16],
    'criterion' : ['gini', 'entropy'],
    'n_jobs' : [-1,1,None]
}

clf_cv = GridSearchCV(estimator=RandomForestClassifier(), param_grid=parameters, cv= 5)

# Model and fit
model= clf_cv.fit(X_train, y_train)
y_pred_cv = model.predict(X_test)
```

```
#Best score and best parameter for n_neighbors
print("Best Score:" + str(clf_cv.best_score_))
print("Best Parameters: " + str(clf_cv.best_params_))
```

```
Best Score:0.7864240506329114
Best Parameters: {'criterion': 'gini', 'max_depth': 14, 'max_features': 'log2', 'n_estimators': 150, 'n_jobs': -1}
```

```
#Fit and score the best number of neighbors
RF_model = RandomForestClassifier(n_estimators=200, criterion="gini", max_depth =12, max_features ='log2',n_jobs=-1)
RF_model.fit(X_train, y_train)
predictions = RF_model.predict(X_test)
```

```
#Model evaluation
f1 = f1_score(y_test, predictions, average='weighted')
ac = accuracy_score(y_test, predictions, normalize=True)
recall = recall_score(y_test, predictions, average="weighted")
precision = precision_score(y_test, predictions, average="weighted")
roc1 = roc_auc_score(y_test, predictions)

print("********* Random Forest with GridSearch Results *********")
print('Random Forest: Accuracy = %.3f' % (ac))
print('Random Forest: F1 Score = %.3f' % (f1))
print('Random Forest: Recall = %.3f' % (recall))
print('Random Forest: Precision = %.3f' % (precision))
print('Random Forest: roc_auc = %.3f' % (roc1))
```

```
********* Random Forest with GridSearch Results *********
Random Forest: Accuracy = 0.789
Random Forest: F1 Score = 0.784
Random Forest: Recall = 0.789
Random Forest: Precision = 0.787
Random Forest: roc_auc = 0.754
```

To evaluate the performance of the model in detail, the confusion matrix and the classification reports are generated. The model predicted the presence of cancer with a probability of 0.6 (recall of 1) and the absence of cancer with 0.90.

Classification Report

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.90      0.84       108
           1       0.78      0.60      0.68        63

    accuracy                           0.79       171
   macro avg       0.79      0.75      0.76       171
weighted avg       0.79      0.79      0.78       171
```

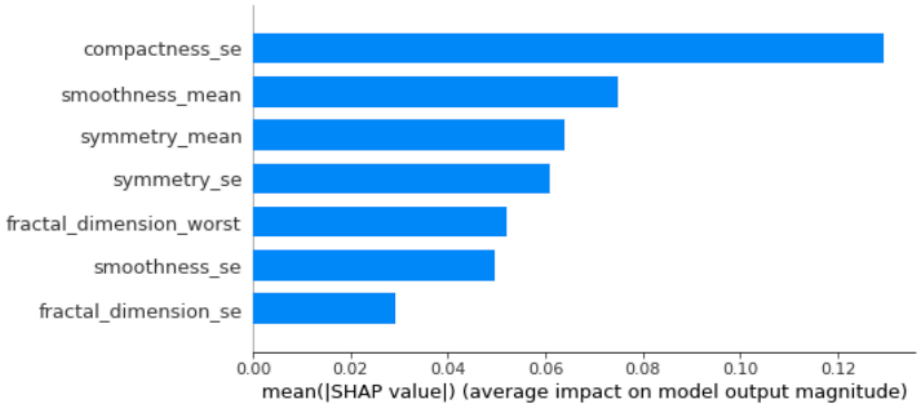
Confusion Matrix

```
print(confusion_matrix(y_test,y_pred))
```

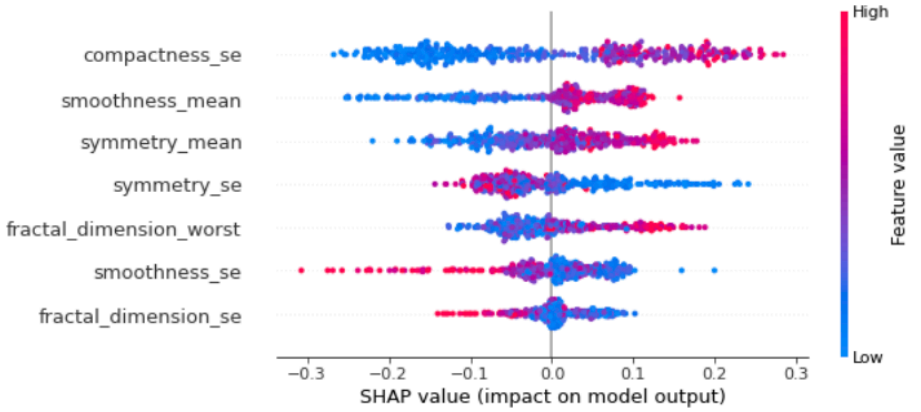
```
[[97 11]
 [25 38]]
```

The random forest algorithm can be used as a regression or classification model. In either case it tends to be a bit of a black box, where understanding what's happening under the hood can be difficult. Plotting the feature importance is one way that you can gain a perspective on which features are driving the model predictions. The importance of features and their impact on the model's output can be explained with shap values and their summary plots. Comparing the shap summary plot, compactness_se and smoothness_mean are the most important features affecting the classification decision or model output.

```
import shap
shap_values = shap.TreeExplainer(RF_model).shap_values(X_train)
shap.summary_plot(shap_values[1], X_train, plot_type="bar")
```




```
import shap
shap_values = shap.TreeExplainer(RF_model).shap_values(X_train)
shap.summary_plot(shap_values[1],X_train)
```



B. Logistic Regression

```
lr = LogisticRegression(max_iter=1000)
model_lr= lr.fit(X_train, y_train)
y_pred = model_lr.predict(X_test)

#Model evaluation
f1 = f1_score(y_test, y_pred, average='weighted')
ac = accuracy_score(y_test, y_pred, normalize=True)
recall = recall_score(y_test, y_pred, average="weighted")
precision = precision_score(y_test, y_pred, average="weighted")
roc2 = roc_auc_score(y_test, y_pred)

print("********* Linear Regression Results *********")
print('Linear Regression: Accuracy = %.3f' % (ac))
print('Linear Regression: F1 Score = %.3f' % (f1))
print('Linear Regression: Recall = %.3f' % (recall))
print('Linear Regression: Precision = %.3f' % (precision))
print('Linear Regression: roc_auc = %.3f' % (roc2))
```

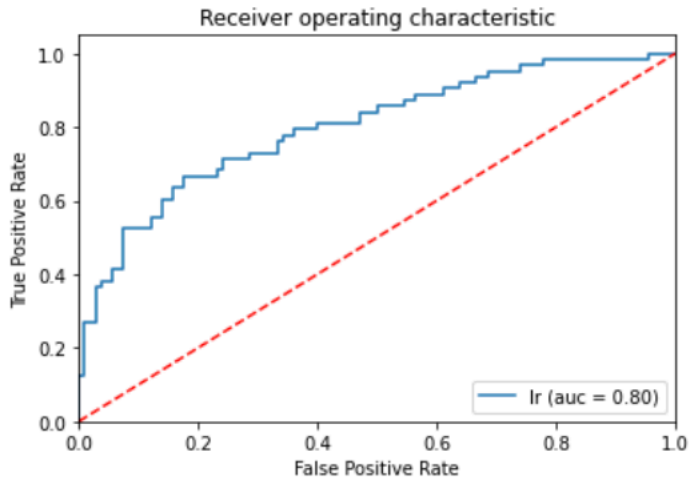
```
********* Linear Regression Results *********
Linear Regression: Accuracy = 0.754
Linear Regression: F1 Score = 0.748
Linear Regression: Recall = 0.754
Linear Regression: Precision = 0.750
Linear Regression: roc_auc = 0.716
```

Classification Report

```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.78      0.86      0.82       108
           1       0.71      0.57      0.63        63

    accuracy                           0.75       171
   macro avg       0.74      0.72      0.72       171
weighted avg       0.75      0.75      0.75       171
```

AUC-ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. The model has AUC = 0.8 > 0.5. It is perfectly able to distinguish between classes.



C. Support Vector Machine Classifier

SVM depends on supervised learning models and trained by learning algorithms. A SVM generates parallel partitions by generating two parallel lines. For each category of data in a high-dimensional space and uses almost all attributes. It separates the space in a single pass to generate flat and linear partitions. Divide the 2 categories by a clear gap that should be as wide as possible. Do this partitioning by a plane called hyperplane. An SVM creates hyperplanes that have the largest margin in a high-dimensional space to separate given data into classes. The margin between the 2 classes represents the longest distance between closest data points of those classes.

```
from sklearn.svm import SVC
svc = SVC()
svc.fit(X_train,y_train)
y_pred_svc = svc.predict(X_test)

#Model evaluation
f1 = f1_score(y_test, y_pred_svc, average='weighted')
ac = accuracy_score(y_test, y_pred_svc, normalize=True)
recall = recall_score(y_test, y_pred_svc, average="weighted")
precision = precision_score(y_test, y_pred_svc, average="weighted")
roc3 = roc_auc_score(y_test, y_pred_svc)

print("********* SVC Results *********")
print('SVC : Accuracy = %.3f' % (ac))
print('SVC: F1 Score = %.3f' % (f1))
print('SVC: Recall = %.3f' % (recall))
print('SVC: Precision = %.3f' % (precision))
print('SVC: roc_auc = %.3f' % (roc3))
```

```
********* SVC Results *********
SVC : Accuracy = 0.772
SVC: F1 Score = 0.757
SVC: Recall = 0.772
SVC: Precision = 0.777
SVC: roc_auc = 0.717
```

Classification Report

```
print(classification_report(y_test, y_pred_svc))
```

```
              precision    recall  f1-score   support

           0       0.76      0.93      0.84       108
           1       0.80      0.51      0.62        63

    accuracy                           0.77       171
   macro avg       0.78      0.72      0.73       171
weighted avg       0.78      0.77      0.76       171
```

To improve the performance of the support vector machine we will need to select the best parameters for the model. The standard way of doing it is by doing a grid search.

Tuning the hyper_parameters

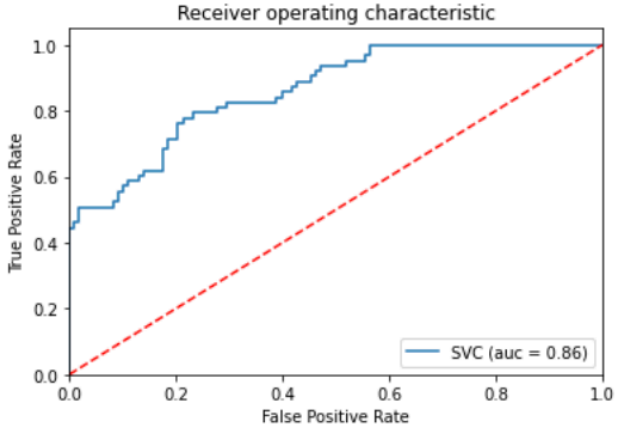
```
#GridSearchCV helps us combine an estimator with a grid search preamble to tune hyper_parameters
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.1,1,10,100],
              'gamma': [1, 0.1, 0.01, 0.001],
              'kernel': ['rbf','sigmoid','poly']
             }
grid_svc = GridSearchCV(svc, param_grid, refit=True, verbose=2)
grid_svc.fit(X_train, y_train)

#Best score and best parameter
print("Best Score:" + str(grid_svc.best_estimator_))
print("Best Parameters: " + str(grid_svc.best_params_))
```

```
[CV] ................. C=0.1, gamma=0.1, kernel=sigmoid, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=sigmoid .............................
[CV] ................. C=0.1, gamma=0.1, kernel=sigmoid, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=sigmoid .............................
[CV] ................. C=0.1, gamma=0.1, kernel=sigmoid, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=poly ................................
[CV] ................... C=0.1, gamma=0.1, kernel=poly, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=poly ................................
[CV] ................... C=0.1, gamma=0.1, kernel=poly, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=poly ................................

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s

[CV] ................... C=0.1, gamma=0.1, kernel=poly, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=poly ................................
[CV] ................... C=0.1, gamma=0.1, kernel=poly, total=   0.0s
[CV] C=0.1, gamma=0.1, kernel=poly ................................
[CV] ................... C=0.1, gamma=0.1, kernel=poly, total=   0.0s
[CV] C=0.1, gamma=0.01, kernel=rbf ................................
[CV] ................... C=0.1, gamma=0.01, kernel=rbf, total=   0.0s
```

```
grid_svc_pred = grid_svc.predict(X_test)

#Model evaluation
f1 = f1_score(y_test, grid_svc_pred, average='weighted')
ac = accuracy_score(y_test, grid_svc_pred, normalize=True)
recall = recall_score(y_test, grid_svc_pred, average="weighted")
precision = precision_score(y_test, grid_svc_pred, average="weighted")
roc3 = roc_auc_score(y_test, grid_svc_pred)

print("********* SVC Results *********")
print('SVC : Accuracy = %.3f' % (ac))
print('SVC: F1 Score = %.3f' % (f1))
print('SVC: Recall = %.3f' % (recall))
print('SVC: Precision = %.3f' % (precision))
print('SVC: roc_auc = %.3f' % (roc3))
```

```
********* SVC Results *********
SVC : Accuracy = 0.778
SVC: F1 Score = 0.768
SVC: Recall = 0.778
SVC: Precision = 0.778
SVC: roc_auc = 0.731
```

This model has AUC = 0.86 greater than logistic regression model. The cost of incorrectly classifying the presence of the cancer is riskier than incorrectly classifying the absence of cancer. This implies that magnitude of the metric RECALL should be maximum.



D. KNeighbors Classifier

Knn is essentially classification by finding the most similar data points in the training data, and making an educated guess based on their classifications. K is number of nearest neighbors that the classifier will use to make its prediction. KNN makes predictions based on the outcome of the K neighbors closest to that point. One of the most popular choices to measure this distance is known as Euclidean.

To determine the value of k, we will apply KNN algorithm to the training dataset across different values of k from 1 to 20:

```
from sklearn.neighbors import KNeighborsClassifier

test_scores = []
train_scores = []

for i in range(1,20):

    knn = KNeighborsClassifier(i)
    knn.fit(X_train,y_train)

    train_scores.append(knn.score(X_train,y_train))
    test_scores.append(knn.score(X_test,y_test))

print(train_scores)
print(test_scores)
```

```
[1.0, 0.8743718592964824, 0.8693467336683417, 0.8442211055276382, 0.821608040201005, 0.821608040201005, 0.80150753768
84422, 0.7964824120603015, 0.7914572864321608, 0.7914572864321608, 0.7889447236180904, 0.7939698492462312, 0.78894472
36180904, 0.7864321608040201, 0.7864321608040201, 0.7989949748743719, 0.7763819095477387, 0.7964824120603015, 0.77135
67839195979]
[0.7017543859649122, 0.7192982456140351, 0.7076023391812866, 0.7719298245614035, 0.7660818713450293, 0.78362573099415
2, 0.7953216374269005, 0.7719298245614035, 0.7953216374269005, 0.7660818713450293, 0.7660818713450293, 0.760233918128
6549, 0.7543859649122807, 0.7426900584795322, 0.7719298245614035, 0.7485380116959064, 0.7485380116959064, 0.771929824
5614035, 0.7660818713450293]
```

Identify the number of neighbors that resulted in the max score in the training dataset

```
print("Number of neighbors that resulted max score in the training set: {}".format(train_scores.index(max(train_scores)
```

```
Number of neighbors that resulted max score in the training set: 1
```

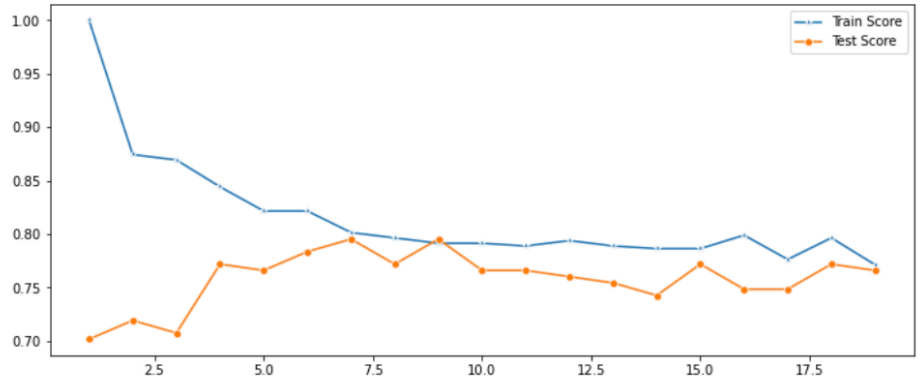
Identify the number of neighbors that resulted in the max score in the testing dataset

```
print("Number of neighbors that resulted max score in the testing set: {}".format(test_scores.index(max(test_scores))+1
```

```
Number of neighbors that resulted max score in the testing set: 7
```

Plot the train and test model performance by number of neighbors

```
plt.figure(figsize=(12,5))
p = sns.lineplot(range(1,20),train_scores,marker='*',label='Train Score')
p = sns.lineplot(range(1,20),test_scores,marker='o',label='Test Score')
```



It is underfitting because training accuracy decreases and testing accuracy seems decreasing or flat

With the optimal number of neighbors 7, we fitted and trained the model again. The accuracy of the model is 0.795

```
]: knn = KNeighborsClassifier(n_neighbors=7)
   knn.fit(X_train,y_train)
   y_pred_knn = knn.predict(X_test)

   #Model evaluation
   f1 = f1_score(y_test, y_pred_knn, average='weighted')
   ac = accuracy_score(y_test, y_pred_knn, normalize=True)
   recall = recall_score(y_test, y_pred_knn, average="weighted")
   precision = precision_score(y_test, y_pred_knn, average="weighted")
   roc4 = roc_auc_score(y_test, y_pred_knn)

   print("********* KNeighborsClassifier Results *********")
   print('KNN: Accuracy = %.3f' % (ac))
   print('KNN: F1 Score = %.3f' % (f1))
   print('KNN: Recall = %.3f' % (recall))
   print('KNN: Precision = %.3f' % (precision))
   print('KNN: roc_auc = %.3f' % (roc4))
```

```
********* KNeighborsClassifier Results *********
KNN: Accuracy = 0.795
KNN: F1 Score = 0.789
KNN: Recall = 0.795
KNN: Precision = 0.794
KNN: roc_auc = 0.759
```

Classification Report

```
: print(classification_report(y_test, y_pred_knn))
```

	precision	recall	f1-score	support
0	0.80	0.90	0.85	108
1	0.78	0.62	0.69	63
accuracy			0.80	171
macro avg	0.79	0.76	0.77	171
weighted avg	0.79	0.80	0.79	171

Conclusion

There is more cost associated with the improper classification of the presence of malignant. As a result, the model predicting malignant should be very accurate in properly classifying the presence of malignant. As we compare all the models above, KNeighbors Classifier model perform the best with an accuracy of 80% and especially the recall is 80% highest among the models. “Compactness_se” and “smoothness_mean” are the most important features affecting the classification decision or model output. The healthcare community can perform additional research corresponding to these attributes to help prevent pervasion of breast cancer.

According to the cancer priority as one of the important health issues in the world that imposes mortality and costs, there is an urgent need for survival prediction strategies. One of the main goals in cancer patients is the estimation of survival. It leads to better management, optimal uses of resources and providing individualized treatment. Using machine learning techniques can help to prevent potential errors in survival estimation, provide appropriate and individualized treatments to patients and improve the prognosis of cancers.