Student ID: 8655295

Module: 6006CEM Machine Learning

Course: Computer Science

Table of Contents

Introduction

Breast Cancer is one of the most common cancers in woman. According to Cancer Research UK, 30% of new female cancer cases belong to breast cancer and is the leading cause of cancer death in women under 50, in the UK, making it a prevalent health issue.

Breast Cancer can come in two forms: benign or malignant. The early diagnosis of the cancer can improve the chance of survival by giving patients more time to undergo cancer treatment. If the type of breast cancer can be predicted and classified early on, it reduces the chance of misdiagnosis and can prevent patients having to go through unnecessary treatments.

The constant evolution of ML learning algorithms has been crucial for the prediction of breast cancer diagnosis, which has been decreasing the deathly rate of people that carry this cancer. Feature extraction and classification algorithms employed in it led to the design of an efficient system (S. P. Rajamohana, 2019).

The objective with this report is critically analysing the dataset above and testing the accuracy of Logistic Regression, K-Neighbors Classifier and Random Forest Classifier when it comes to the diagnosis prediction, malignant or benign.

Dataset description

The dataset used, "**Breast Cancer Wisconsin (Diagnostic) Data Set"**, was taken from UCI Machine Learning repository. It was created by Dr. William H. Wolberg, physician at the University of Wisconsin.

The dataset contains the following variables:

Attribute Information:

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

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

3) radius
4) texture
5) perimeter
6) area
7) smoothness
8) compactness
9) concavity
10) concave points
11) symmetry
12) fractal dimension

All of these values contain a "mean", a "worst" and a "standard" value, so we have a total of 32 variables to work with.

Data Analysis

We start by running the command head() to observe our dataset:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	te
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	

5 rows × 33 columns

Figure 1 - data.head() command

Next, using shape(), we could see our dataset contains 569 rows with 33 columns.

Data Pre-processing

To test our classification models, pre-processing of the data has to be made, for it to be in a correct format for the testing and evaluation of the models.

Handling missing values

First, we verified if any of the dataset values are null using the command info():

```
In [4]: 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
```

Figure 2 - Data.info() command

As we can observe, column “Unnamed:32” is null so we proceed to drop it since it is not relevant for our case study. Column “id” was also dropped.

Then, we verify if we missed any null values, with “data.isna().sum()” :

```
In [7]: #Checks if variables have any null values
        data.isna().sum()

Out[7]: diagnosis                0
        radius_mean              0
        texture_mean             0
        perimeter_mean           0
        area_mean                0
        smoothness_mean          0
        compactness_mean         0
        concavity_mean           0
        concave points_mean      0
        symmetry_mean            0
        fractal_dimension_mean   0
        radius_se                0
        texture_se               0
        perimeter_se             0
        area_se                  0
        smoothness_se            0
        compactness_se           0
        concavity_se             0
        concave points_se        0
        symmetry_se              0
        fractal_dimension_se     0
        radius_worst             0
        texture_worst            0
        perimeter_worst          0
        area_worst               0
        smoothness_worst         0
        compactness_worst        0
        concavity_worst          0
        concave points_worst     0
        symmetry_worst           0
        fractal_dimension_worst  0
        dtype: int64
```

Figure 3 - data.isna().sum() command

As per observation of the image above, there are no null values so we can proceed with the pre-processing of our data.

Handling categorical values

The next step would be to handle categorical features. Categorical variables contain label values instead of numeric values, so we need to transform them for it to be used with the models.

Looking at our head() command from early, we can see that the ‘diagnosis’ column contains letters instead of numbers. So we run the unique() command on our ‘diagnosis’ column:

```
#Prepare data for prediction
data.diagnosis.unique()
```

```
array(['M', 'B'], dtype=object)
```

Figure 4 - unique() command

As we can observe, our ‘diagnosis’ column is type object, instead of float like the other columns so we need to transform that ‘M’ and ‘B’ to numbers by mapping ‘M’ to be the number ‘1’ and ‘B’ to be number ‘0’.

We ran our head() command again and it no longer has letters in that column.

```
In [9]: # We transform our diagnosis into int for the prediction to be accurate
        #Handle non - categorical values
        data['diagnosis'] = data['diagnosis'].map({'M':1,'B':0})
        data.head()
```

Out[9]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
0	1	17.99	10.38	122.80	1001.0	0.11840	0.27760
1	1	20.57	17.77	132.90	1326.0	0.08474	0.07864
2	1	19.69	21.25	130.00	1203.0	0.10960	0.15990
3	1	11.42	20.38	77.58	386.1	0.14250	0.28390
4	1	20.29	14.34	135.10	1297.0	0.10030	0.13280

Figure 5 - data.head() to verify categorical change

A barchart was plotted to observe the difference in quantity of Malignant and Benign values:

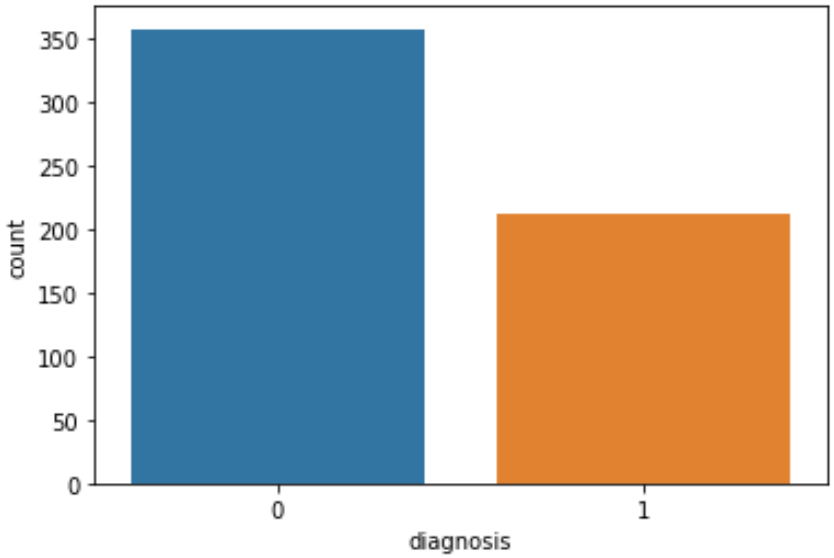

Figure 6 - Barchart

As per observation of Figure 6, there are significantly more Benign cases than Malignant in the dataset.

Feature Selection

We now proceed with selecting the most relevant features from our data. “The aim of feature selection is to inform decisions about which measures (“features” in machine learning parlance) in the data set should be included in data collection or analysis to optimally predict the outcomes with a minimum number of predictors” (R Brick, Timothy, 2017). Some features become less relevant or are too similar to other, so if we reduce the number of features, we accomplish a more accurate result.

There are several methods for feature selection, however the feature selection will be made based on correlation with each other.

First, the data is transformed into a correlation format, where every feature is compared with each other:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
diagnosis	1.000000	0.730029	0.415185	0.742636	0.708984
radius_mean	0.730029	1.000000	0.323782	0.997855	0.987357
texture_mean	0.415185	0.323782	1.000000	0.329533	0.321086
perimeter_mean	0.742636	0.997855	0.329533	1.000000	0.986507
area_mean	0.708984	0.987357	0.321086	0.986507	1.000000

Figure 7 - correlated data

Looking at the values, it's not really obvious which are more correlated to each other, so we plot our correlated data into a heatmap:

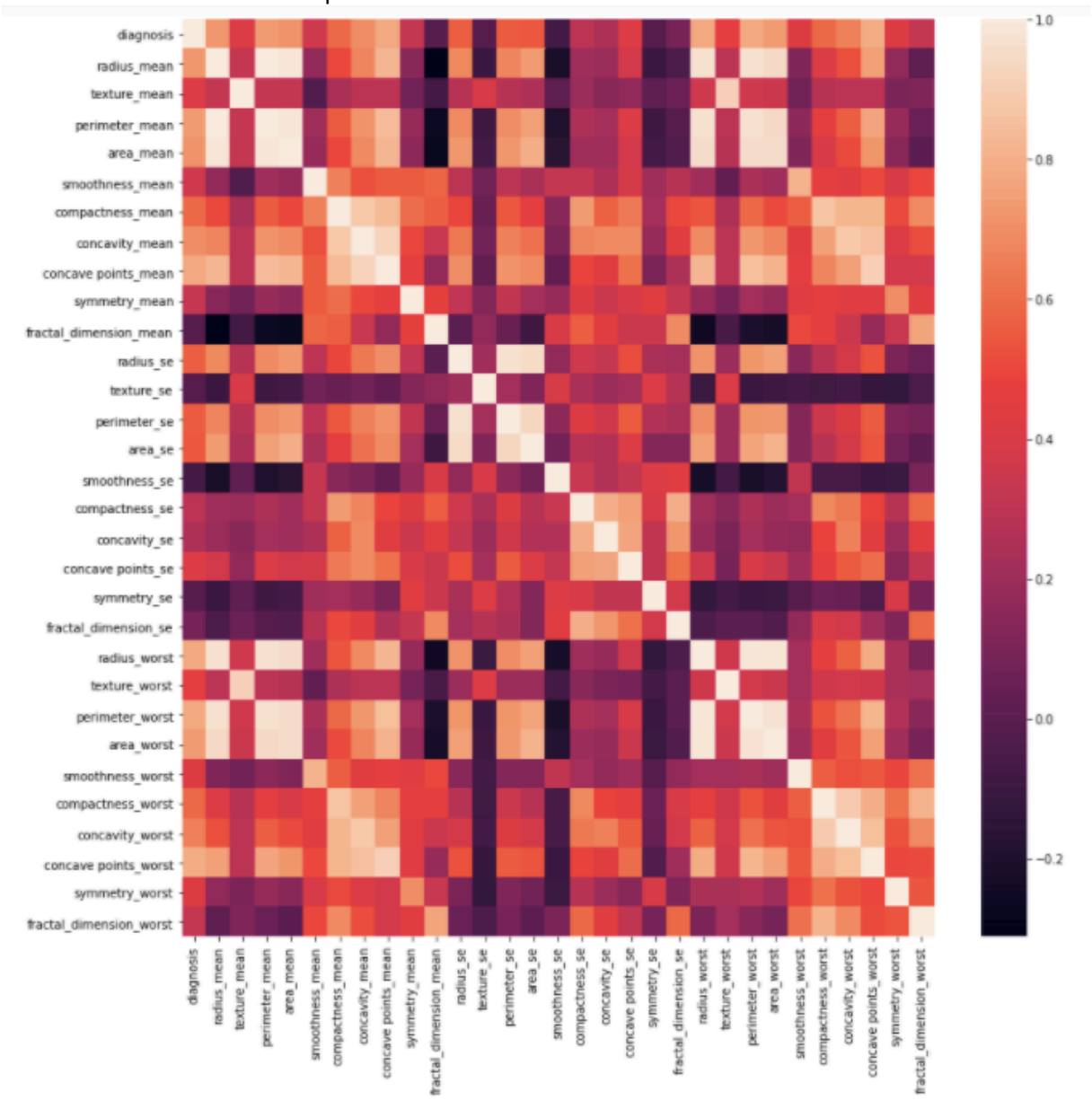

Figure 8 - heatmap

If two or more features have high correlation, it's only worth keeping one of them, since they are equally relevant to the dataset.

As we can observe, perimeter, area and radius are highly correlated, in all their stages (se, mean and worst) so we will be keeping only the 'radius_se', 'radius_mean' and 'radius_worst', dropping 'perimeter_mean', 'perimeter_worst', 'perimeter_se', 'area_worst', 'area_se', 'area_mean'. The data now consists of 24 features instead of 30.

To test if feature selection increases our accuracy, we are going to work with two separate dataframes for our model implementation: 'data', which contains the 31 initially already cleaned values and 'data_tuned', 'data' that suffered feature selection and contains 25 columns.

Scale and Split data

The next step was scaling the data. Our data contains features that variate on unit, magnitude and range. Scaling helps bring all the features to the same magnitude, for the learning models to understand better and give more accurate results.

For scaling, the function "StandardScaler()" was used to scale our "X" values, that correspond to all the features except our target, "diagnosis", that corresponds to our "Y".

Splitting of data was also made. When splitting the data, we end up with two sets of data: a training set and a testing set. The training set of data is going to be what our model is going to learn from and then the model is going to predict the "diagnosis" on the testing set.

To split the data, train_test_split() was used, where the training data size corresponds to 70% and the testing data size corresponds to 30% of the data.

With our data ready to be processed, a Principal Component Analysis (PCA) was plotted in order to summarize all of our information in a graph:

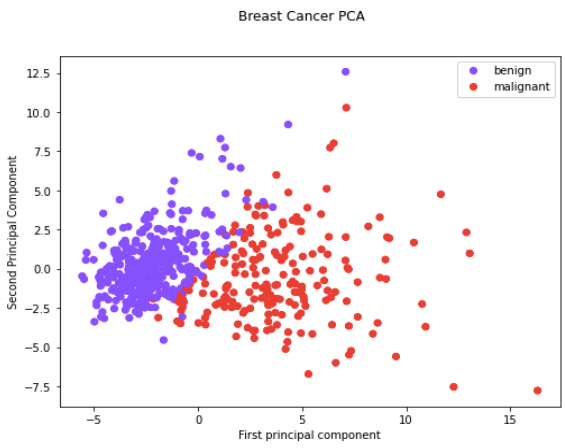

Figure 9 - PCA

As we can see in the figure above, the "Benign" cases represent less variance than the "Malignant" cases, in both components.

Implementation

With the data ready to be processed, we can implement our learning models: Logistic Regression, K-Neighbors Classifier and Random Forest Classifier. All the models were tested with two datasets: our cleaned data (data) and our data that suffered feature selection (data_tuned).

For all the models, we first selected 'X' and 'Y' variables and we split the data into a 70/30 percentage, where 70% corresponds to the training set and 30% corresponds to the testing set. We start by fitting the training variables into the model and then we predict the results with our predict() function:

```
#Training the model
model = LogisticRegression()
model.fit(X_train, y_train)
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
```

Figure 10 - Training Logistic Regression

For all the models, Cross Validation was implemented with the 'cross_val_score' function. Since the models tend to go into overfitting, this is a regularisation technique that helps avoiding that happening. It works by testing our data into new subsets of our data instead of only our training set, to see how it would perform outside of our sample data:

```
#Cross Validation
scores = cross_val_score(model, X_test, y_test, cv=5, scoring='accuracy')
```

Figure 11 - Cross Validation

The function takes the model, the testing variables, a splitting strategy equal to 5 folds (subsets of data) and an estimator for accuracy. The function tests the model in the 5 folds and then stores the values in "scores".

For each model, a confusion matrix is plotted. This matrix helps with the visualisation of the comparison between predicted values and actual values, quantifying the performance of a machine learning algorithm.

For the models that take "data_tuned" i.e data with feature selection, there were more processes done to improve the accuracy of the model. Hyperparameter tunning was performed in the form of a Random Search and Grid Search. Hyperparameters control the behaviour of a machine learning model and "the ultimate goal is to find an optimal combination of hyperparameters that minimizes a predefined loss function to give better results." (Lee, Admond, 2019).

For the Logistic Regression, RandomizedSearch was implemented:

```
#RANDOM SEARCH CV
dual=[True,False]
#Defines parameters we want to use
max_iter=[100,110,120,130,140]
C = [1.0,1.5,2.0,2.5]
param_grid = dict(dual=dual,max_iter=max_iter,C=C)
random_model = RandomizedSearchCV(model, param_grid, random_state=1, cv=5, verbose=0, n_jobs=-1)
#Fits Model
random_model.fit(X_train, y_train)
y_train_pred = random_model.predict(X_train)
y_test_pred = random_model.predict(X_test)
```

Figure 12 - RandomizedSearchCV

For the K-Neighbors Classifier and Random Forest classifier, GridSearch was implemented:

```
#GRID SEARCH
#Defining parameters
knn_para = dict(n_neighbors=list(range(1,31)))
grid_model = GridSearchCV(model, knn_para, cv=5, scoring='accuracy')
grid_model.fit(X_train, y_train)
y_train_pred = grid_model.predict(X_train)
y_test_pred = grid_model.predict(X_test)
```

```
#GRID SEARCH
#Defining parameters
forest_para = {
    'n_estimators': [50, 150, 250],
    'max_features': ['sqrt', 0.25, 0.5, 0.75, 1.0],
    'min_samples_split': [2, 4, 6]
}
grid_model = GridSearchCV(model, forest_para, cv=5, scoring='accuracy', n_jobs=-1)
grid_model.fit(X_train, y_train)
y_train_pred = grid_model.predict(X_train)
y_test_pred = grid_model.predict(X_test)
```

Figure 13, 14 – GridSearchCV

GridSearch iterates through every combination of parameters possible and stores a model for each of these. It's more accurate, but it's takes more time to process. RandomizedSearch is quicker to process but can be less accurate since it uses random combinations, with the possibility of the most optimal not being there.

Results, Evaluations and Conclusions

For each model, we evaluated the training accuracy, the testing accuracy and the overall model performance in our data set. Cross Validation Accuracy was also evaluated, and, for better visualisation, a confusion matrix was plotted for each model.

Logistic Regression

As we can see in the figure below, the logistic regression model tested with 'data' achieved a 99% training accuracy, a 98% testing accuracy and an overall model accuracy and cross validation accuracy of 98%.

```
Logistic Regression:

Model training accuracy:  0.9899497487437185
Model testing accuracy:  0.9766081871345029
Model accuracy:  0.9766081871345029
Cross Validation Accuracy:  0.976638655462185
```

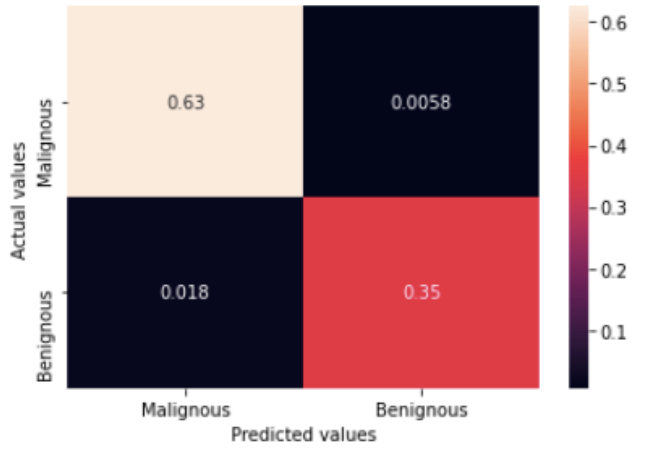

Figure 15 - Logistic Regression Results

Looking at the logistic regression tested with "data_tuned", with RandomSearch implemented, training accuracy was 99%, testing accuracy was 96% and overall accuracy was 96%. Our cross-validation score is also evaluated at 96%.

```
Logistic Regression With Tuned Data:

Model training accuracy:  0.9874371859296482
Model testing accuracy:  0.9649122807017544
Model accuracy:  0.9649122807017544
Cross Validation Accuracy:  0.9647058823529413
Best: 0.979968 using {'max_iter': 140, 'dual': False, 'C': 1.5}
```

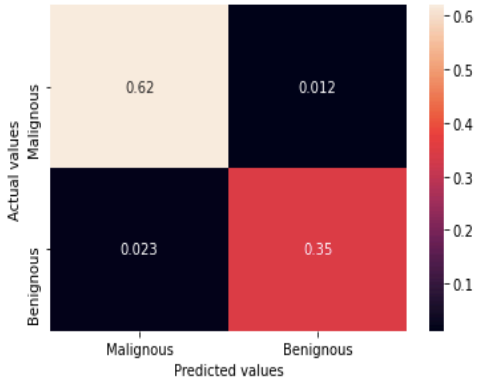

Figure 16 - Logistic Regression results with data_tuned

K-Neighbors Classifier

As we can see in the figure below, the KNN Classifier tested with our 'data' achieved a 97% training accuracy and a 96% testing accuracy. The model evaluation concluded with a model accuracy of 86% and cross-validation accuracy of 95%.

```
KNeigbours Classifier:

Model training accuracy:  0.9698492462311558
Model testing accuracy:  0.9590643274853801
Model accuracy:  0.9590643274853801
Cross Validation Accuracy:  0.953109243697479
```

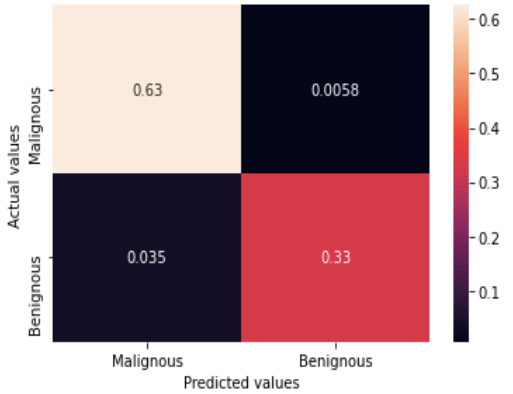

Figure 17 - KNN Classifier results

Looking at the KNN Classifier tested with "data_tuned", with GridSearch implemented, we have a training accuracy of 96%, a testing accuracy and model accuracy of 94% and a cross validation accuracy of 91%. GridSearch determined that a 95% accuracy is achieved with 23 neighbours used

```
KNeighbors Classifier With Tuned Data:

Model training accuracy:  0.957286432160804
Model testing accuracy:  0.9415204678362573
Model accuracy:  0.9415204678362573
Cross Validation Accuracy:  0.9058823529411765
Best: 0.954873 using {'n_neighbors': 23}
```

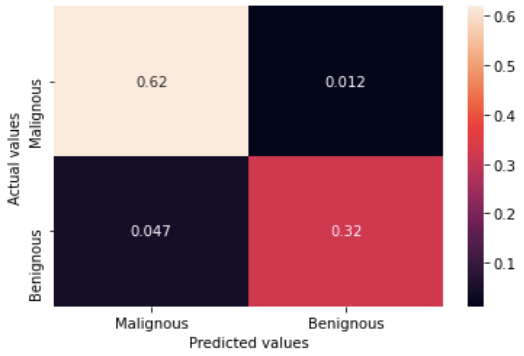

Figure 18 - KNN Classifier with tuned_data

Random Forest Classifier

Observing our Random Forest model, we achieved a training accuracy of 100%, a testing accuracy of 98%, an overall model accuracy of 98% and a cross validation accuracy of 95%.

```
Random Forest Classifier:

Model training accuracy:  1.0
Model testing accuracy:  0.9824561403508771
Model accuracy:  0.9824561403508771
Cross Validation Accuracy:  0.9534453781512605
```

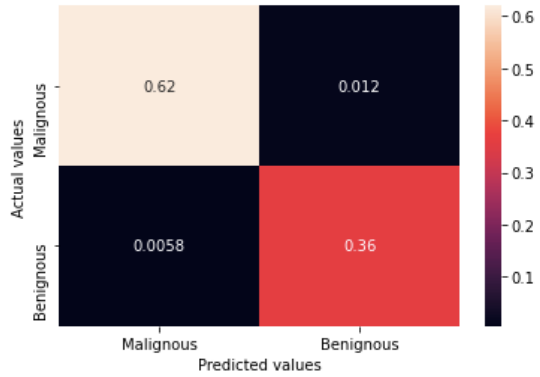

Figure 19 - Random Forest Classifier Results

Even without feature selection and hyperparameter tunning, this model performed the best so far compared to Logistic Regression and KNN Classifier. It is very important to know what model to

choose considering the dataset in question, and this was the best choice so far. The training accuracy being 100% shows how well this model learns and prepares to test the data.

The Random Forest model using 'data_tuned' and hyperparameter tunning with GridSearch also performed well. The model training accuracy is of 100%, the testing accuracy is 95%, the overall model accuracy is 95% and the cross-validation accuracy is a bit better, at 95.3%.

```
Random Forest Classifier With Tuned Data and Grid Search:

Model training accuracy:  1.0
Model testing accuracy:  0.9473684210526315
Model accuracy:  0.9473684210526315
Cross Validation Accuracy:  0.9532773109243697
Best: 0.954873 using {'max_features': 'sqrt', 'min_samples_split': 4, 'n_estimators': 50}
```

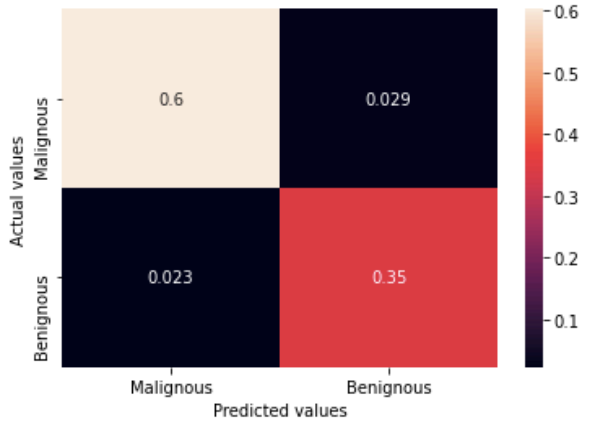

Figure 10 - Random Forest Classifier Results with data_tuned

Although the values are good, (with a training set still at 100%), the overall accuracy and training accuracy decreased.

Conclusion

The model that performed better using both our datasets was Random Forest Classifier and the worst was KNN Classifier.

Inside each model, the model that used a hyperparameter tunning method and was tested with the data that suffered feature selection, performed worse than the model that was tested with the normal dataset. This could be because they hyperparameter combination was not optimal or the features chosen in feature selection weren't the best, decreasing the accuracy of the model.

Nonetheless, all models present an accuracy above 90%, which shows to be a good thing if bias aren't considered.

References

- Rajamohana, S. P. (2019) 'Analysis of Classification Algorithms for Breast Cancer Prediction'. [online] available from <https://link.springer.com/chapter/10.1007/978-981-32-9949-8_36> [26 November 2020]
- Brick, Timothy R. (2017) 'Feature Selection Methods for Optimal Design of Studies for Developmental Inquiry'. *The Journals of Gerontology* [online] available from <https://academic.oup.com/psychsocgerontology/article/73/1/113/2970271> [27 November 2020]
- Lee, A. (2019) '' 'Why You Should Do Feature Engineering First, Hyperparameter Tuning Second as a Data Scientist'. *Medium* [online] available from https://towardsdatascience.com/why-you-should-do-feature-engineering-first-hyperparameter-tuning-second-as-a-data-scientist-334be5eb276c> [27 of November 2020]