Transitioning Machine Learning Algorithms

2023

AUGUST 13TH, 2023

Early Detection Inc.
Authored by: Josh Barber

Final Report

Abstract

Our organization's current approach to the detection of malignant breast cancer diagnoses, as informed by our utilization of William Wolberg's Breast Cancer dataset, exhibits limitations in success. The existing machine learning model, which relies upon linear regression techniques, achieves a coefficient of determination of 75%. Our rigorous exploration of linear regression variations, including Lasso and Ridge, failed to yield notable enhancements in outcomes. Evidently, linear regression methodologies prove suboptimal in comparison to more sophisticated classification models.

Upon transitioning to classification-oriented methodologies, significant advancements were observed. Specifically, a logistic regression model attained an impressive accuracy rate of 96%, while a random forest model similarly achieved a 96% accuracy rate. Given these remarkable outcomes, we advocate for a strategic shift in our approach towards the detection of malignant breast cancer diagnoses. Adopting classification models promises to be a pivotal step in advancing our corporate trajectory and enhancing our contribution to the field.

Where we are

Currently, our utilization revolves around a dataset that has noisy data. Within this dataset, a total of 699 entries are documented. Notably, certain entries exhibit missing data, while instances of duplications are also present. In its entirety, the dataset comprises of 11 columns, most of which contribute to the data's comprehensive representation. Acknowledging the pivotal significance of data cleanliness, we have diligently undertaken a series of measures to enhance the integrity of our results.

The necessity for data cleansing is underscored by several key reasons. Firstly, it promotes the elimination of inaccuracies stemming from missing values, thereby enabling a more comprehensive and accurate analysis. Furthermore, the elimination of duplicate entries ensures the quality of our dataset and prevents skewed results from influencing our outcomes. The extraction of irrelevant attributes is another integral aspect of data refinement, exemplified by the removal of the "Sample code number" column. This strategic omission is substantiated by a calculated chi-contingency test, affirming its insignificance within our analytical framework. It is worth noting that we have also applied a K-Nearest Neighbors (KNN) imputer to effectively address instances of missing values, enhancing the cohesiveness of our dataset.

Code snippet of original data cleansing:

```
import os
import pandas as pd
from sklearn.impute import KNNImputer
from scipy.stats import chi2_contingency

# Find contigency
def find_contigency(df: pd.DataFrame):
    columnDependenciesString = ""
```

```
            columns = df.columns
            for c in columns:
                if c != "Class":
                    contigency= pd.crosstab(df[c], df['Class'])
                    chi, p, dof, expected = chi2_contingency(contigency)
                    if p > 0.05:
                        columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" %
p)
                    else:
                        columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f"
% p)
            print("Find dependencies with current features:\n\n"+columnDependenciesString)

        col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell
Shape', 'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli',
'Mitosis', 'Class']
        df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)
        beforeCount = len(df)
        df = df[~df.duplicated()]
        afterCount = len(df)

        print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

        # Lets change the Class' column values from 2 and 4 to 0 - 1
        df["Class"] = df["Class"].replace(2, 0)
        df["Class"] = df["Class"].replace(4, 1)
        # Find feature importance
        find_contigency(df)

        # This has no significance to the data, time to drop
        df.drop("Sample code number", axis=1, inplace=True)
        knn_imp = KNNImputer(n_neighbors=10, weights='distance', metric='nan_euclidean')
        df["Bare Nuclei"] = knn_imp.fit_transform(df)



        df.head()
```

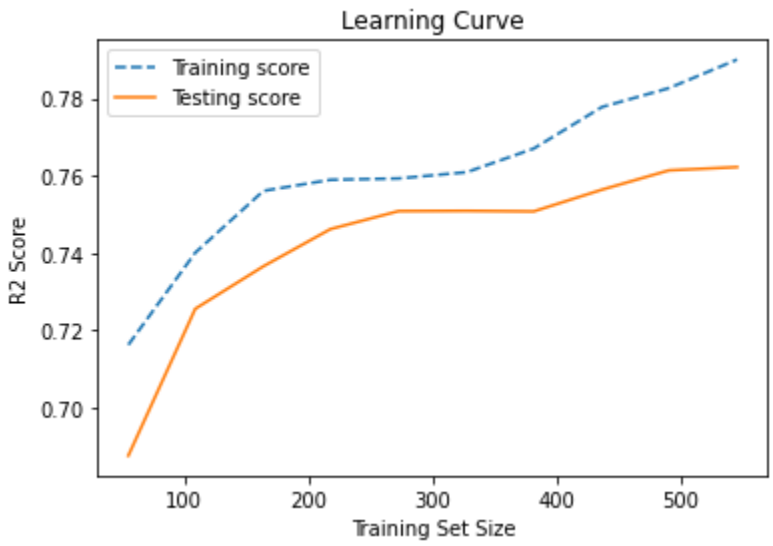

Figure 1. Training score vs Testing score

Illustrated in Figure 1 is a comparative evaluation of training and testing scores of our current linear model. The learning curve is a graphical representation that tracks the progress of our machine learning model's learning process. It serves as a visual depiction of how well the model is adapting and improving as it receives more exposure to the data.

The learning curve consists of two essential lines: the training score and the testing score. These lines offer insights into how the model's performance evolves during its learning journey. The training score, as showcased in the learning curve, reflects the model's proficiency in comprehending the training data. Initially, the training score is approximately 0.72, indicating that when the model began its learning process, its performance stood at around 72%. As the model continues to analyze and learn from the training data, its performance progressively improves, ultimately reaching a score of about 0.80. The testing score, depicted alongside the training score in the learning curve, signifies the model's success in making predictions on new and unseen data. At the outset, the testing score starts at roughly 0.68. This indicates that when the model encountered previously unobserved data, its accuracy stood at 68%. However, with each iteration of learning and refinement using the training data, the model becomes better equipped to handle fresh data. This enhancement is mirrored in the rise of the testing score to approximately 0.76.

Initially, our linear regression model exhibited a commendable coefficient of determination (R2 score) at 75% and a marginally elevated Mean Squared Error (MSE) of 0.06 on our test data. An R2 score of 75% indicates that approximately 75% of the variability observed in the dependent variable is explained by our linear regression model. This means the model's predictions capture a significant portion of the underlying patterns and trends in the data. In other words, the higher the R2 score, the better the model's ability to represent the observed data points. In our scenario, the marginally elevated MSE of 0.06 indicates that, on average, the squared differences between our linear regression model's predictions and the true data points are 0.06. The MSE serves as a metric to gauge the overall "fit" of the model's predictions to the actual data. Lower MSE values signify that the model's predictions are closer to the observed values, indicating higher predictive accuracy. Remarkably, scores retrieved through cross-validation seek the potential for refinement as

seen in Figure 2. The training score portrayed by cross-validation had a notable elevation to the R2 score of 85% and a substantially reduced MSE of 0.02, signifying an avenue for enhancing our original model's precision.

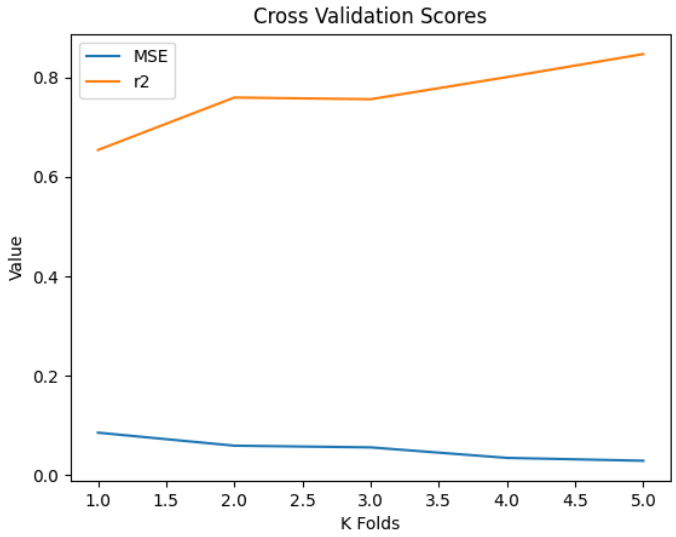

Figure 2. Cross Validation Scores

Furthermore, the elucidation of potential feature importance, as demonstrated in Figure 3, casts light on the attributes deemed most influential by the original linear regression model. The ascendancy of "Uniformity of Cell Size," "Uniformity of Cell Shape," and "Bare Nuclei" which would become more prominent with further future testing. These three features emphasize their significance within our analytical framework being top 3 of 4 features.

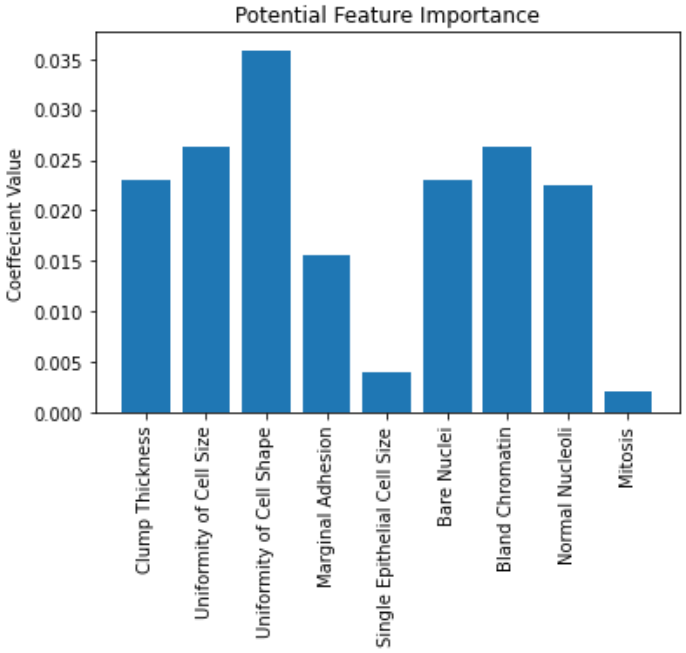

Figure 3. Potential Feature Importance

An endeavor was made to explore the significance of features through the utilization of a linear regression variation known as Lasso. Regrettably, this attempt yielded unsatisfactory outcomes. The coefficient of determination was merely 0.19, and the Mean Squared Error (MSE) stood at 0.18. Despite these discouraging results, the analysis did manage to underscore the importance of certain features. These features are "Uniformity of Cell Size," "Uniformity of Cell Shape," and "Bare Nuclei". Evidenced by their non-zero coefficient values, contrasting with the remaining features that exhibited coefficient values of 0 emphasizing their importance.

Subsequently, another effort was undertaken to employ a linear regression variation called Ridge. However, this endeavor proved to be less impactful, promising a potential that wasn't fully realized. Through multiple rounds of cross-validation, attempts were made to identify the optimal alpha value for the original Ridge model. The exploration led to a selected alpha value of 673.42, which, while considerably high, yielded an R2 score of 0.75 and an MSE of 0.06, mirroring the outcomes of the initial model.

Code snippet for linear regression and its variations (Lasso and Ridge):

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from scipy.stats import chi2_contingency
from sklearn.impute import KNNImputer
from sklearn.model_selection import train_test_split, cross_validate, learning_curve
from sklearn import linear_model
from sklearn.linear_model import RidgeCV
from sklearn.metrics import mean_squared_error, r2_score

%matplotlib inline

# Find contigency
def find_contigency(df: pd.DataFrame):
    columnDependenciesString = ""
    columns = df.columns
    for c in columns:
        if c != "Class":
            contigency= pd.crosstab(df[c], df['Class'])
            chi, p, dof, expected = chi2_contingency(contigency)
            if p > 0.05:
                columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" % p)
            else:
                columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f" % p)
    print("Find dependencies with current features:\n\n"+columnDependenciesString)
```

```
# Perform Linear regression, and with Lasso and Ridge
def linear_regression(df: pd.DataFrame, name: str, cv=5):
    y = df['Class']
    X = df.iloc[:,~df.columns.isin(['Class'])]

    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])
    lin_reg_modes = ['','Lasso','Ridge']
    print(name+" Linear Regression Dataset:\n")

    for x in lin_reg_modes:
        if x == 'Lasso':
            regr = linear_model.Lasso()
        elif x == 'Ridge':
            r_alphas = np.logspace(0, 5, 100)
            ridge_model = RidgeCV(alphas=r_alphas, scoring='r2')
            ridge_model = ridge_model.fit(x_train, y_train)
            regr = linear_model.Ridge(alpha=ridge_model.alpha_)
        else:
            regr = linear_model.LinearRegression()

        scores = cross_validate(regr, X, y, cv=cv, scoring=('r2', 'neg_mean_squared_error'),
return_train_score=True)
        regr.fit(x_train, y_train)
        y_pred = regr.predict(x_test)

        linear_regression_results = "\tLinear Regression " + x + ":\n\n" \
        + "\t\tCoefficient of determination: {}\n".format("%.2f" % r2_score(y_test, y_pred)) \
        + "\t\tMean Square Error: {}\n".format("%.2f" % mean_squared_error(y_test, y_pred)) \
        + "\t\tCross Validation ({} fold) Scores:\n\t\tNegative Mean Square: {}\n\t\tr2 scores: {}\n".format(cv,
str(scores["test_neg_mean_squared_error"]), str(scores["test_r2"])) \
        + ("\t\tRidge Model alpha value {}\n".format("%.2f" % ridge_model.alpha_) if x == 'Ridge' else '') \
        + "\t\tFinal Coefficients:" + str(regr.coef_) + "\n\n"

        print(linear_regression_results)

        # Produce learning curve
        train_sizes, train_scores, test_scores = learning_curve(regr, X, y, cv=cv, train_sizes=np.linspace(0.1,
0.99, 10))

        # Create mean of train and test scores
        train_mean = np.mean(train_scores, axis=1)
        test_mean = np.mean(test_scores, axis=1)
```

```
    x= [i for i in range(1, cv + 1)]
    plt.plot(x, abs(scores["test_neg_mean_squared_error"]), label="MSE")
    plt.plot(x, scores["test_r2"], label="r2")
    plt.legend()
    plt.title("Cross Validation Scores")
    plt.ylabel("Value")
    plt.xlabel("K Folds")
    plt.show()

    # Plot learning curve lines (mean of training and test scores)
    plt.figure()
    plt.plot(train_sizes, train_mean, '--',  label="Training score")
    plt.plot(train_sizes, test_mean,  label="Testing score")

    # Add title and labels and show the plot
    plt.title("Learning Curve")
    plt.xlabel("Training Set Size")
    plt.ylabel("R2 Score")
    plt.legend(loc="best")
    plt.show()

    # Show coeffecient weights, possible feature importance
    plt.figure()
    plt.title("Potential Feature Importance")
    plt.bar([x for x in X.columns], abs(regr.coef_))
    plt.xticks(rotation=90)
    plt.ylabel("Coeffecient Value")
    plt.show()

col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell Shape',
'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli', 'Mitosis',
'Class']
df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)
beforeCount = len(df)
df = df[~df.duplicated()]
afterCount = len(df)

print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

# Lets change the Class' column values from 2 and 4 to 0 - 1
df["Class"] = df["Class"].replace(2, 0)
df["Class"] = df["Class"].replace(4, 1)

# Find feature importance
```

```
find_contigency(df)

# This has no significance to the data, time to drop
df.drop("Sample code number", axis=1, inplace=True)
knn_imp = KNNImputer(n_neighbors=10, weights='distance', metric='nan_euclidean')
df["Bare Nuclei"] = knn_imp.fit_transform(df)

df.head()

linear_regression(df, "Data")
```

What we could do

We have approached data cleansing from a fresh perspective this time around. Instead of relying on the K-Nearest Neighbors (KNN) imputer to fill in missing values, we opted to remove those entries altogether due to concerns about potential data noise introduced by the imputer. Alongside the elimination of duplicate entries and the exclusion of the "Sample code number" column, we have taken a more rigorous approach by identifying and addressing outlier entries. Our commitment to maintaining data integrity led us to address these outliers, with 61 entries being carefully removed following the principles of Gaussian distribution. Notably, the "Single Epithelial Cell Size" column accounted for 30 outliers, and the "Mitosis" column accounted for 31 of these outliers.

As a result of these adjustments, we re-executed the linear model and observed a slight enhancement in the R2 value, which increased by 5% and now stands at 80% when making inferences on the test dataset. The Mean Squared Error (MSE) stands at 0.05. This suggests that the KNN imputer might not be the most suitable approach for analyzing this data with it inserting calculated values. Furthermore, the removal of outliers has contributed to the linear model's accuracy.

Code snippet of better data cleansing:

```
import pandas as pd
import os
from scipy.stats import chi2_contingency
from numpy import mean
from numpy import std

# Find outliers with Gaussian distribution, and drop if wanted
def find_outliers(name: str, df: pd.DataFrame, dropInPlace: bool):
    columns = df.columns
    outliersList = []
    outlierText = ""
    for c in columns:
```

```
                data_mean, data_std = mean(df[c]), std(df[c])
                cut_off = data_std * 3
                lower, upper = data_mean - cut_off, data_mean + cut_off

                outliers = df.loc[(df[c] < lower) | (df[c] > upper)]

                for o in outliers.index:
                    if len(outliersList) > 0:
                        index = None
                        for ol in outliersList:
                            if ol != o:
                                index = o
                                break
                        if index != None:
                            outliersList.append(o)
                    else:
                        outliersList.append(o)

                if(len(outliers.index) > 0 ):
                    outlierText += "{} Outliers for column {}\n".format(len(outliers), c)

            droppedText = "\n"
            if (dropInPlace):
                df.drop(outliers.index, axis=0, inplace=True)
                droppedText = "{} rows dropped".format(len(outliersList)) + "\n"

            outlierText = "Finding Outliers for {}\n\nIn total, there are {} rows with outliers".format(name,
len(outliersList)) + "\n" + outlierText + droppedText
            print(outlierText)

        # Find contigency
        def find_contigency(df: pd.DataFrame):
            columnDependenciesString = ""
            columns = df.columns
            for c in columns:
                if c != "Class":
                    contigency= pd.crosstab(df[c], df['Class'])
                    chi, p, dof, expected = chi2_contingency(contigency)
                    if p > 0.05:
                        columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" %
p)
                    else:
                        columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f"
% p)
```

```
        print("Find dependencies with current features:\n\n"+columnDependenciesString)

    col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell
Shape', 'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli',
'Mitosis', 'Class']
    df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)
    beforeCount = len(df)
    df = df[~df.duplicated()]
    afterCount = len(df)

    print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

    # Lets change the Class' column values from 2 and 4 to 0 - 1
    df["Class"] = df["Class"].replace(2, 0)
    df["Class"] = df["Class"].replace(4, 1)



    # Find feature importance
    find_contigency(df)

    # This has no significance to the data, time to drop
    df.drop("Sample code number", axis=1, inplace=True)

    df.dropna(inplace=True)

    find_outliers("Clean Dataset", df, True)

    df.head()
```

In the realm of improving our linear regression model, we employed cross-validation. Cross-validation is a technique used in machine learning and statistics to assess the performance of a predictive model and ensure that its results are not overly influenced by the way the data is split into training and testing sets. It is especially useful when working with limited amounts of data or when you want to have a better understanding of how well your model generalizes to new, unseen data. Initially, our model was evaluated, yielding an R2 score of 0.71. A higher R2 score indicates better model fit, and the initial score of 0.71 indicated room for improvement.

The most intriguing progress surfaced after the application of cross-validation, which entails training and testing the model on distinct data subsets. This brought about significant enhancements. The R2 score surged from 0.71 to 0.87, resembling the transition from solving 71% of a puzzle to solving 87%, highlighting an improved fit to the data. Furthermore, the MSE plummeted from 0.07 to 0.02, signifying the model's

predictions became closer to actual values. This analogy can be likened to gaining proficiency in hitting a target, with darts landing nearer to the bullseye. The R2 score and MSE improvements observed post cross-validation reflect the model's heightened proficiency in comprehending the data and generating precise predictions. This advancement is pivotal, augmenting the model's effectiveness and reliability within real-world scenarios.

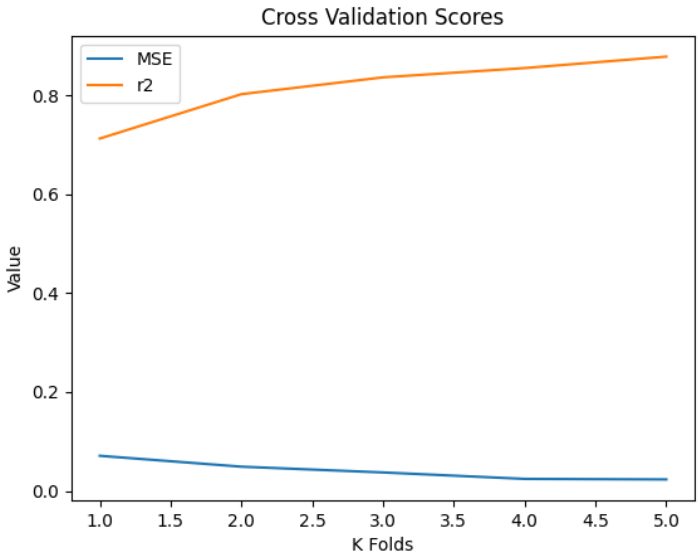

Figure 4. Cross-validation results of the r2 score and the MSE score

When it comes to diagnosis, we want to make sure we are as accurate as possible. There is the potential to achieve an R2 score of 87%. That would be 13 out of 100 people getting an inaccurate diagnosis. This ratio is too high, and we need to approach this problem differently and more effectively. We have decided to switch from linear regression to classification models instead as this was driven by the nature of the problem we are dealing with. Linear regression is primarily used for predicting continuous values, like predicting the price of a house based on its features. However, in our case, we are dealing with a classification task where we want to categorize cases into specific classes (malignant or not malignant). This is a fundamentally different problem than predicting a continuous value.

Classification models are designed specifically for tasks like these, where the goal is to assign data points to distinct categories or classes. These models consider the relationships between features and classes to make accurate categorizations. Given that our aim is to classify breast cancer cases as malignant or not, using classification models is a more suitable approach.

In our context, we will be using a logistic regression model and a random forest classification model in this case. Logistic regression is a fundamental classification algorithm that models the probability of an instance belonging to a particular class. It is a linear model that uses the logistic function to map its output into a probability range between 0 and 1. In the case of binary classification, like malignant or benign, logistic regression calculates the likelihood of an input belonging to the positive class (e.g., malignant) based on the features. It's relatively simple and interpretable, as the coefficients associated with each feature provide insight into their impact on the classification decision. However, logistic regression assumes a linear

relationship between features and the log-odds of the target class, which might not capture complex interactions in the data.

A random forest, on the other hand, is an ensemble learning technique that consists of multiple decision trees. Each tree is built using a subset of the data and a random subset of features, making the model robust and capable of capturing nonlinear relationships and interactions between features. When making predictions, the random forest combines the predictions of individual trees to arrive at a final classification. This ensemble approach tends to be more accurate and resistant to overfitting compared to a single decision tree. Random forests also provide feature importance scores, which help in identifying which features contribute the most to the classification.

By performing a Logistic regression model on our dataset, we have been given excellent results. A superb accuracy score of 96%. Accuracy in this case is the ratio of correct predictions to total predictions. It is a measure of overall correctness and is calculated as (true positives + true negatives) / (true positives + true negatives + false positives + false negatives). As seen in Figure 5., there is a confusion matrix that lays out the true positives, true negatives, false positives, and false negatives for the test data from the logistic regression model. When it comes to this analytic framework, we considered it a binary classification. The model either predicts malignant diagnosis or not, nothing in between. This allows us to generate this confusion matrix. In our case, the calculated accuracy is 0.966, which means the model is correct about 96.6% of the time.

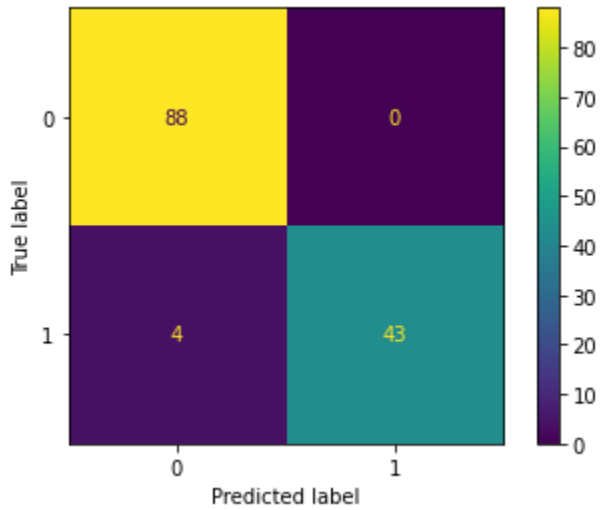

Figure 5. Confusion Matrix of Test data in the logistic regression model

The AUC score (Area Under Curve) is a measurement of how well the model can distinguish between the two classes (malignant and benign in this case). A value of 1 would indicate perfect separation, while a value of 0.5 suggests random guessing. So, with our logistic regression model achieving a mean AUC score of 0.9939 indicates that the model is excellent at distinguishing between the two classes, with truly little overlap. Recall (Sensitivity) is the ratio of true positives to the total actual positives. It shows how well the model finds all positive cases and is calculated as tp / (tp + fn). Here, the recall is 0.968, indicating the model correctly identified 96.8% of actual positive cases. Precision is another measurement of the ratio of true positives to the total predicted positives. It tells us how many of the predicted positive cases are actually positive and is calculated as tp / (tp + fp). In our case, the precision is 0.979, meaning that when the model predicts a case as malignant, it is correct about 97.9% of the time. Specificity is the ratio of true negatives to the total actual negatives. It shows how well the model finds all negative cases and is calculated as tn / (tn + fp). Here, the specificity is 0.961, indicating the model correctly identified 96.1% of actual negative cases. Log loss is another

measurement tool, which measures the performance of a classification model where the prediction is a probability value between 0 and 1. It quantifies how well the predicted probabilities match the true classes. Lower log loss values indicate better performance. This model's log loss is around 1.02 for test data, which is a measure of how well the predicted probabilities align with the actual classes.

In summary, the logistic regression model has performed remarkably well. It has achieved high accuracy, recall, precision, and AUC score, indicating its strong ability to classify between malignant and benign cases accurately. Much better than the linear regression model we have been previously using.

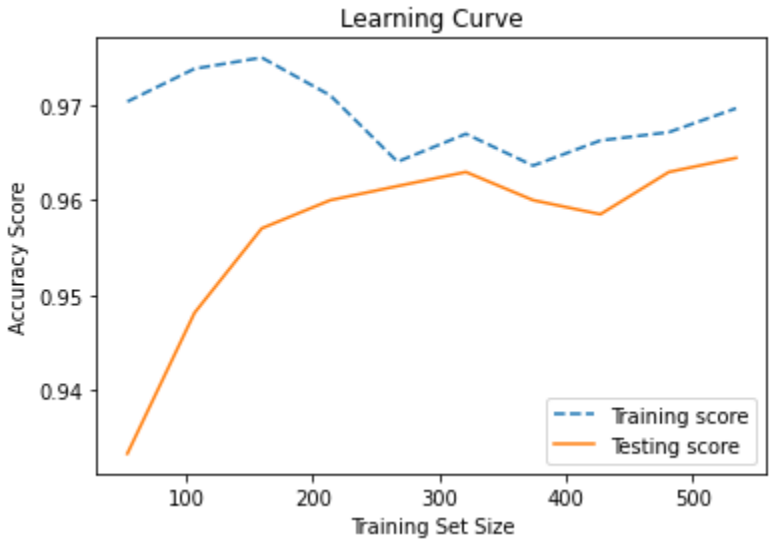

Figure 6. to show no overfitting of the logistic model

Attaining high accuracy can sometimes stem from overfitting the model, a situation that is problematic when dealing with new and unanticipated data. Like thoroughly memorizing answers to practice questions before a test, an overfitted model tends to overly specialize on training data intricacies. This makes it less adaptable to novel data scenarios, leading to diminished performance. However, illustrated in Figure 6, the transformative trajectory of the model's accuracy during its learning process becomes apparent that it may be slightly overfitted. As shown, the logistic regression model's attainment of a 96% accuracy threshold may have some overfitting patterns, which signifies that the model may be on some merits merely memorization based. Some of the patterns exhibited such as the gap between training and testing accuracy, along with the fluctuations in the testing accuracy, could be indicative of overfitting, especially as the training dataset size increases.

Code snippet of logistic regression:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from numpy import mean
from numpy import std
from scipy.stats import chi2_contingency
```

```
        from sklearn.model_selection import train_test_split, cross_val_score, learning_curve,
validation_curve
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import mean_squared_error, r2_score, ConfusionMatrixDisplay, log_loss,
confusion_matrix, accuracy_score

        %matplotlib inline

        # Find outliers with Gaussian distribution, and drop if wanted
        def find_outliers(name: str, df: pd.DataFrame, dropInPlace: bool):
            columns = df.columns
            outliersList = []
            outlierText = ""
            for c in columns:
                data_mean, data_std = mean(df[c]), std(df[c])
                cut_off = data_std * 3
                lower, upper = data_mean - cut_off, data_mean + cut_off

                outliers = df.loc[(df[c] < lower) | (df[c] > upper)]

                for o in outliers.index:
                    if len(outliersList) > 0:
                        index = None
                        for ol in outliersList:
                            if ol != o:
                                index = o
                                break
                        if index != None:
                            outliersList.append(o)
                    else:
                        outliersList.append(o)

                if(len(outliers.index) > 0 ):
                    outlierText += "{} Outliers for column {}\n".format(len(outliers), c)

            droppedText = "\n"
            if (dropInPlace):
                df.drop(outliers.index, axis=0, inplace=True)
                droppedText = "{} rows dropped".format(len(outliersList)) + "\n"

            outlierText = "Finding Outliers for {}\n\nIn total, there are {} rows with outliers".format(name,
len(outliersList)) + "\n" + outlierText + droppedText
            print(outlierText)

        # Find contigency
        def find_contigency(df: pd.DataFrame):
            columnDependenciesString = ""
            columns = df.columns
            for c in columns:
```

```
                if c != "Class":
                    contigency= pd.crosstab(df[c], df['Class'])
                    chi, p, dof, expected = chi2_contingency(contigency)
                    if p > 0.05:
                        columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" %
p)
                    else:
                        columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f"
% p)
            print("Find dependencies with current features:\n\n"+columnDependenciesString)

        def logistic_regression(df: pd.DataFrame, name: str, cv=5):
            y = df['Class']
            X = df.iloc[:,~df.columns.isin(['Class'])]
            x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])

            log_reg = LogisticRegression(solver='liblinear')

            cv_score = cross_val_score(log_reg, x_train, y_train, cv=cv, scoring='roc_auc')

            # Fit classifier with training data
            log_reg.fit(x_train, y_train)

            # Get predictions
            y_train_predict = log_reg.predict(x_train)
            y_test_predict = log_reg.predict(x_test)

            # Print results
            cmTrain = confusion_matrix(y_train, y_train_predict)
            cmTest = confusion_matrix(y_test, y_test_predict)
            tp = cmTrain[0][0] + cmTest[0][0]
            tn = cmTrain[1][1] + cmTest[1][1]
            fp = cmTrain[0][1] + cmTest[0][1]
            fn = cmTrain[1][0] + cmTest[1][0]

            acc = (tp + tn) / (tp + tn + fp + fn)
            precision = tp / (tp + fp)
            recall = tp / (tp + fn)
            sensitivity = recall
            specificity = tn / (fp + tn)

            results = "\t\tAccuracy: {}\n".format("%.2f" % accuracy_score(y_pred=y_test_predict,
y_true=y_test)) \
            + "\t\tMean Cross Validation Area Under Curve Score: {}\n".format(cv_score.mean()) \
            + "\t\tConfusion Matrix (Training + Test): tp: {} tn: {} fp: {} fn: {}\n".format(tp,tn,fp,fn) \
            + "\t\tTotals (Training + Test): Accuracy: {} Recall: {} Precision: {} Sensitivity: {} Specificity:
{}\n".format(round(acc,3), round(recall,3), round(precision,3), round(sensitivity,3), round(specificity,3))  \
            + "\t\tLog loss on training data: {}\n".format(log_loss(y_train,y_train_predict)) \
```

```
            + "\t\tLog loss on test data: {} \n".format(log_loss(y_test,y_test_predict))

            print(name+" Logistic Regression Dataset:\n")
            print(results)

            # Plot Confusion matrix
            ConfusionMatrixDisplay.from_estimator(log_reg, x_test, y_test)

            # Produce learning curve
            train_sizes, train_scores, test_scores = learning_curve(log_reg, X, y, cv=cv,
train_sizes=np.linspace(.1, 0.99, 10))

            # Create mean of train and test scores
            train_mean = np.mean(train_scores, axis=1)
            test_mean = np.mean(test_scores, axis=1)

            # Plot learning curve lines (mean of training and test scores)
            plt.figure()
            plt.plot(train_sizes, train_mean, '--',  label="Training score")
            plt.plot(train_sizes, test_mean,  label="Testing score")

            # Add title and labels and show the plot
            plt.title("Learning Curve")
            plt.xlabel("Training Set Size")
            plt.ylabel("Accuracy Score")
            plt.legend(loc="best")
            plt.show()

            # Define the range of parameter to be tested
            param_range = [0.001, 0.05, 0.1, 0.5, 1.0, 10.0]
            train_scores, test_scores = validation_curve(log_reg, X, y, param_name="C",
param_range=param_range, cv=cv)
            # Calculate mean for training and test scores
            train_mean = np.mean(train_scores, axis=1)
            test_mean = np.mean(test_scores, axis=1)

            # Plot validation curve lines (mean of training and test scores)
            plt.figure()
            plt.plot(param_range, train_mean, '--',label="Training score")
            plt.plot(param_range, test_mean, label="Testing score")

            # Add title and labels and show the plot
            plt.title("Validation Curve")
            plt.ylim([0.5, 1.0])
            plt.xlabel("Value of regularization term")
            plt.ylabel("Accuracy Score")
            plt.legend(loc="best")
            plt.show()
```

```
            # Show coeffecient weights, possible feature importance
            plt.figure()
            plt.bar([x for x in X.columns], abs(log_reg.coef_[0]))
            plt.ylabel("Coeffecient Value")
            plt.xticks(rotation=90)
            plt.show()

        col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell
Shape', 'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli',
'Mitosis', 'Class']
        df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)
        beforeCount = len(df)
        df = df[~df.duplicated()]
        afterCount = len(df)



        print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

        # Lets change the Class' column values from 2 and 4 to 0 - 1
        df["Class"] = df["Class"].replace(2, 0)
        df["Class"] = df["Class"].replace(4, 1)

        # Find feature importance
        find_contigency(df)

        # This has no significance to the data, time to drop
        df.drop("Sample code number", axis=1, inplace=True)

        df.dropna(inplace=True)

        find_outliers("Clean Dataset", df, True)

        df.head()

        logistic_regression(df, "Logistic Regression")
```

Although overfitting is indicative of producing bad results when faced with unanticipated data. This model still performs very well considering the AUC and log loss scores. Instead of running the risk of an overfitted model, perhaps we can have better luck with the random forest and hopefully deter from overfitting. A random forest is like a group of experts coming together to make a decision. Imagine you want to make a decision, and you ask multiple people for their opinions. Each person is like a "tree" that offers an opinion based on a subset of information. Then, you combine all these opinions to make the final decision. In the case of a random forest, these "trees" are individual decision trees, and they work together to improve the accuracy and reliability of predictions. Unfortunately, it is also prone to overfitting.

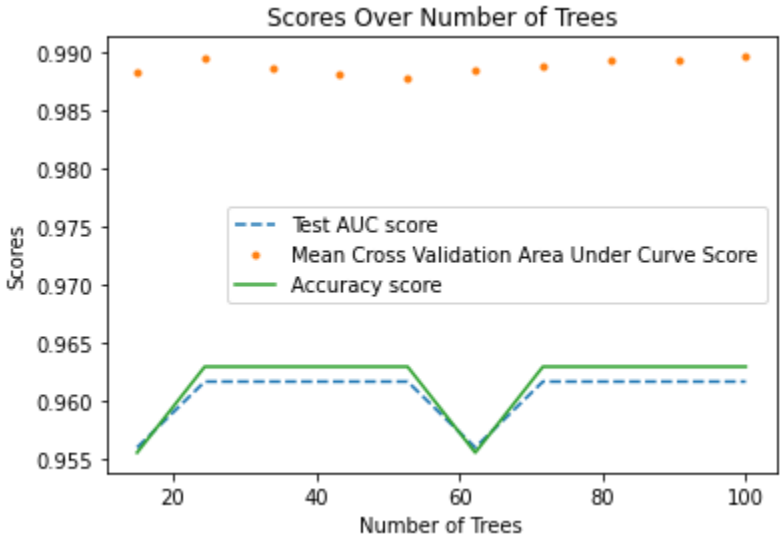

Figure 7. Random Forest scores

In our case, we have generated 100 trees in the forest and achieved an accuracy score of 96%. The Mean Cross Validation Area Under Curve (AUC) Score is 0.989. The mean AUC score tells us how well the random forest model can distinguish between the two classes (malignant and benign) on average. A value close to 1 indicates excellent separation. In this case, the mean AUC score of 0.9896 suggests that the random forest is very good at distinguishing between malignant and benign cases. We can see as illustrated in Figure 7 our scores dependent on how many trees in the forest.

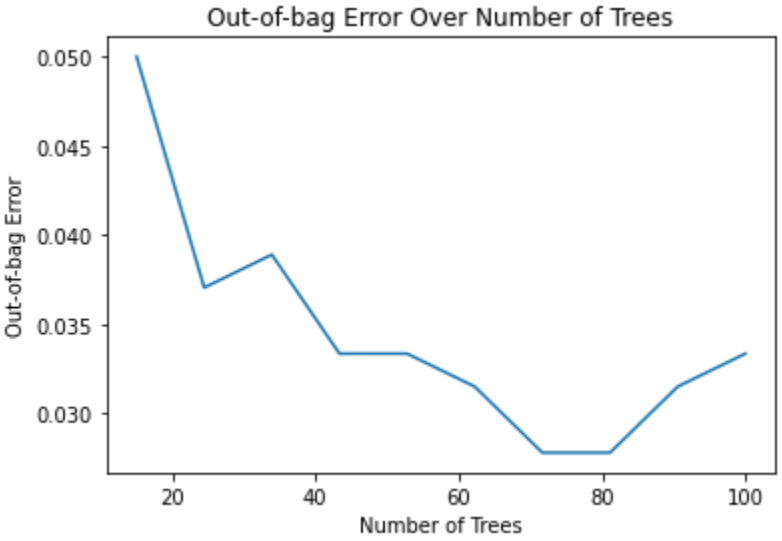

Figure 8. Out-of-bag Error vs number of Trees

In a random forest, each tree is trained on a different subset of the data. The OOB error measures how well each individual tree performs on the data it did not see during training. It is like testing each "tree expert" on problems they have not encountered before. In our case, achieving an OOB error of 0.03 (3%) is very good. A low OOB error suggests that each tree in the forest is performing well on unseen data, which in turn contributes to the overall accuracy of the random forest. Therefore the random forest is not overfitted whereas the logistic regression model potentially could be.

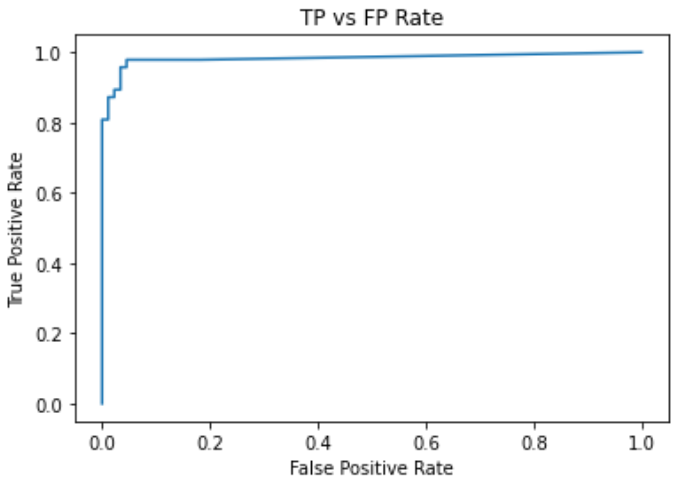

Figure 9. TP vs FP rate of Random Forest.

In medical diagnosis, the aim is to strike a balance between true positives, which is correctly identifying cases of disease, and false positives which is incorrectly identifying cases as disease. While it is important to catch as many actual cases as possible, having a high TP rate, it is equally crucial to minimize the number of false positives, which is a low FP rate. A high TP rate is indicative of the model's effectiveness in correctly identifying true cases, and a low FP rate shows that the model is being cautious in not falsely diagnosing cases. Achieving this balance ensures that the diagnostic process is reliable, accurate, and trustworthy for both patients and healthcare professionals. While it is important to focus on true positives and the model's ability to catch actual cases of disease, false positives should not be overlooked. Striking a balance between true positives and false positives is essential for a medical model to provide accurate and responsible diagnostic outcomes. As can be seen in Figure 9, the TP vs FP rate is close to 100% true positive. We still want to get as close to 100% accuracy as possible, as a misdiagnosis in the medical field is what we want to avoid.

Code snippet of random forest:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from numpy import mean
from numpy import std
from scipy.stats import chi2_contingency
from sklearn.model_selection import train_test_split, cross_validate
from sklearn.metrics import roc_auc_score, roc_curve, accuracy_score
from sklearn.ensemble import RandomForestClassifier

%matplotlib inline

# Find outliers with Gaussian distribution, and drop if wanted
def find_outliers(name: str, df: pd.DataFrame, dropInPlace: bool):
    columns = df.columns
    outliersList = []
    outlierText = ""
```

```
            for c in columns:
                data_mean, data_std = mean(df[c]), std(df[c])
                cut_off = data_std * 3
                lower, upper = data_mean - cut_off, data_mean + cut_off

                outliers = df.loc[(df[c] < lower) | (df[c] > upper)]

                for o in outliers.index:
                    if len(outliersList) > 0:
                        index = None
                        for ol in outliersList:
                            if ol != o:
                                index = o
                                break
                        if index != None:
                            outliersList.append(o)
                    else:
                        outliersList.append(o)

                if(len(outliers.index) > 0 ):
                    outlierText += "{} Outliers for column {}\n".format(len(outliers), c)

            droppedText = "\n"
            if (dropInPlace):
                df.drop(outliers.index, axis=0, inplace=True)
                droppedText = "{} rows dropped".format(len(outliersList)) + "\n"

            outlierText = "Finding Outliers for {}\n\nIn total, there are {} rows with outliers".format(name,
len(outliersList)) + "\n" + outlierText + droppedText
            print(outlierText)

        # Find contigency
        def find_contigency(df: pd.DataFrame):
            columnDependenciesString = ""
            columns = df.columns
            for c in columns:
                if c != "Class":
                    contigency= pd.crosstab(df[c], df['Class'])
                    chi, p, dof, expected = chi2_contingency(contigency)
                    if p > 0.05:
                        columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" %
p)
                    else:
                        columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f"
% p)
            print("Find dependencies with current features:\n\n"+columnDependenciesString)

        def random_forest(df: pd.DataFrame, name: str, cv=5):
            y = df['Class']
```

```
            X = df.iloc[:,~df.columns.isin(['Class'])]

            x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])

            # number of tress to produce
            num = 100
            numTrees = np.linspace(15, num, 10)
            rf = RandomForestClassifier(oob_score=True, random_state=0, n_jobs=8)

            error_array = []
            roc_auc_score_array = []
            cv_score_array = []
            accuracy_score_array = []
            # Iterate over number of trees
            print(name+" Random Forest Dataset:\n")
            print("\t\tNumber of trees in forest:")
            for i in range(len(numTrees)):
               print("\t\t {}".format(int(numTrees[i])))

               rf.set_params(n_estimators=int(numTrees[i]))

               scores = cross_validate(rf, X, y, cv=cv, scoring='roc_auc', return_train_score=True)
               cv_score_array.append(np.mean(scores['test_score']))

               rf.fit(x_train, y_train)

               y_predict = rf.predict(x_test)

               roc_auc_score_array.append(roc_auc_score(y_test, y_predict))
               accuracy_score_array.append(accuracy_score(y_pred=y_predict, y_true=y_test))

               oob_error = 1 - rf.oob_score_
               error_array.append(oob_error)

            # Evaluate model performance on the test data
            y_rf_pred_prob = rf.predict_proba(x_test)

            results = "\t\tRandom Forest scores with {} trees\n".format(num) \
            + "\t\tAccuracy: {}\n".format("%.2f" % accuracy_score(y_pred=y_predict, y_true=y_test)) \
            + "\t\tMean Cross Validation Area Under Curve Score: {}\n".format(np.mean(cv_score_array[-1]))
\
            + "\t\tRandom Forest Area Under Curve Score: {}\n".format(roc_auc_score(y_test,
y_rf_pred_prob[:,1]))

            print(results)

            # Plot scores graph
            plt.figure()
            plt.title("Scores Over Number of Trees")
```

```
            plt.plot(numTrees, roc_auc_score_array,'--', label="Test AUC score")
            plt.plot(numTrees, cv_score_array, '.', label="Mean Cross Validation Area Under Curve Score")
            plt.plot(numTrees, accuracy_score_array, label="Accuracy score")
            plt.ylabel("Scores")
            plt.xlabel("Number of Trees")
            plt.legend(loc="best")
            plt.show()

            # Plot OOB error graph
            plt.figure()
            plt.title("Out-of-bag Error Over Number of Trees")
            plt.plot(numTrees, error_array)
            plt.ylabel("Out-of-bag Error")
            plt.xlabel("Number of Trees")
            plt.show()

            # Show coeffecient weights, possible feature importance
            plt.figure()
            plt.title("Potential Feature Importance")
            plt.bar([x for x in X.columns], abs(rf.feature_importances_))
            plt.xticks(rotation=90)
            plt.ylabel("Coeffecient Value")
            plt.show()

            fpr, tpr, _ = roc_curve(y_test, y_rf_pred_prob[:,1])

            # Plot FalsePositiveRate to TruePositiveRate
            plt.figure()
            plt.title("TP vs FP Rate")
            plt.plot(fpr,tpr)
            plt.ylabel("True Positive Rate")
            plt.xlabel("False Positive Rate")
            plt.show()

        col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell
Shape', 'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli',
'Mitosis', 'Class']
        df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)
        beforeCount = len(df)
        df = df[~df.duplicated()]
        afterCount = len(df)

        print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

        # Lets change the Class' column values from 2 and 4 to 0 - 1
        df["Class"] = df["Class"].replace(2, 0)
        df["Class"] = df["Class"].replace(4, 1)
```

```
# Find feature importance
find_contigency(df)

# This has no significance to the data, time to drop
df.drop("Sample code number", axis=1, inplace=True)

df.dropna(inplace=True)

find_outliers("Clean Dataset", df, True)

df.head()

random_forest(df, "Random Forest")
```

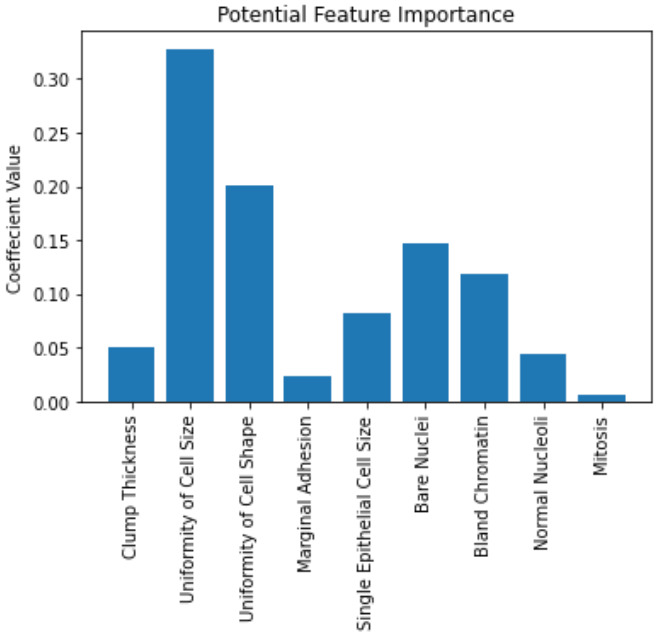

Figure 10. Feature importance of Random Forest model.

Previously highlighted, the feature importance indicates the pivotal role of certain attributes like "Uniformity of Cell Size," "Uniformity of Cell Shape," and "Bare Nuclei" in prediction. This concept can be likened to identifying key elements in a recipe. These attributes significantly contributed to the accuracy of the random forest's classification ability, as demonstrated in Figure 10. The model's effectiveness, underscored by AUC scores, OOB error, TP vs FP, and feature importance, suggests its potential alignment with our company's objectives.

Further refinement might be achieved by incorporating a novel feature to bolster model performance. It's evident from Figure 10 that "Uniformity of Cell Size," "Uniformity of Cell Shape," and "Bare Nuclei" are standout attributes. Confirming this, the chi contingency test identifies these columns as the most

interdependent. Combining these attributes into a new feature, possibly through averaging, holds promise for enhancing the random forest's accuracy.

Upon reevaluating the random forest with this new feature, the accuracy decreased from 96% to 94%. However, Figure 11 depicts the assimilation of the new feature as the most influential. Unfortunately, in this context, accuracy takes precedence, and integrating the new feature might not be the optimal choice.

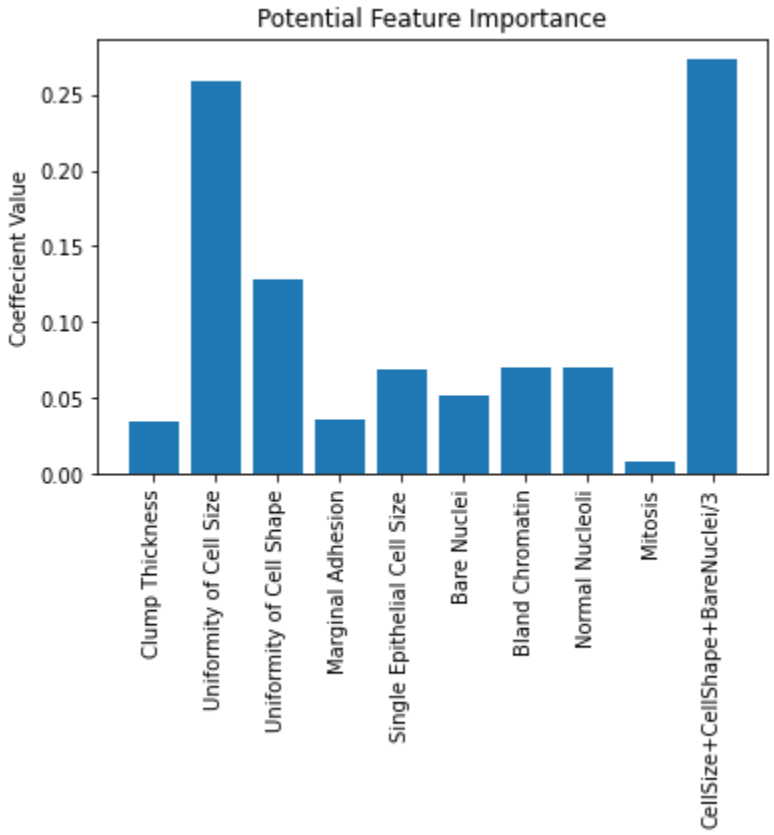

Figure 11. Feature importance with new feature in random Forest.

Code snippet of adding the new feature:

```
# Essentially just add this line of code before the random_forest function call in the previous code snippet
# To add the new feature to the dataset

df["CellSize+CellShape+BareNuclei/3"] = (df["Uniformity of Cell Size"] + df["Uniformity of Cell Shape"] +
df["Bare Nuclei"]) / 3
```

Final Directional Path

In conclusion, our investigation into improving the detection of malignant breast cancer diagnoses based on William Wolberg's Breast Cancer dataset has yielded significant insights. Initially relying on linear regression techniques, we encountered limitations with an achieved coefficient of determination of 75%. Even

after exploring variations like Lasso and Ridge, notable enhancements remained elusive. The transition to classification models marked a turning point, where both a logistic regression model and a random forest model achieved remarkable accuracy rates of 96%. These findings advocate for a strategic shift towards classification models in our pursuit of improved breast cancer diagnosis.

Our journey began with a dataset containing noisy and missing data, necessitating rigorous data cleansing efforts. We addressed missing values, duplicates, and outliers to bolster data integrity. The utilization of cross-validation exposed room for enhancement in the linear regression model, leading to a higher R2 score of 87% and a reduced Mean Squared Error (MSE) of 0.02. Transitioning to classification models proved highly promising. The logistic regression model exhibited excellent accuracy, AUC score, recall, precision, and specificity, effectively outperforming the linear regression model. However, the potential for overfitting was observed, prompting consideration of the random forest model.

The random forest approach demonstrated impressive accuracy and AUC scores, reflecting its capability to distinguish between classes effectively. The Out-of-Bag (OOB) error, an indicator of model generalization, remained low, suggesting that overfitting was minimized. Feature importance analysis highlighted key attributes like "Uniformity of Cell Size," "Uniformity of Cell Shape," and "Bare Nuclei," suggesting their critical role in classification. Despite the potential benefits of a novel feature, its integration resulted in decreased accuracy, prompting caution in its implementation.

In essence, the journey from linear regression to classification models has showcased the potential for improved accuracy and robustness in breast cancer diagnosis. The logistic regression and random forest models have demonstrated significant promise, advocating for their adoption to enhance our diagnostic capabilities and contribute effectively to the field. By embracing classification methodologies, we strive to achieve more accurate and reliable outcomes, thereby making a meaningful impact on the lives of patients.

Code Implementation

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os

from numpy import mean
from numpy import std
from scipy.stats import chi2_contingency
from sklearn.impute import KNNImputer
from sklearn.model_selection import train_test_split, cross_validate, cross_val_score, learning_curve,
validation_curve
from sklearn import linear_model
from sklearn.linear_model import LogisticRegression, RidgeCV
from sklearn.metrics import mean_squared_error, r2_score, ConfusionMatrixDisplay, log_loss,
confusion_matrix, roc_auc_score, roc_curve, accuracy_score
```

```
from sklearn.ensemble import RandomForestClassifier

%matplotlib inline

col_names = ['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of Cell Shape',
'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin', 'Normal Nucleoli', 'Mitosis',
'Class']
df = pd.read_csv(os.path.join(os.path.abspath(''),"breast+cancer+wisconsin+original\\breast-cancer-
wisconsin.data"), na_values="?", names=col_names)

pd.set_option("display.min_rows", df.shape[0]+1)
pd.set_option('display.max_rows', df.shape[0]+1)

# Find outliers with Gaussian distribution, and drop if wanted
def find_outliers(name: str, df: pd.DataFrame, dropInPlace: bool):
    columns = df.columns
    outliersList = []
    outlierText = ""
    for c in columns:
        data_mean, data_std = mean(df[c]), std(df[c])
        cut_off = data_std * 3
        lower, upper = data_mean - cut_off, data_mean + cut_off

        outliers = df.loc[(df[c] < lower) | (df[c] > upper)]

        for o in outliers.index:
            if len(outliersList) > 0:
                index = None
                for ol in outliersList:
                    if ol != o:
                        index = o
                        break
                if index != None:
                    outliersList.append(o)
            else:
                outliersList.append(o)

        if(len(outliers.index) > 0 ):
            outlierText += "{} Outliers for column {}\n".format(len(outliers), c)

    droppedText = "\n"
    if (dropInPlace):
        df.drop(outliers.index, axis=0, inplace=True)
        droppedText = "{} rows dropped".format(len(outliersList)) + "\n"
```

```
    outlierText = "Finding Outliers for {}\n\nIn total, there are {} rows with outliers".format(name,
len(outliersList)) + "\n" + outlierText + droppedText
    print(outlierText)



# Find contigency
def find_contigency(df: pd.DataFrame):
    columnDependenciesString = ""
    columns = df.columns
    for c in columns:
        if c != "Class":
            contigency= pd.crosstab(df[c], df['Class'])
            chi, p, dof, expected = chi2_contingency(contigency)
            if p > 0.05:
                columnDependenciesString += "{} column INDEPENDENT -> {} p-value\n".format(c, "%.2f" % p)
            else:
                columnDependenciesString += "{} column DEPENDENT -> {} p-value\n".format(c, "%.100f" % p)
    print("Find dependencies with current features:\n\n"+columnDependenciesString)



# Perform Linear regression, and with Lasso and Ridge
def linear_regression(df: pd.DataFrame, name: str, cv=5):

    y = df['Class']
    X = df.iloc[:,~df.columns.isin(['Class'])]

    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])

    lin_reg_modes = ['','Lasso','Ridge']

    print(name+" Linear Regression Dataset:\n")

    for x in lin_reg_modes:
        if x == 'Lasso':
            regr = linear_model.Lasso()
        elif x == 'Ridge':
            r_alphas = np.logspace(0, 5, 100)
            ridge_model = RidgeCV(alphas=r_alphas, scoring='r2')
            ridge_model = ridge_model.fit(x_train, y_train)
            regr = linear_model.Ridge(alpha=ridge_model.alpha_)
        else:
            regr = linear_model.LinearRegression()
```

```
    scores = cross_validate(regr, X, y, cv=cv, scoring=('r2', 'neg_mean_squared_error'),
return_train_score=True)

    regr.fit(x_train, y_train)
    y_pred = regr.predict(x_test)

    linear_regression_results = "\tLinear Regression " + x + ":\n\n" \
    + "\t\tCoefficient of determination: {}\n".format("%.2f" % r2_score(y_test, y_pred)) \
    + "\t\tMean Square Error: {}\n".format("%.2f" % mean_squared_error(y_test, y_pred)) \
    + "\t\tCross Validation ({} fold) Scores:\n\t\tNegative Mean Square: {}\n\t\tr2 scores: {}\n".format(cv,
str(scores["test_neg_mean_squared_error"]), str(scores["test_r2"])) \
    + ("\t\tRidge Model alpha value {}\n".format("%.2f" % ridge_model.alpha_) if x == 'Ridge' else '') \
    + "\t\tFinal Coefficients:" + str(regr.coef_) + "\n\n"

    print(linear_regression_results)

    # Produce learning curve
    train_sizes, train_scores, test_scores = learning_curve(regr, X, y, cv=cv, train_sizes=np.linspace(0.1,
0.99, 10))
    # Create mean of train and test scores
    train_mean = np.mean(train_scores, axis=1)
    test_mean = np.mean(test_scores, axis=1)

    x= [i for i in range(1, cv + 1)]
    plt.plot(x, abs(scores["test_neg_mean_squared_error"]), label="MSE")
    plt.plot(x, scores["test_r2"], label="r2")
    plt.legend()
    plt.title("Cross Validation Scores")
    plt.ylabel("Value")
    plt.xlabel("K Folds")
    plt.show()

    # Plot learning curve lines (mean of training and test scores)
    plt.figure()
    plt.plot(train_sizes, train_mean, '--',  label="Training score")
    plt.plot(train_sizes, test_mean,  label="Testing score")

    # Add title and labels and show the plot
    plt.title("Learning Curve")
    plt.xlabel("Training Set Size")
    plt.ylabel("R2 Score")
    plt.legend(loc="best")
    plt.show()

    # Show coeffecient weights, possible feature importance
```

```
        plt.figure()
        plt.title("Potential Feature Importance")
        plt.bar([x for x in X.columns], abs(regr.coef_))
        plt.xticks(rotation=90)
        plt.ylabel("Coeffecient Value")
        plt.show()

def logistic_regression(df: pd.DataFrame, name: str, cv=5):
    y = df['Class']
    X = df.iloc[:,~df.columns.isin(['Class'])]

    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])

    log_reg = LogisticRegression(solver='liblinear')

    cv_score = cross_val_score(log_reg, x_train, y_train, cv=cv, scoring='roc_auc')

    # Fit classifier with training data
    log_reg.fit(x_train, y_train)

    # Get predictions
    y_train_predict = log_reg.predict(x_train)
    y_test_predict = log_reg.predict(x_test)

    # Print results
    cmTrain = confusion_matrix(y_train, y_train_predict)
    cmTest = confusion_matrix(y_test, y_test_predict)
    tp = cmTrain[0][0] + cmTest[0][0]
    tn = cmTrain[1][1] + cmTest[1][1]
    fp = cmTrain[0][1] + cmTest[0][1]
    fn = cmTrain[1][0] + cmTest[1][0]

    acc = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    sensitivity = recall
    specificity = tn / (fp + tn)

    results = "\t\tAccuracy: {}\n".format("%.2f" % accuracy_score(y_pred=y_test_predict, y_true=y_test)) \
    + "\t\tMean Cross Validation Area Under Curve Score: {}\n".format(cv_score.mean()) \
    + "\t\tConfusion Matrix (Training + Test): tp: {} tn: {} fp: {} fn: {}\n".format(tp,tn,fp,fn) \
    + "\t\tTotals (Training + Test): Accuracy: {} Recall: {} Precision: {} Sensitivity: {} Specificity:
{}\n".format(round(acc,3), round(recall,3), round(precision,3), round(sensitivity,3), round(specificity,3))  \
    + "\t\tLog loss on training data: {}\n".format(log_loss(y_train,y_train_predict)) \
```

```
    + "\t\tLog loss on test data: {} \n".format(log_loss(y_test,y_test_predict))

    print(name+" Logistic Regression Dataset:\n")
    print(results)

    # Plot Confusion matrix
    ConfusionMatrixDisplay.from_estimator(log_reg, x_test, y_test)

    # Produce learning curve
    train_sizes, train_scores, test_scores = learning_curve(log_reg, X, y, cv=cv, train_sizes=np.linspace(.1,
0.99, 10))

    # Create mean of train and test scores
    train_mean = np.mean(train_scores, axis=1)
    test_mean = np.mean(test_scores, axis=1)

    # Plot learning curve lines (mean of training and test scores)
    plt.figure()
    plt.plot(train_sizes, train_mean, '--',  label="Training score")
    plt.plot(train_sizes, test_mean,  label="Testing score")

    # Add title and labels and show the plot
    plt.title("Learning Curve")
    plt.xlabel("Training Set Size")
    plt.ylabel("Accuracy Score")
    plt.legend(loc="best")
    plt.show()

    # Define the range of parameter to be tested
    param_range = [0.001, 0.05, 0.1, 0.5, 1.0, 10.0]
    train_scores, test_scores = validation_curve(log_reg, X, y, param_name="C", param_range=param_range,
cv=cv)
    # Calculate mean for training and test scores
    train_mean = np.mean(train_scores, axis=1)
    test_mean = np.mean(test_scores, axis=1)

    # Plot validation curve lines (mean of training and test scores)
    plt.figure()
    plt.plot(param_range, train_mean, '--',label="Training score")
    plt.plot(param_range, test_mean, label="Testing score")

    # Add title and labels and show the plot
    plt.title("Validation Curve")
    plt.ylim([0.5, 1.0])
    plt.xlabel("Value of regularization term")
```

```
    plt.ylabel("Accuracy Score")
    plt.legend(loc="best")
    plt.show()

    # Show coeffecient weights, possible feature importance
    plt.figure()
    plt.bar([x for x in X.columns], abs(log_reg.coef_[0]))
    plt.ylabel("Coeffecient Value")
    plt.xticks(rotation=90)
    plt.show()


def random_forest(df: pd.DataFrame, name: str, cv=5):
    y = df['Class']
    X = df.iloc[:,~df.columns.isin(['Class'])]

    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=df['Class'])

    # number of tress to produce
    num = 100
    numTrees = np.linspace(15, num, 10)
    rf = RandomForestClassifier(oob_score=True, random_state=0, n_jobs=8)

    error_array = []
    roc_auc_score_array = []
    cv_score_array = []
    accuracy_score_array = []
    # Iterate over number of trees
    print(name+" Random Forest Dataset:\n")
    print("\t\tNumber of trees in forest:")
    for i in range(len(numTrees)):
        print("\t\t {}".format(int(numTrees[i])))

        rf.set_params(n_estimators=int(numTrees[i]))

        scores = cross_validate(rf, X, y, cv=cv, scoring='roc_auc', return_train_score=True)
        cv_score_array.append(np.mean(scores['test_score']))

        rf.fit(x_train, y_train)

        y_predict = rf.predict(x_test)

        roc_auc_score_array.append(roc_auc_score(y_test, y_predict))
        accuracy_score_array.append(accuracy_score(y_pred=y_predict, y_true=y_test))
```

```
    oob_error = 1 - rf.oob_score_
    error_array.append(oob_error)


# Evaluate model performance on the test data
y_rf_pred_prob = rf.predict_proba(x_test)

results = "\t\tRandom Forest scores with {} trees\n".format(num) \
+ "\t\tAccuracy: {}\n".format("%.2f" % accuracy_score(y_pred=y_predict, y_true=y_test)) \
+ "\t\tMean Cross Validation Area Under Curve Score: {}\n".format(np.mean(cv_score_array[-1])) \
+ "\t\tRandom Forest Area Under Curve Score: {}\n".format(roc_auc_score(y_test, y_rf_pred_prob[:,1]))

print(results)

# Plot scores graph
plt.figure()
plt.title("Scores Over Number of Trees")
plt.plot(numTrees, roc_auc_score_array,'--', label="Test AUC score")
plt.plot(numTrees, cv_score_array, '.', label="Mean Cross Validation Area Under Curve Score")
plt.plot(numTrees, accuracy_score_array, label="Accuracy score")
plt.ylabel("Scores")
plt.xlabel("Number of Trees")
plt.legend(loc="best")
plt.show()


# Plot OOB error graph
plt.figure()
plt.title("Out-of-bag Error Over Number of Trees")
plt.plot(numTrees, error_array)
plt.ylabel("Out-of-bag Error")
plt.xlabel("Number of Trees")
plt.show()


# Show coeffecient weights, possible feature importance
plt.figure()
plt.title("Potential Feature Importance")
plt.bar([x for x in X.columns], abs(rf.feature_importances_))
plt.xticks(rotation=90)
plt.ylabel("Coeffecient Value")
plt.show()
```

```
    fpr, tpr, _ = roc_curve(y_test, y_rf_pred_prob[:,1])

    # Plot FalsePositiveRate to TruePositiveRate
    plt.figure()
    plt.title("TP vs FP Rate")
    plt.plot(fpr,tpr)
    plt.ylabel("True Positive Rate")
    plt.xlabel("False Positive Rate")
    plt.show()



print("Breast Cancer Wisconsin Dataset!\n")
# Find duplicates and remove them
beforeCount = len(df)
df = df[~df.duplicated()]
afterCount = len(df)
print("Removed {} duplicates from the dataset!\n".format(beforeCount - afterCount))

# Find feature importance
find_contigency(df)
# This has no significance to the data, time to drop
df.drop("Sample code number", axis=1, inplace=True)

# Lets change the Class' column values from 2 and 4 to 0 - 1
df["Class"] = df["Class"].replace(2, 0)
df["Class"] = df["Class"].replace(4, 1)

# Create two different datasets, one clean and the other dirty
df_dirty = df.copy()
knn_imp = KNNImputer(n_neighbors=10, weights='distance', metric='nan_euclidean')
df_dirty["Bare Nuclei"] = knn_imp.fit_transform(df_dirty)

df_clean = df.copy().dropna()

find_outliers("Dirty Dataset", df_dirty, False)
find_outliers("Clean Dataset", df_clean, True)

linear_regression(df_dirty, "Dirty")
linear_regression(df_clean, "Clean")

logistic_regression(df_dirty, "Dirty")
logistic_regression(df_clean, "Clean")

random_forest(df_dirty, "Dirty")
```

```
random_forest(df_clean, "Clean")



# Since the dirty dataset seems to perform worse, maybe we can create a feature to increase model performance

# Combine features and get average Uniformity of Cell Size and Uniformity of Cell Shape
df_dirty["CellSize+CellShape+BareNuclei/3"] = (df_dirty["Uniformity of Cell Size"] + df_dirty["Uniformity of Cell Shape"] + df_dirty["Bare Nuclei"]) / 3

logistic_regression(df_dirty, "Dirty New Feature")
random_forest(df_dirty, "Dirty New Feature")

# It was a success with the Dirty dataset, maybe add the new feature to the clean dataset
df_clean["CellSize+CellShape+BareNuclei/3"] = (df_clean["Uniformity of Cell Size"] + df_clean["Uniformity of Cell Shape"] + df_clean["Bare Nuclei"]) / 3

logistic_regression(df_dirty, "Clean New Feature")
random_forest(df_dirty, "Clean New Feature")
```