project

August 22, 2025

1 Breast Cancer Prediction Project

This project is the result of a general classification problem - determining whether a female has breast cancer or not.

[ ]:
```

```

2 Importing Project Tools and Libraries

Getting all the necessary project tools and libraries for the project at the start is essential for a concise and efficient project workflow. All the libraries used in this project are mentioned and imported in the below cell.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")

np.random.seed(seed = 2)

import sklearn
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪RandomizedSearchCV, GridSearchCV
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,␣
↪recall_score, f1_score

from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer

from joblib import dump, load
```

[ ]:

```

```

3 Fetching the Dataset

The dataset to be fed to the respective Machine Learning Models is fetched. Notice that it is part of the built-in dataset as provided by sklearn. It is fetched in the form of a python bunch object and is thus shifted into a Pandas DataFrame

[2]:
```
obj = load_breast_cancer()

dataFrame = pd.DataFrame(obj.data, columns = obj.feature_names)

dataFrame["target"] = obj.target

dataFrame
```

[2]:
```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          17.99         10.38          122.80     1001.0          0.11840
1          20.57         17.77          132.90     1326.0          0.08474
2          19.69         21.25          130.00     1203.0          0.10960
3          11.42         20.38           77.58      386.1          0.14250
4          20.29         14.34          135.10     1297.0          0.10030
..           …             …               …          …                …
564        21.56         22.39          142.00     1479.0          0.11100
565        20.13         28.25          131.20     1261.0          0.09780
566        16.60         28.08          108.30      858.1          0.08455
567        20.60         29.33          140.10     1265.0          0.11780
568         7.76         24.54           47.92      181.0          0.05263

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.27760         0.30010              0.14710         0.2419
1             0.07864         0.08690              0.07017         0.1812
2             0.15990         0.19740              0.12790         0.2069
3             0.28390         0.24140              0.10520         0.2597
4             0.13280         0.19800              0.10430         0.1809
..                …               …                    …             …
564           0.11590         0.24390              0.13890         0.1726
565           0.10340         0.14400              0.09791         0.1752
566           0.10230         0.09251              0.05302         0.1590
567           0.27700         0.35140              0.15200         0.2397
568           0.04362         0.00000              0.00000         0.1587
```

```
     mean fractal dimension  …  worst texture  worst perimeter  worst area  \
0                   0.07871  …          17.33           184.60      2019.0
1                   0.05667  …          23.41           158.80      1956.0
2                   0.05999  …          25.53           152.50      1709.0
3                   0.09744  …          26.50            98.87       567.7
4                   0.05883  …          16.67           152.20      1575.0
..                      …  …            …                …           …
564                 0.05623  …          26.40           166.10      2027.0
565                 0.05533  …          38.25           155.00      1731.0
566                 0.05648  …          34.12           126.70      1124.0
567                 0.07016  …          39.42           184.60      1821.0
568                 0.05884  …          30.37            59.16       268.6

     worst smoothness  worst compactness  worst concavity  \
0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                …                  …                …
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     worst concave points  worst symmetry  worst fractal dimension  target
0                  0.2654          0.4601                  0.11890       0
1                  0.1860          0.2750                  0.08902       0
2                  0.2430          0.3613                  0.08758       0
3                  0.2575          0.6638                  0.17300       0
4                  0.1625          0.2364                  0.07678       0
..                    …             …                      …     …
564                0.2216          0.2060                  0.07115       0
565                0.1628          0.2572                  0.06637       0
566                0.1418          0.2218                  0.07820       0
567                0.2650          0.4087                  0.12400       0
568                0.0000          0.2871                  0.07039       1

[569 rows x 31 columns]
```

[ ]:

4 Getting insights from the Dataset

The first and most important step is to find out the shape and form in which the dataset has been presented to us. It involves things like checking for missing values, analyzing data types and sample size, analyzing the cardinalities of attributes, determining attribute relationships etc.

[3]:
```
sum(dataFrame.isna().sum())
```

[3]:
```
0
```

[4]:
```
correlation = dataFrame.corr()

correlation.head()
```

[4]:
```
                 mean radius  mean texture  mean perimeter  mean area  \
mean radius         1.000000      0.323782        0.997855   0.987357
mean texture        0.323782      1.000000        0.329533   0.321086
mean perimeter      0.997855      0.329533        1.000000   0.986507
mean area           0.987357      0.321086        0.986507   1.000000
mean smoothness     0.170581     -0.023389        0.207278   0.177028

                 mean smoothness  mean compactness  mean concavity  \
mean radius             0.170581          0.506124        0.676764
mean texture           -0.023389          0.236702        0.302418
mean perimeter          0.207278          0.556936        0.716136
mean area               0.177028          0.498502        0.685983
mean smoothness         1.000000          0.659123        0.521984

                 mean concave points  mean symmetry  mean fractal dimension  \
mean radius                 0.822529       0.147741               -0.311631
mean texture                0.293464       0.071401               -0.076437
mean perimeter              0.850977       0.183027               -0.261477
mean area                   0.823269       0.151293               -0.283110
mean smoothness             0.553695       0.557775                0.584792

                 …  worst texture  worst perimeter  worst area  \
mean radius      …       0.297008         0.965137    0.941082
mean texture     …       0.912045         0.358040    0.343546
mean perimeter   …       0.303038         0.970387    0.941550
mean area        …       0.287489         0.959120    0.959213
mean smoothness  …       0.036072         0.238853    0.206718

                 worst smoothness  worst compactness  worst concavity  \
mean radius              0.119616           0.413463         0.526911
mean texture             0.077503           0.277830         0.301025
mean perimeter           0.150549           0.455774         0.563879
mean area                0.123523           0.390410         0.512606
mean smoothness          0.805324           0.472468         0.434926

                 worst concave points  worst symmetry  \
mean radius                  0.744214        0.163953
mean texture                 0.295316        0.105008
```

```
mean perimeter                0.771241        0.189115
mean area                     0.722017        0.143570
mean smoothness               0.503053        0.394309

                  worst fractal dimension    target
mean radius                      0.007066 -0.730029
mean texture                     0.119205 -0.415185
mean perimeter                   0.051019 -0.742636
mean area                        0.003738 -0.708984
mean smoothness                  0.499316 -0.358560

[5 rows x 31 columns]
```

[ ]:

4.0.1 Eliminating useless columns as interpreted from correlation

[5]:

```python
frameCols = [att for att in dataFrame]

col = len(correlation) - 1

for i in range(0, len(correlation)):
    if abs(correlation[frameCols[i]]["target"]) <= 0.2:
        dataFrame = dataFrame.drop({frameCols[i]}, axis = 1)
```

[6]:

```python
dataFrame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 26 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   radius error             569 non-null    float64
 10  perimeter error          569 non-null    float64
 11  area error               569 non-null    float64
 12  compactness error        569 non-null    float64
 13  concavity error          569 non-null    float64
 14  concave points error     569 non-null    float64
 15  worst radius             569 non-null    float64
```

```
 16  worst texture           569 non-null    float64
 17  worst perimeter         569 non-null    float64
 18  worst area              569 non-null    float64
 19  worst smoothness        569 non-null    float64
 20  worst compactness       569 non-null    float64
 21  worst concavity         569 non-null    float64
 22  worst concave points    569 non-null    float64
 23  worst symmetry          569 non-null    float64
 24  worst fractal dimension 569 non-null    float64
 25  target                  569 non-null    int64
dtypes: float64(25), int64(1)
memory usage: 115.7 KB
```

[7]:

```
correlation = dataFrame.corr()

correlation.head()
```

[7]:

```
                 mean radius  mean texture  mean perimeter  mean area  \
mean radius         1.000000      0.323782        0.997855   0.987357
mean texture        0.323782      1.000000        0.329533   0.321086
mean perimeter      0.997855      0.329533        1.000000   0.986507
mean area           0.987357      0.321086        0.986507   1.000000
mean smoothness     0.170581     -0.023389        0.207278   0.177028

                 mean smoothness  mean compactness  mean concavity  \
mean radius             0.170581          0.506124        0.676764
mean texture           -0.023389          0.236702        0.302418
mean perimeter          0.207278          0.556936        0.716136
mean area               0.177028          0.498502        0.685983
mean smoothness         1.000000          0.659123        0.521984

                 mean concave points  mean symmetry  radius error  …  \
mean radius                 0.822529       0.147741      0.679090  …
mean texture                0.293464       0.071401      0.275869  …
mean perimeter              0.850977       0.183027      0.691765  …
mean area                   0.823269       0.151293      0.732562  …
mean smoothness             0.553695       0.557775      0.301467  …

                 worst texture  worst perimeter  worst area  worst smoothness  \
mean radius           0.297008         0.965137    0.941082          0.119616
mean texture          0.912045         0.358040    0.343546          0.077503
mean perimeter        0.303038         0.970387    0.941550          0.150549
mean area             0.287489         0.959120    0.959213          0.123523
mean smoothness       0.036072         0.238853    0.206718          0.805324

                 worst compactness  worst concavity  worst concave points  \
mean radius               0.413463         0.526911              0.744214
```

```
mean texture                 0.277830         0.301025              0.295316
mean perimeter               0.455774         0.563879              0.771241
mean area                    0.390410         0.512606              0.722017
mean smoothness              0.472468         0.434926              0.503053

                 worst symmetry  worst fractal dimension    target
mean radius            0.163953                 0.007066 -0.730029
mean texture           0.105008                 0.119205 -0.415185
mean perimeter         0.189115                 0.051019 -0.742636
mean area              0.143570                 0.003738 -0.708984
mean smoothness        0.394309                 0.499316 -0.358560

[5 rows x 26 columns]
```

[ ]:

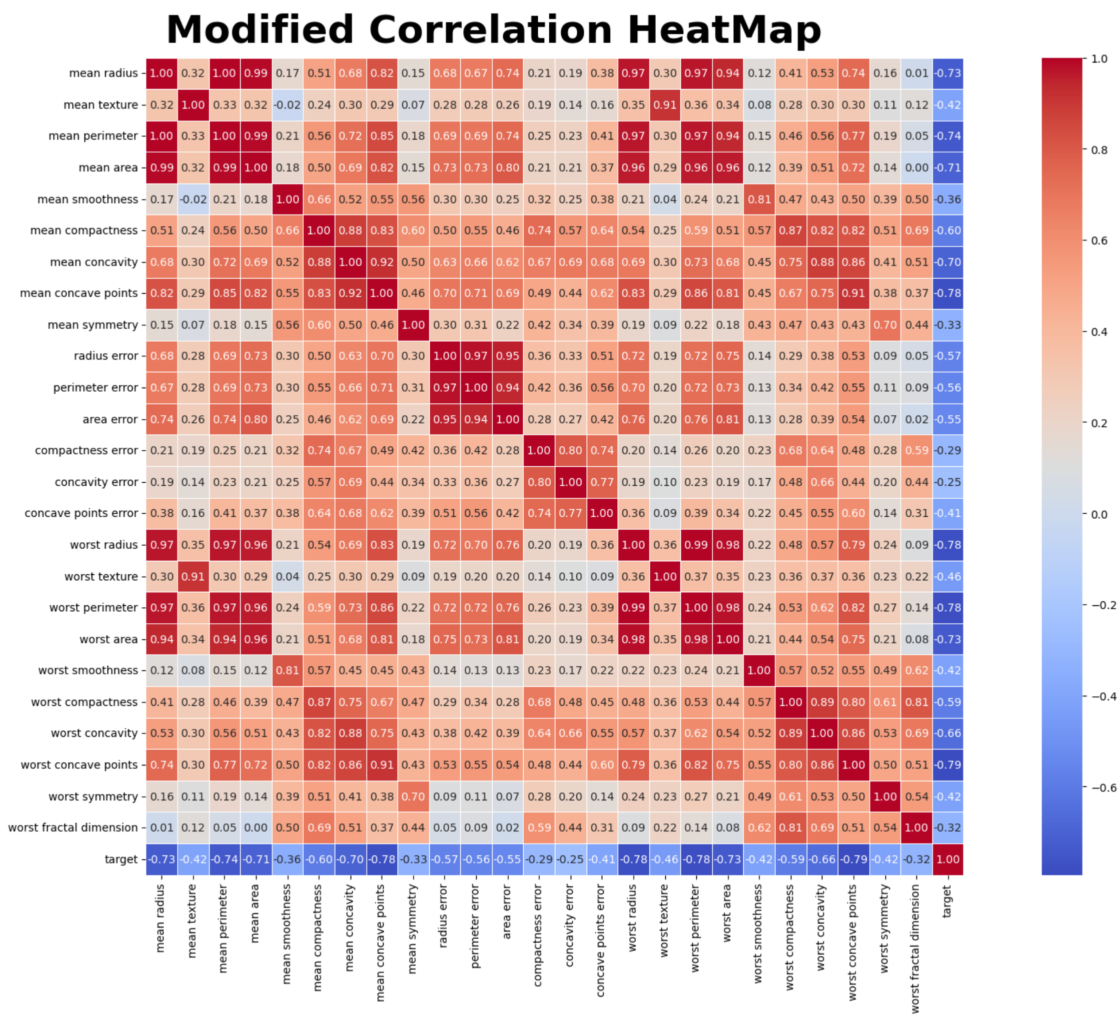
5 Analyzing the Modified Correlation through a HeatMap

A correlation is responsible for showing the type of relationship attributes have with each other (can be one of either two). It also shows relationship between attributes (features) and the column that needs to be predicted (target)

In other words, correlation represents the degree of relationship between variables (features)

This degree can be either: - Positive Degree (when both attributes have a positive correlation - this means that they are directly proportional) - Negative Degree (when either attribute (or both) have a negative correlation - this means that they are inversely proportional)

[8]:
```
fig, plot = plt.subplots(figsize = (20, 13))

fig.suptitle("Modified Correlation HeatMap", fontsize = 35, fontweight = "bold")

sns.heatmap(
    correlation,
    annot = True,
    fmt = ".2f",
    square = True,
    cbar = True,
    linewidths = 0.5,
    cmap = "coolwarm"
)

fig.savefig("Correlation HeatMap.png")

plt.tight_layout()
plt.show()
```



Strongest Features having influence on Target

- worst concave points
- worst perimeter
- worst radius
- worst area
- mean concave points
- mean perimeter
- mean radius

```
[ ]:
```

6 Splitting the Data into X & Y

The features and the target variable are split into separate frames so as to prepare the features specifically for the data preprocessing phase

[9]:
```
x, y = dataFrame.drop("target", axis = 1), dataFrame["target"]

y = pd.DataFrame(y, columns = ["target"])
```

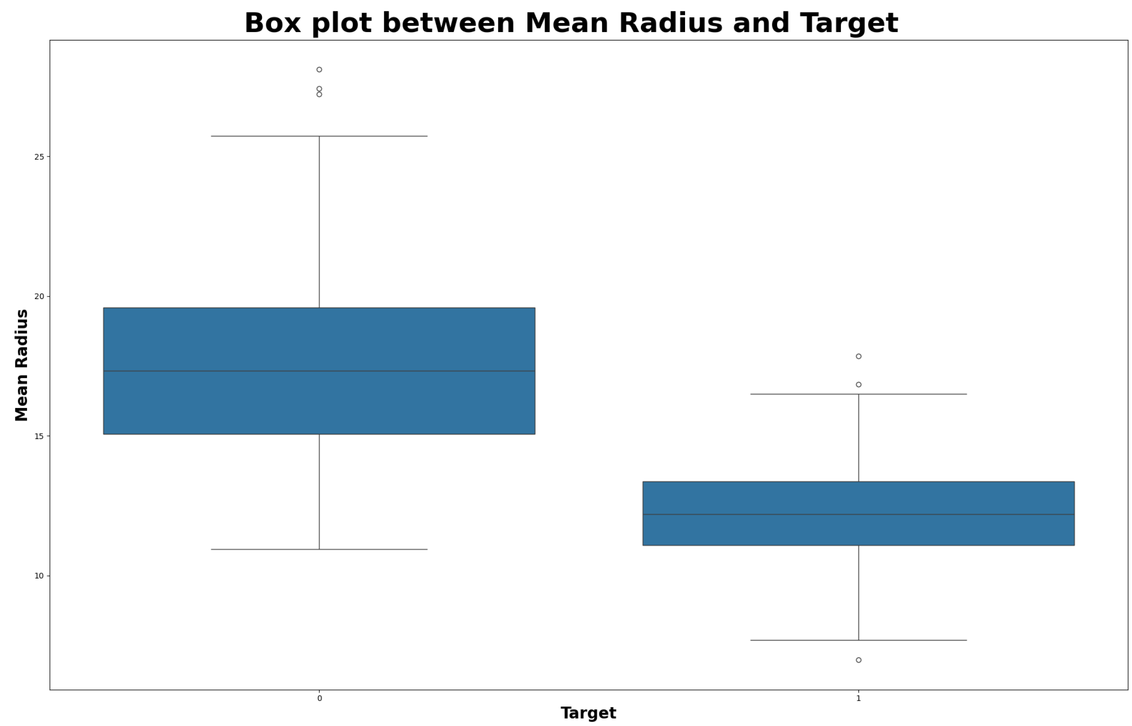
[ ]:
```

```

7 Analyzing Effect of a few Features on prediction of Target through BoxPlot

As earlier mentioned, a correlation between two variables explains the relationship between them. In this case, it is important to analyze the relationship of a few features with the target column to determine as to how a feature is going to help some machine learning model in predicting each value for the target column

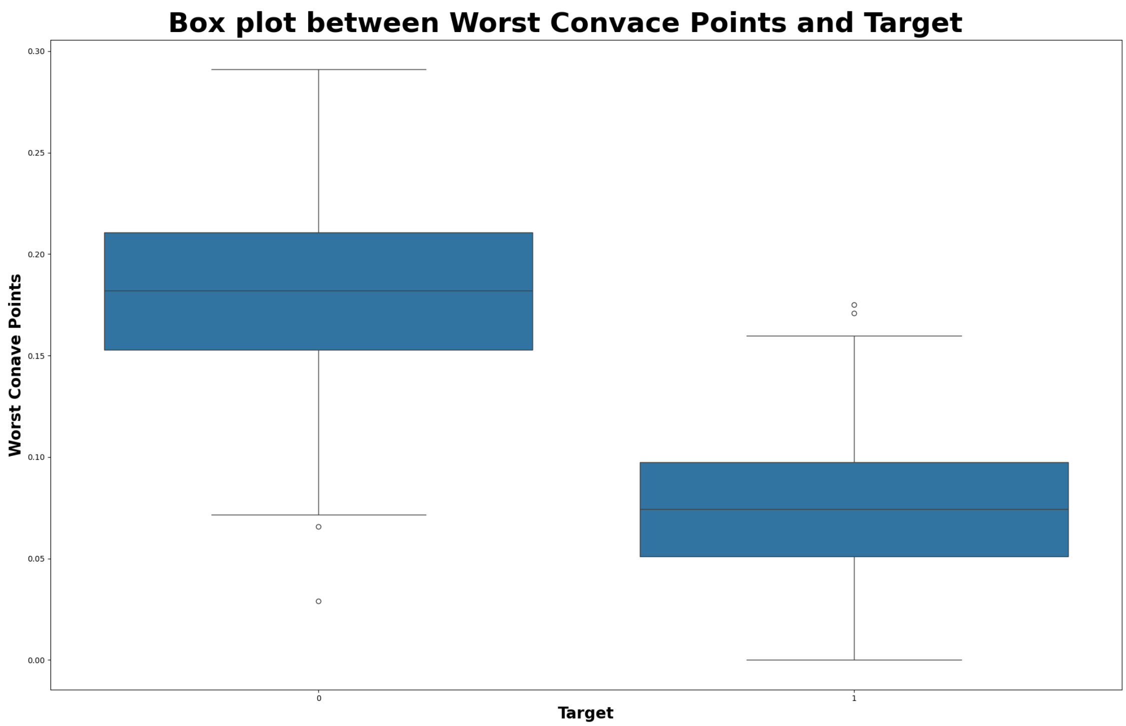
The box-plot below is responsible for showing how one feature helps in prediction one class of the target column. The more the two boxes overlap, the more confusion the model can have in distinguishing classes (it is optimal to have no overlaps). The smaller the size of a box, the more the variability of data for that feature meaning less consistency (smaller box sizes are preferred for lesser variability and more data consistency). The small dots below and above the lower and upper whiskers represent data outliers (they can be any invalid data entry, useless or meaningless information for that feature)

[10]:
```
fig, plot = plt.subplots(figsize = (20, 13))

sns.boxplot(x = "target", y = "mean radius", data = dataFrame, ax = plot)

fig.suptitle("Box plot between Mean Radius and Target", fontsize = 35,␣
 ↪fontweight = "bold")
plot.set_xlabel("Target", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Mean Radius", fontsize = 20, fontweight = "bold")

plt.tight_layout()
plt.show()
```

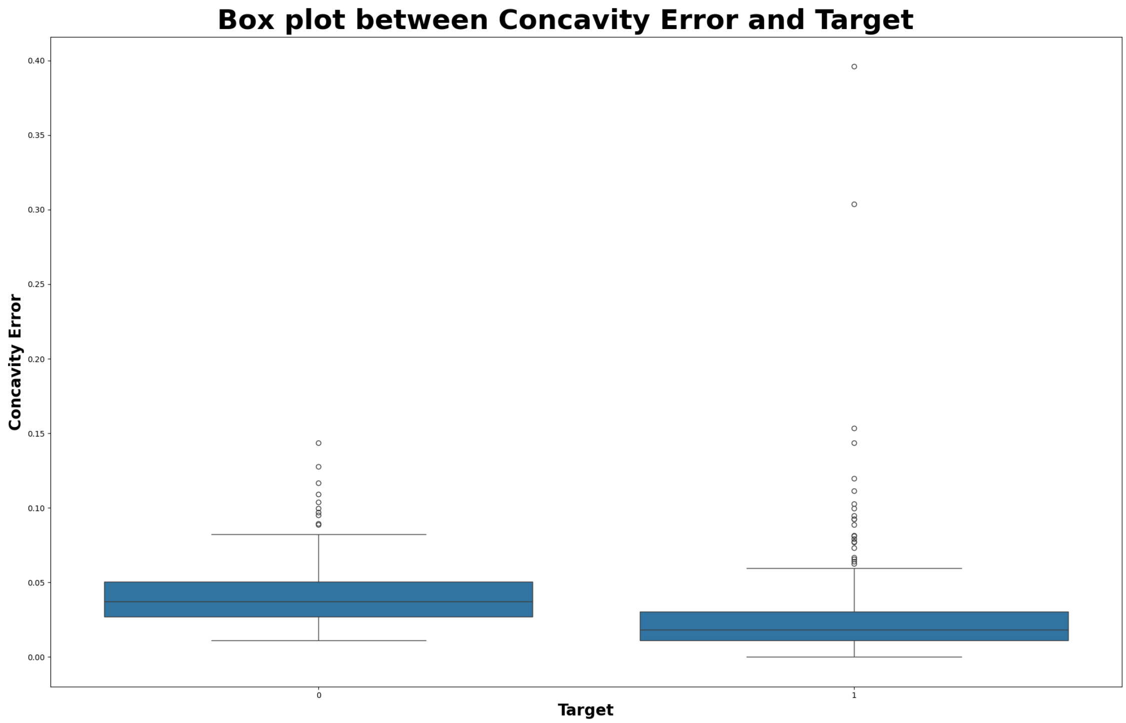


[11]:

```
fig, plot = plt.subplots(figsize = (20, 13))

sns.boxplot(x = "target", y = "worst concave points", data = dataFrame, ax = plot)

fig.suptitle("Box plot between Worst Convace Points and Target", fontsize = 35, fontweight = "bold")
plot.set_xlabel("Target", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Worst Conave Points", fontsize = 20, fontweight = "bold")

plt.tight_layout()
plt.show()
```




```
[12]: fig, plot = plt.subplots(figsize = (20, 13))

sns.boxplot(x = "target", y = "concavity error", data = dataFrame, ax = plot)

fig.suptitle("Box plot between Concavity Error and Target", fontsize = 35,␣
 ↪fontweight = "bold")
plot.set_xlabel("Target", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Concavity Error", fontsize = 20, fontweight = "bold")

plt.tight_layout()
plt.show()
```




```
[ ]:
```

8 Setting up the Data Processing Pipeline Workflow

8.0.1 Using Pipeline to implement imputation, column transformers and ML models

```
[13]: numericCols = [att for att in x]

numericTransformer = Pipeline(steps = [
    ("impute", SimpleImputer(strategy = "mean"))
])

preprocessor = ColumnTransformer(transformers = [
    ("numeric", numericTransformer, numericCols)
])

rfc = Pipeline(steps = [
    ("preprocessor", preprocessor),
    ("model", RandomForestClassifier())
])

svc = Pipeline(steps = [
    ("preprocessor", preprocessor),
    ("model", SVC())
```

```
])

lr = Pipeline(steps = [
    ("preprocessor", preprocessor),
    ("model", LogisticRegression())
])
```

[ ]:

9 Splitting the data into Training, Validation and Testing Sets

- Training Data = 80%
- Testing Data = 20%

[14]:
```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.20)
```

10 Training each Classification Model on the Training Set

The training set will be used to train each classification model on data (this is the data that will be responsible for the model to learn and recognize patterns at the time of testing)

[15]:
```
rfc.fit(xtrain, ytrain);
```

[16]:
```
svc.fit(xtrain, ytrain);
```

[17]:
```
lr.fit(xtrain, ytrain);
```

[ ]:

11 Creating a single Function for Calculating Classification Performance Metrics for each Model

After all 3 models have been trained on the training dataset, it is now time to actually test their performance and metrics by making predictions on unseen data (this is called the testing portion of the dataset split).

For better code efficiency and consistency, a single function has been defined so as to calculate the classification performance metrics for any respective classification machine learning model easily, just by passing a few specified parameters

[18]:
```
model_dict = {
    "Random Forest Classifier" : rfc,
    "Support Vector Classifier" : svc,
    "Logistic Regression" : lr
}
```

```python
def calculatePerformanceMetrics(models, ytest, xtest, printResults : bool) ->␣
↪[pd.DataFrame]:

    results0, results1 = [], []

    for model in models:

        ypredicted = models[model].predict(xtest)
        prec0, prec1 = precision_score(ytest, ypredicted, pos_label = 0),␣
↪precision_score(ytest, ypredicted, pos_label = 1)
        rec0, rec1 = recall_score(ytest, ypredicted, pos_label = 0),␣
↪recall_score(ytest, ypredicted, pos_label = 1)
        f1_0, f1_1 = f1_score(ytest, ypredicted, pos_label = 0),␣
↪f1_score(ytest, ypredicted, pos_label = 1)
        mean_acc = accuracy_score(ytest, ypredicted)

        results0.append([prec0, rec0, f1_0])
        results1.append([prec1, rec1, f1_1])

        if printResults:
            print(f"\nCalculating Performance Metrics for {model}:")
            print(f"\nPerformance for Class = 0")
            print(f"Precision: {prec0}\nRecall: {rec0}\nF1 Score: {f1_0}")
            print(f"\nPerformance for Class = 1")
            print(f"Precision: {prec1}\nRecall: {rec1}\nF1 Score: {f1_1}")
            print(f"\nMean Accuracy: {mean_acc}")

    resultsFrame0 = pd.DataFrame(
        [res for res in results0],
        columns = ["Precision", "Recall", "F1 Score"],
        index = list(models.keys())
    )

    resultsFrame1 = pd.DataFrame(
        [res for res in results1],
        columns = ["Precision", "Recall", "F1 Score"],
        index = list(models.keys())
    )

    return [resultsFrame0, resultsFrame1]
```

[19]:
```python
res = calculatePerformanceMetrics(model_dict, ytest, xtest, printResults = True)
```

```
Calculating Performance Metrics for Random Forest Classifier:

Performance for Class = 0
```

```
Precision: 0.9130434782608695
Recall: 0.9333333333333333
F1 Score: 0.9230769230769231

Performance for Class = 1
Precision: 0.9558823529411765
Recall: 0.9420289855072463
F1 Score: 0.948905109489051

Mean Accuracy: 0.9385964912280702

Calculating Performance Metrics for Support Vector Classifier:

Performance for Class = 0
Precision: 0.9047619047619048
Recall: 0.8444444444444444
F1 Score: 0.8735632183908046

Performance for Class = 1
Precision: 0.9027777777777778
Recall: 0.9420289855072463
F1 Score: 0.9219858156028369

Mean Accuracy: 0.9035087719298246

Calculating Performance Metrics for Logistic Regression:

Performance for Class = 0
Precision: 0.9111111111111111
Recall: 0.9111111111111111
F1 Score: 0.9111111111111111

Performance for Class = 1
Precision: 0.9420289855072463
Recall: 0.9420289855072463
F1 Score: 0.9420289855072463

Mean Accuracy: 0.9298245614035088
```

[ ]:

```

```

12 Visualizing the Classification Performance Metrics for each Model w.r.t each Class using BarPlot

It is always better to understand and interpret results and calculations visually using plots. A simple bar plot showing every models performance for each predictive class (0 or 1 - Breast Cancer Yes/No) is constructed. The 3 most important classification metrics for each model w.r.t each predictive class are also represented in the below bar plot.

[20]:
```
res = calculatePerformanceMetrics(model_dict, ytest, xtest, printResults =␣
 ↪False)
classificationReportFrame0, classificationReportFrame1 = res[0], res[1]
```

[21]:
```
classificationReportFrame0
```

[21]:

	Precision	Recall	F1 Score
Random Forest Classifier	0.913043	0.933333	0.923077
Support Vector Classifier	0.904762	0.844444	0.873563
Logistic Regression	0.911111	0.911111	0.911111

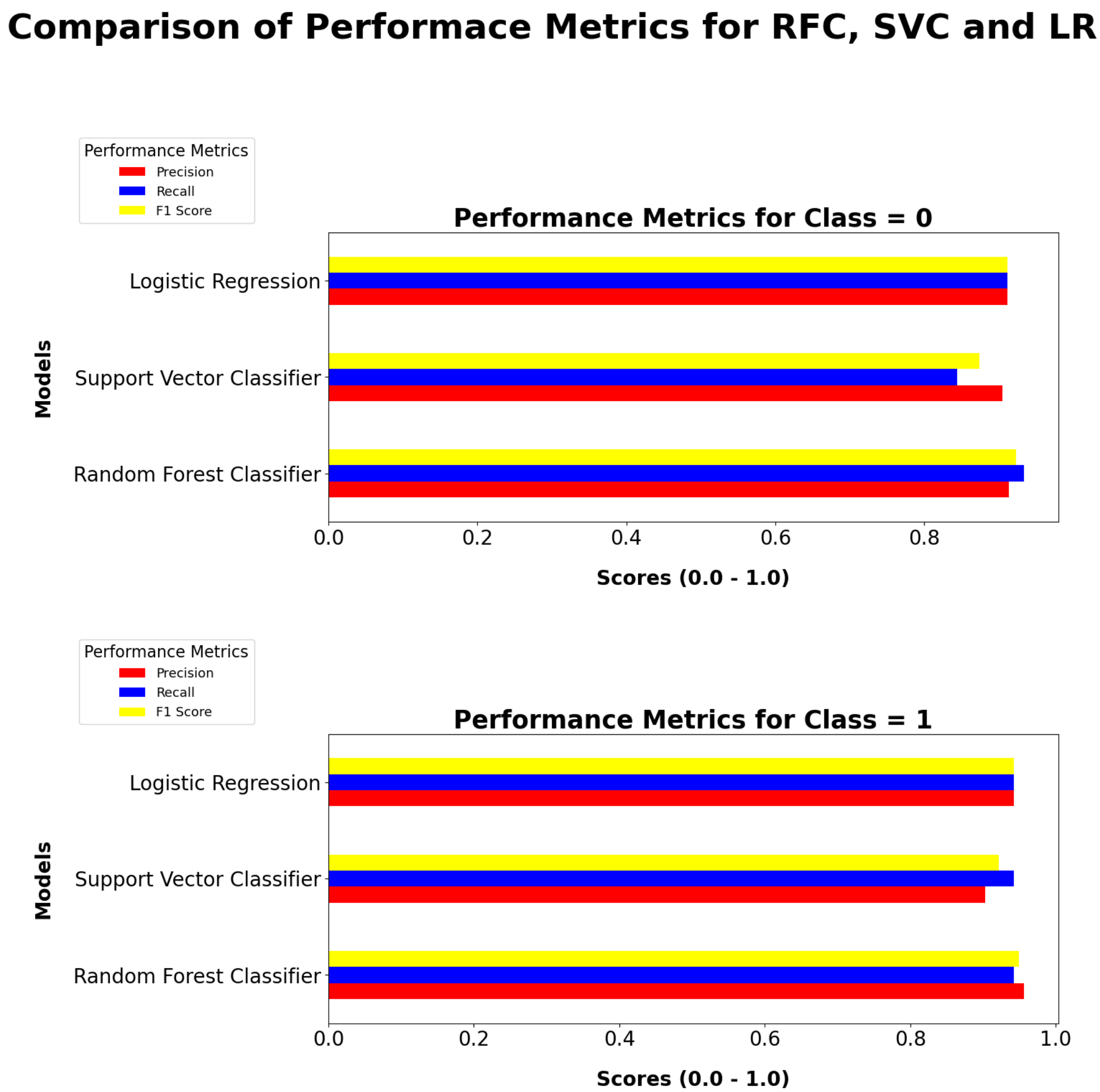
[22]:
```
classificationReportFrame1
```

[22]:

	Precision	Recall	F1 Score
Random Forest Classifier	0.955882	0.942029	0.948905
Support Vector Classifier	0.902778	0.942029	0.921986
Logistic Regression	0.942029	0.942029	0.942029

[23]:
```
fig, (plot1, plot2) = plt.subplots(2, 1, figsize = (15, 15))

classificationReportFrame0.plot(kind = "barh", ax = plot1, color = ["Red",␣
 ↪"Blue", "Yellow"])

fig.suptitle("Comparison of Performace Metrics for RFC, SVC and LR", fontsize =␣
 ↪35, fontweight = "bold", y = 1.02)

plot1.set_title("Performance Metrics for Class = 0", fontsize = 25, fontweight␣
 ↪= "bold")
plot1.set_ylabel("Models", fontsize = 20, fontweight = "bold")
plot1.set_xlabel("Scores (0.0 - 1.0)", fontsize = 20, fontweight = "bold")

plot1.xaxis.labelpad = 20
plot1.yaxis.labelpad = 20

plot1.tick_params(axis = "both", labelsize = 20)

plot1.legend(
    title = "Performance Metrics",
    fontsize = 13,
    title_fontsize = 16,
    loc = "upper left",
    bbox_to_anchor = (-0.35, 1.35)
)
```

```python
classificationReportFrame1.plot(kind = "barh", ax = plot2, color = ["Red",␣
↪"Blue", "Yellow"])

plot2.set_title("Performance Metrics for Class = 1", fontsize = 25, fontweight␣
↪= "bold")
plot2.set_ylabel("Models", fontsize = 20, fontweight = "bold")
plot2.set_xlabel("Scores (0.0 - 1.0)", fontsize = 20, fontweight = "bold")

plot2.xaxis.labelpad = 20
plot2.yaxis.labelpad = 20

plot2.tick_params(axis = "both", labelsize = 20)

plot2.legend(
    title = "Performance Metrics",
    fontsize = 13,
    title_fontsize = 16,
    loc = "upper left",
    bbox_to_anchor = (-0.35, 1.35)
)

fig.savefig("Performance Metrics BarPlot between RFC, SVC and LR.png")

plt.tight_layout(pad = 2.0)
plt.show()
```



[ ]:

```

```

13 Cross Validating the General Accuracy for each Model

Cross Validation is very important to understand in the context of interpreting a machine learning models prediction efficiency, stability and reliability over various different parts of the same dataset. We know that the model is only tested on some specified portion of the dataset split (in our case, 20% for testing data and the rest of the 80% for training data). However, sometimes, more critical and crucial data/learning patterns for the model might exist in some other portion of the complete dataset. To test the model's efficiency, consistency and reliability across the complete dataset, we use Cross Validation

In simple words, it validates the Mean Accuracy Score of a machine learning model by training and then testing it on different "folds" of the original dataset

For example, the "cv" parameters is used to determine the number of "folds" to make of the original complete dataset. Since our partition is described as 80% for training and 20% for testing, it will take 4 folds for training and then test the model on the remaining fold. Then, it will take 4 new folds and then test the model on some other fold (that might have been used as a training fold in some previous iteration)

In this way, all the folds are utilized as training and testing, one by one, based on the split and the value passed to the cv parameter in the cross validation function. This helps in better understanding the general prediction performance of a model

[24]:
```
cross_val_score(rfc, x, y, cv = 5, scoring = "accuracy")
```

[24]:
```
array([0.9122807 , 0.95614035, 0.99122807, 0.95614035, 0.97345133])
```

[25]:
```
cross_val_score(svc, x, y, cv = 5, scoring = "accuracy")
```

[25]:
```
array([0.85087719, 0.89473684, 0.92982456, 0.93859649, 0.9380531 ])
```

[26]:
```
cross_val_score(lr, x, y, cv = 5, scoring = "accuracy")
```

[26]:
```
array([0.93859649, 0.93859649, 0.96491228, 0.95614035, 0.96460177])
```

[ ]:
```

```

14 Confusion Matrix Visualization for each Classification Model using HeatMap

A confusion matrix is used for understanding precision and recall better. It can be interpreted as a point of view of the model itself and as to how it is distinguishing classes for the predictive category.

A confusion matrix is used mostly for binary classification problems (like our current one, in which we have to predict one of only two class choices). The matrix visualizes how many predictions the model is making correctly and incorrectly.

The word "confusion" means that a confusion matrix can help us understand exactly where the model is underperforming, predicting incorrect classes or being unable to efficiently distinguish binary classes from each other

The matrix represents a total of four cases among the 2x2 grid: - True Negative (0, 0): The actual class was 0 and the model correctly predicted it as a 0 - False Positive (0, 1): The actual class was 0 but the model incorrectly predicted it as a 1 (it is also called a false alarm) - False Negative (1, 0): The actual class was 1 but the model incorrectly predicted it as a 0 (wrong prediction) - True Positive (1, 1): The actual class was 1 and the model correctly predicted it as a 1

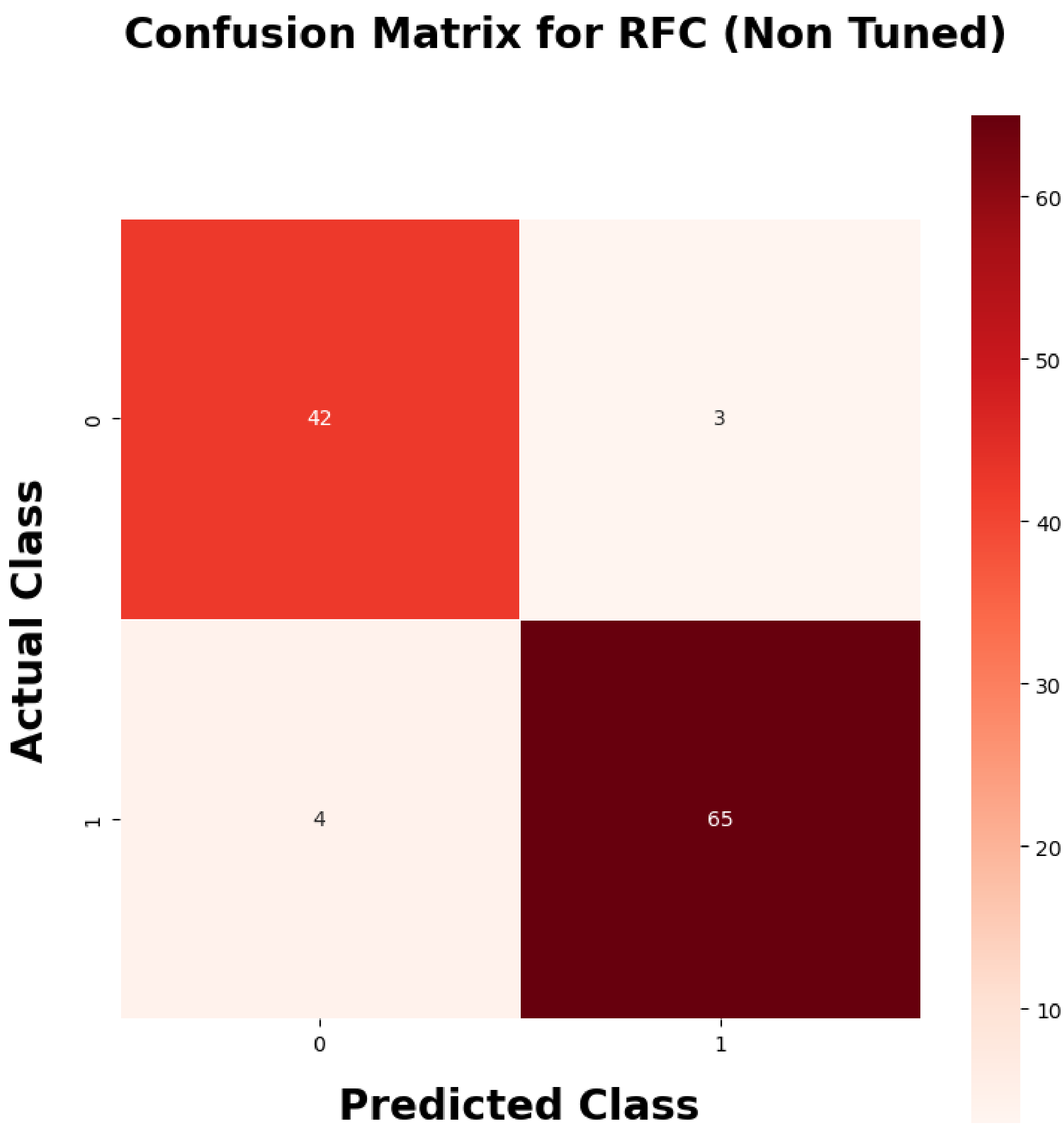
Each value inside each cell of the matrix represents the number of predictions for each case (out of all the 4 confusion cases)

After analyzing all this, it can be concluded that it is best optimal for a model to have majority of the prediction cases in the leading principle diagonal of the confusion matrix (so they fall in either True Negative or True Positive cases). Having more values in the other diagonal are not generally preferred, since they point to model predictive imbalance, wrong predictions or can even lead to class imabalance

[27]:
```
confusion_matrix(ytest, rfc.predict(xtest))
```

[27]:
```
array([[42,  3],
       [ 4, 65]])
```

[28]:
```
fig, plot = plt.subplots(figsize = (8, 8))

sns.heatmap(
    confusion_matrix(ytest, rfc.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Reds",
    cbar = True,
    square = True,
    ax = plot
)

fig.suptitle("Confusion Matrix for RFC (Non Tuned)", fontsize = 20, fontweight␣
 ↪= "bold")

plot.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")

plot.xaxis.labelpad = 15
plot.yaxis.labelpad = 15

plt.tight_layout(pad = 2.0)
plt.show()
```

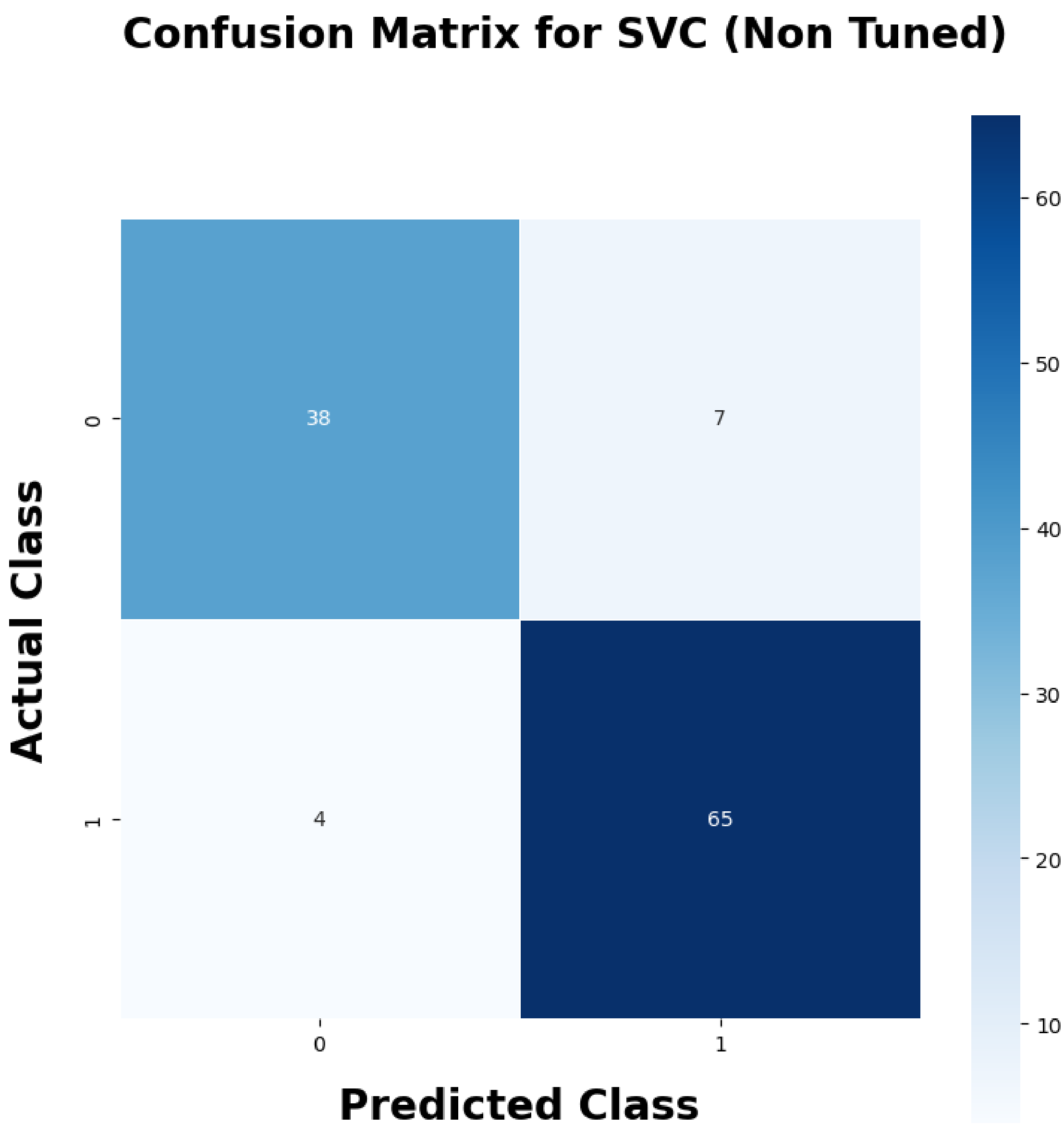


[29]:
```
confusion_matrix(ytest, svc.predict(xtest))
```

[29]:
```
array([[38,  7],
       [ 4, 65]])
```

[30]:
```
fig, plot = plt.subplots(figsize = (8, 8))

sns.heatmap(
    confusion_matrix(ytest, svc.predict(xtest)),
    annot = True,
    linewidths = 0.5,
```

```
    cmap = "Blues",
    cbar = True,
    square = True,
    ax = plot
)

fig.suptitle("Confusion Matrix for SVC (Non Tuned)", fontsize = 20, fontweight␣
 ↪= "bold")

plot.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")

plot.xaxis.labelpad = 15
plot.yaxis.labelpad = 15

plt.tight_layout(pad = 2.0)
plt.show()
```

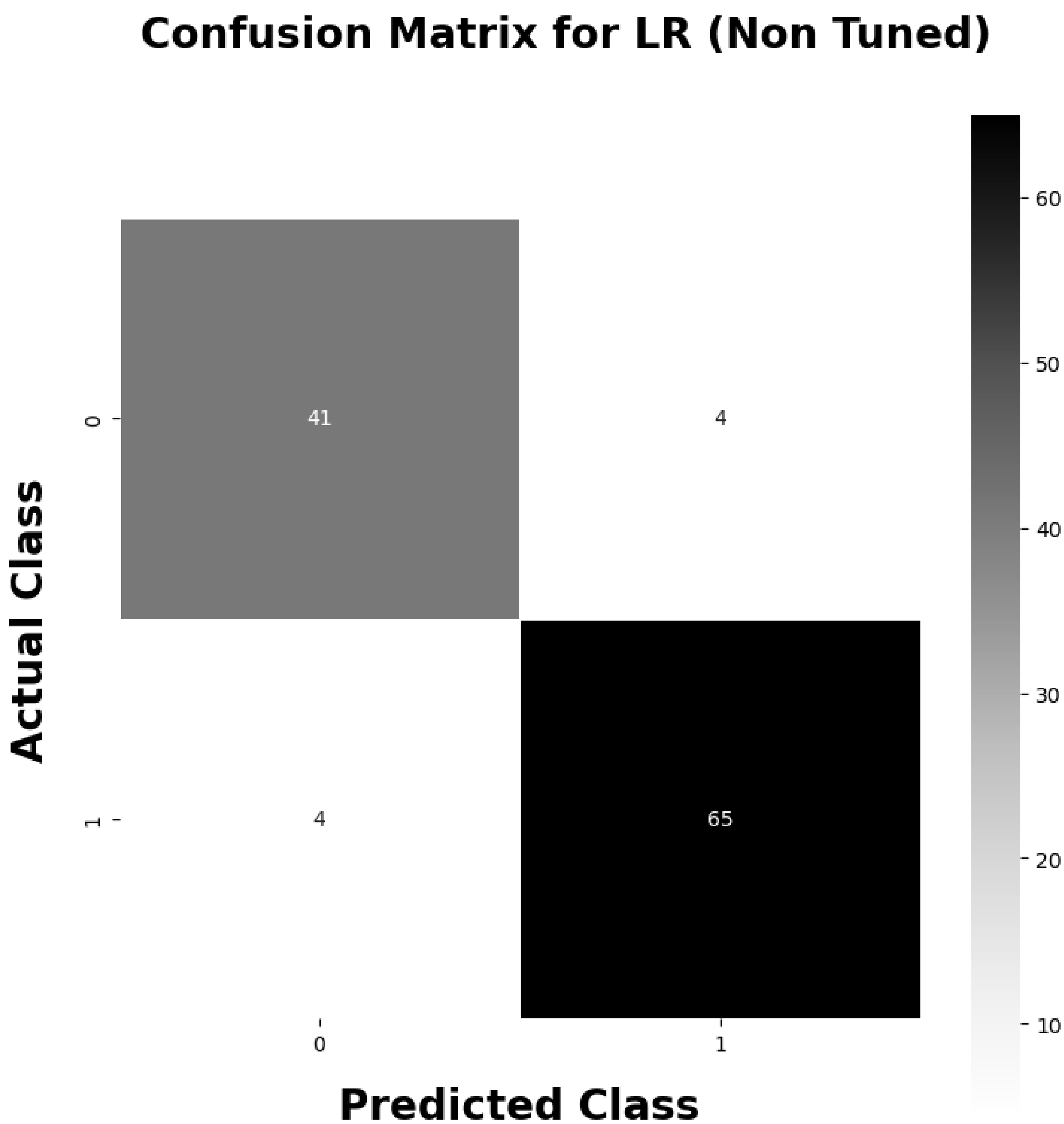


[31]:
```
confusion_matrix(ytest, lr.predict(xtest))
```

[31]:
```
array([[41,  4],
       [ 4, 65]])
```

[32]:
```
fig, plot = plt.subplots(figsize = (8, 8))

sns.heatmap(
    confusion_matrix(ytest, lr.predict(xtest)),
    annot = True,
    linewidths = 0.5,
```

```
    cmap = "Greys",
    cbar = True,
    square = True,
    ax = plot
)

fig.suptitle("Confusion Matrix for LR (Non Tuned)", fontsize = 20, fontweight =␣
 ↪"bold")

plot.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")

plot.xaxis.labelpad = 15
plot.yaxis.labelpad = 15

plt.tight_layout(pad = 2.0)
plt.show()
```




```
[ ]:
```

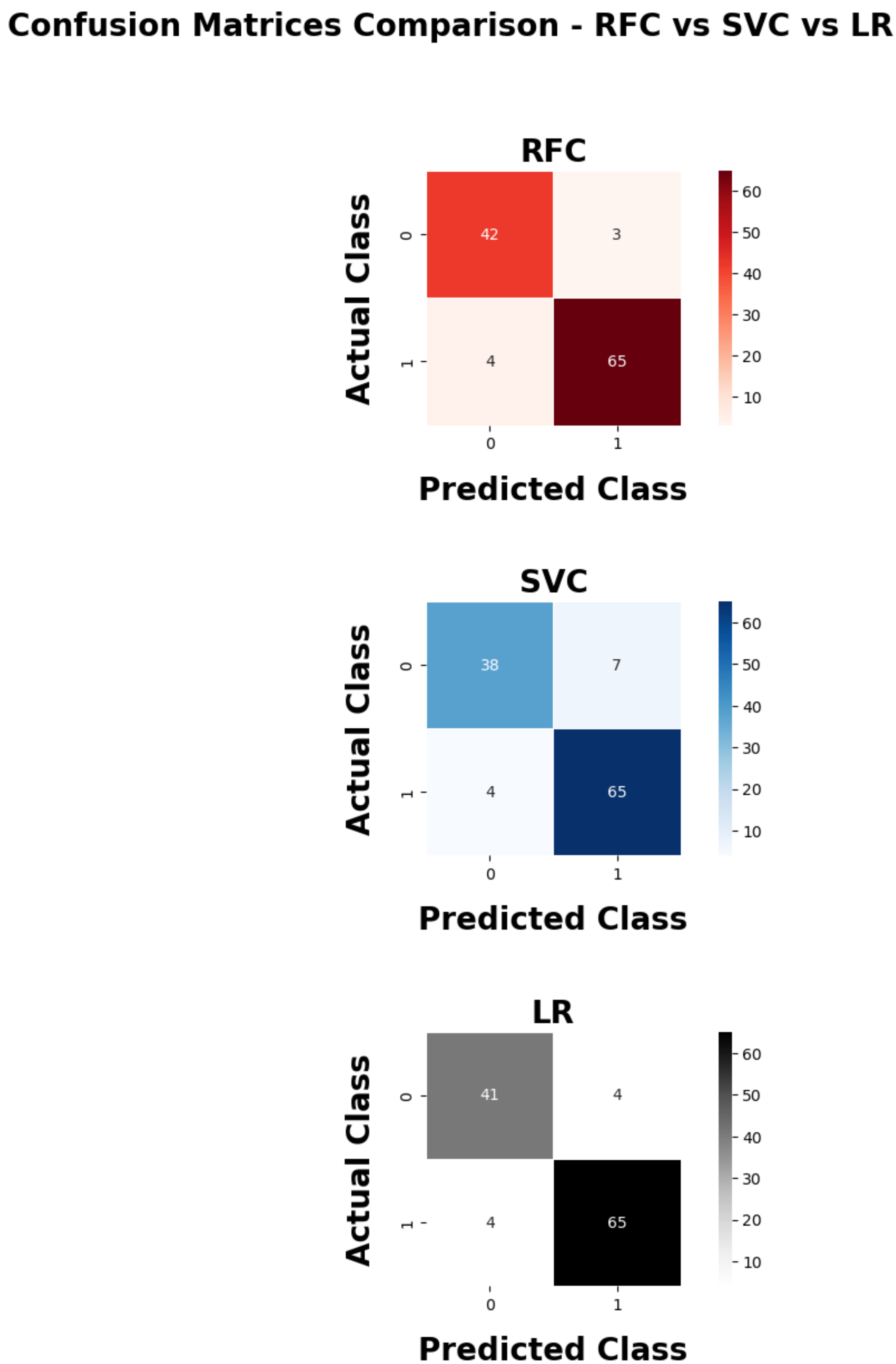
15 Comparing Confusion Matrices of all 3 Models (RFC, SVC, LR) using HeatMap

Comparing all the 3 confusion matrices for Random Forest Classifier, Support Vector Classifier and Logistic Regression

```
[33]: fig, (plot1, plot2, plot3) = plt.subplots(3, 1, figsize = (8, 13))
```

```
sns.heatmap(
    confusion_matrix(ytest, rfc.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Reds",
    cbar = True,
    square = True,
    ax = plot1
)

sns.heatmap(
    confusion_matrix(ytest, svc.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Blues",
    cbar = True,
    square = True,
    ax = plot2
)

sns.heatmap(
    confusion_matrix(ytest, lr.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Greys",
    cbar = True,
    square = True,
    ax = plot3
)

fig.suptitle("Confusion Matrices Comparison - RFC vs SVC vs LR", fontsize = 20, fontweight = "bold")

plot1.set_title("RFC", fontsize = 20, fontweight = "bold")
plot1.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot1.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot1.xaxis.labelpad = 15
plot1.yaxis.labelpad = 15

plot2.set_title("SVC", fontsize = 20, fontweight = "bold")
plot2.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot2.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot2.xaxis.labelpad = 15
plot2.yaxis.labelpad = 15

plot3.set_title("LR", fontsize = 20, fontweight = "bold")
plot3.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
```

```
plot3.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot3.xaxis.labelpad = 15
plot3.yaxis.labelpad = 15

fig.savefig("Confusion Matrix HeatMap between RFC, SVC and LR.png")

plt.tight_layout(pad = 4.0)
plt.show()
```



Confusion Matrices Comparison - RFC vs SVC vs LR
RFC
42
3
4
65
SVC
38
7
4
65
LR
41
4
4
65
Actual Class
Predicted Class
0
1
10
20
30
40
50
60

[ ]:

16 Choosing model for tuning = Random Forest Classifier

Selecting Random Forest Classifier as our choice for model tuning

[ ]:

17 Tuning Hyperparameters of Random Forest Classifier

17.1 Using method of Randomized Search Cross Validation (RSCV)

[34]:
```
rfc_rscv_params = {
    "model__max_features" : ["auto", "sqrt"],
    "model__n_estimators" : [num for num in range(100, 250, 10)],
    "model__min_samples_split" : [num for num in range(2, 6, 1)],
    "model__min_samples_leaf" : [num for num in range(1, 5, 1)],
    "model__max_depth" : [None]
}

rfc_rscv = RandomizedSearchCV(
    estimator = rfc,
    cv = 5,
    param_distributions = rfc_rscv_params,
    verbose = True,
    n_iter = 250
)
```

[35]:
```
rfc_rscv.fit(xtrain, ytrain);
```

```
Fitting 5 folds for each of 250 candidates, totalling 1250 fits
```

[ ]:

17.1.1 Evaluating the best tuned hyperparameters of RFC and best score as of tuning by RSCV

[36]:
```
rfc_rscv.best_params_, rfc_rscv.best_score_
```

[36]:
```
({'model__n_estimators': 130,
  'model__min_samples_split': 4,
  'model__min_samples_leaf': 2,
  'model__max_features': 'sqrt',
  'model__max_depth': None},
```

```
np.float64(0.9648351648351647))
```

[37]:
```
rfc_rscv_best = rfc_rscv.best_estimator_
```

[ ]:
```

```

18 Evaluating new Classification Performance Metrics for RSCV Tuned Random Forest Clasifier

[38]:
```
rfc_rscv_res = calculatePerformanceMetrics({"RSCV Tuned Random Forest␣
↪Classifier" : rfc_rscv_best}, ytest, xtest, printResults = True)
```

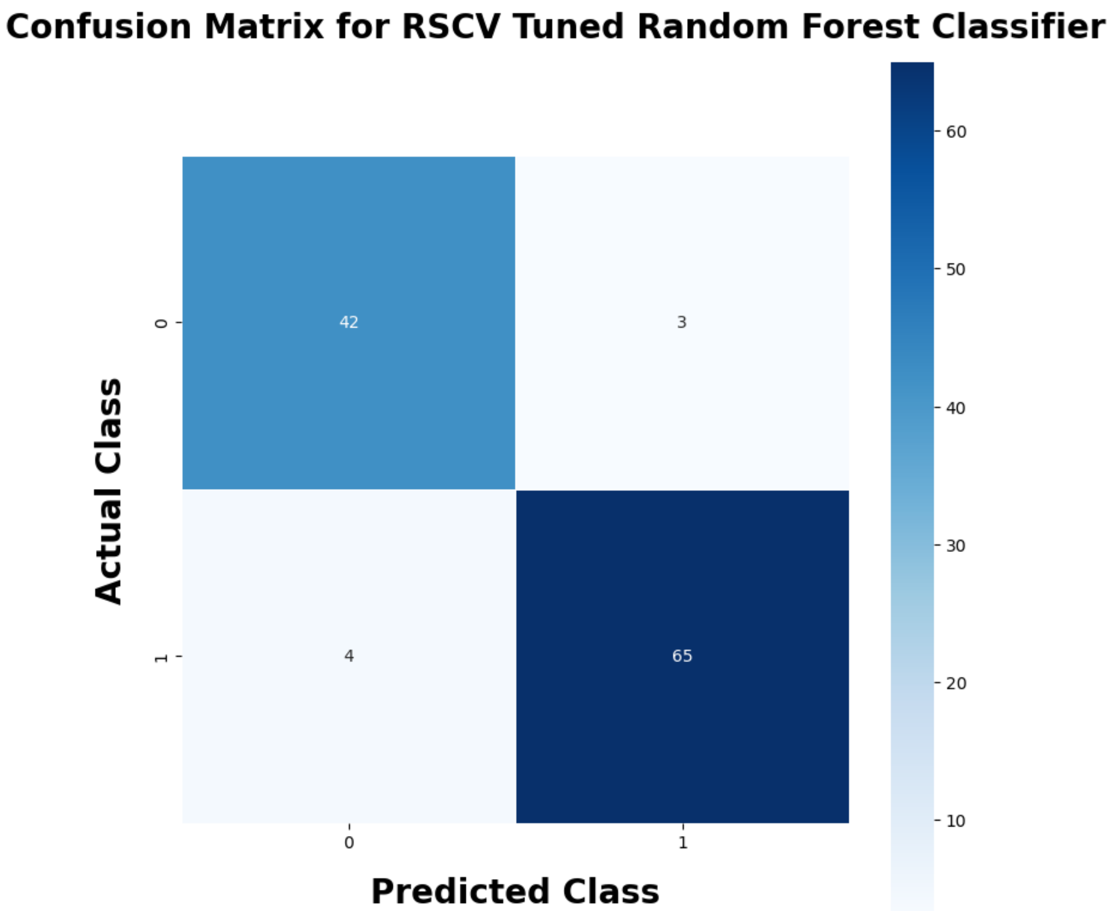
```
Calculating Performance Metrics for RSCV Tuned Random Forest Classifier:

Performance for Class = 0
Precision: 0.9130434782608695
Recall: 0.9333333333333333
F1 Score: 0.9230769230769231

Performance for Class = 1
Precision: 0.9558823529411765
Recall: 0.9420289855072463
F1 Score: 0.948905109489051

Mean Accuracy: 0.9385964912280702
```

[ ]:
```

```

19 Visualizing Confusion Matrix for RSCV Tuned Random Forest Classifier using HeatMap

[39]:
```
fig, plot = plt.subplots(figsize = (8, 8))

sns.heatmap(
    confusion_matrix(ytest, rfc_rscv_best.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Blues",
    cbar = True,
    square = True,
    ax = plot
)

fig.suptitle("Confusion Matrix for RSCV Tuned Random Forest Classifier",␣
↪fontsize = 20, fontweight = "bold")
```

```
plot.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")

plot.xaxis.labelpad = 15
plot.yaxis.labelpad = 15

plt.tight_layout()
plt.show()
```



[ ]:

```
```

19.1 Using method of Grid Search Cross Validation (GSCV)

[40]:

```
rfc_gscv_params = {
    "model__max_features" : ["sqrt"],
    "model__n_estimators" : [num for num in range(150, 200, 10)],
    "model__min_samples_split" : [num for num in range(2, 5, 1)],
```

```
    "model__min_samples_leaf" : [num for num in range(1, 4, 1)],
    "model__max_depth" : [None]
}

rfc_gscv = GridSearchCV(
    estimator = rfc,
    cv = 5,
    param_grid = rfc_gscv_params,
    verbose = True
)
```

[41]:
```
rfc_gscv.fit(xtrain, ytrain);
```

```
Fitting 5 folds for each of 45 candidates, totalling 225 fits
```

[ ]:
```

```

19.1.1 Evaluating the best tuned hyperparameters of RFC and best score as of tuning by RSCV

[42]:
```
rfc_gscv.best_params_, rfc_gscv.best_score_
```

[42]:
```
({'model__max_depth': None,
  'model__max_features': 'sqrt',
  'model__min_samples_leaf': 2,
  'model__min_samples_split': 2,
  'model__n_estimators': 170},
 np.float64(0.9626373626373625))
```

[43]:
```
rfc_gscv_best = rfc_gscv.best_estimator_
```

[ ]:
```

```

20 Evaluating new Classification Performance Metrics for GSCV Tuned Random Forest Classifier

[44]:
```
rfc_gscv_res = calculatePerformanceMetrics({"GSCV Tuned Random Forest␣
↪Classifier" : rfc_gscv_best}, ytest, xtest, printResults = True)
```

```
Calculating Performance Metrics for GSCV Tuned Random Forest Classifier:

Performance for Class = 0
Precision: 0.9148936170212766
Recall: 0.9555555555555556
F1 Score: 0.9347826086956522
```

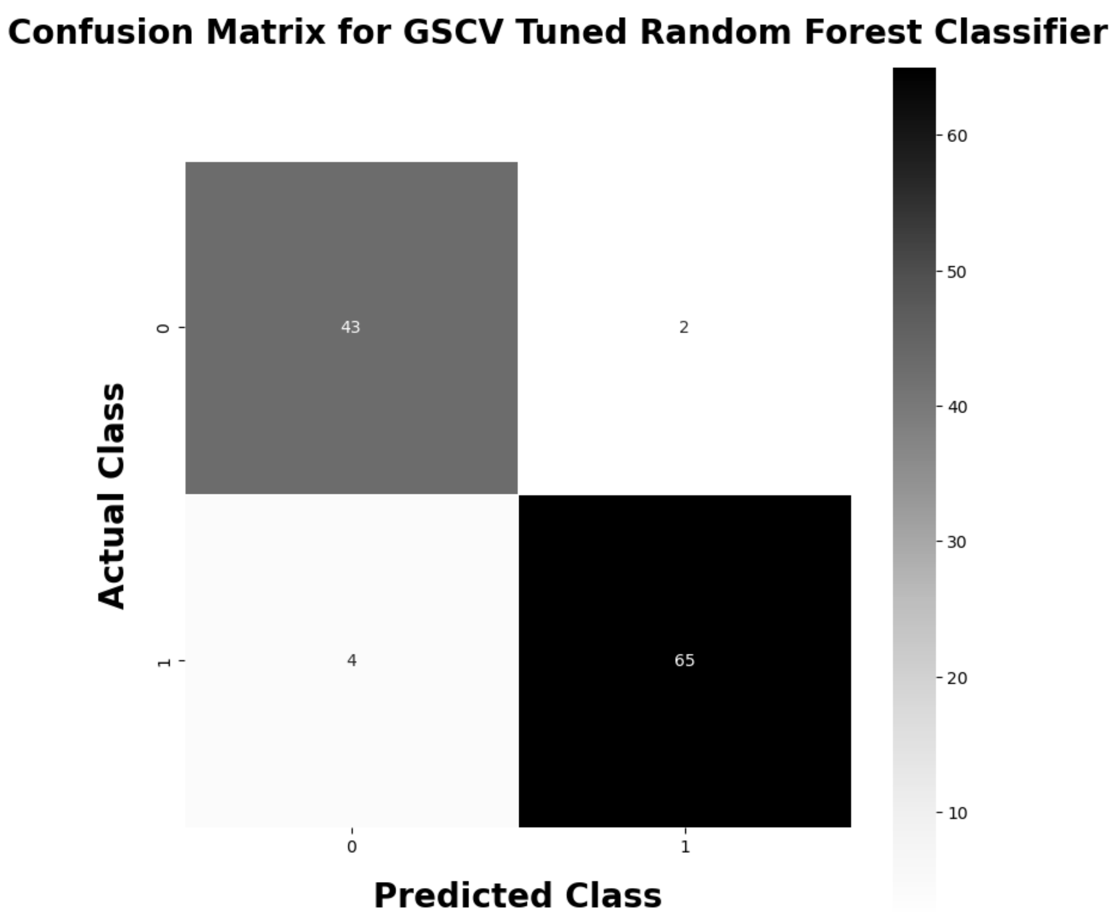
```
Performance for Class = 1
Precision: 0.9701492537313433
Recall: 0.9420289855072463
F1 Score: 0.9558823529411765

Mean Accuracy: 0.9473684210526315
```

[ ]:

21 Visualizing Confusion Matrix for GSCV Tuned Random Forest Classifier

[45]:

```
fig, plot = plt.subplots(figsize = (8, 8))

sns.heatmap(
    confusion_matrix(ytest, rfc_gscv_best.predict(xtest)),
    annot = True,
    cmap = "Greys",
    linewidths = 0.5,
    square = True,
    cbar = True,
    ax = plot
)

fig.suptitle("Confusion Matrix for GSCV Tuned Random Forest Classifier",␣
 ↪fontsize = 20, fontweight = "bold")

plot.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")

plot.xaxis.labelpad = 15
plot.yaxis.labelpad = 15

plt.tight_layout()
plt.show()
```



[ ]:

```

```

22 Comparing Confusion Matrices of all 3 versions of Random Forest Classifier (Non Tuned, RSCV Tuned, GSCV Tuned) using HeatMap

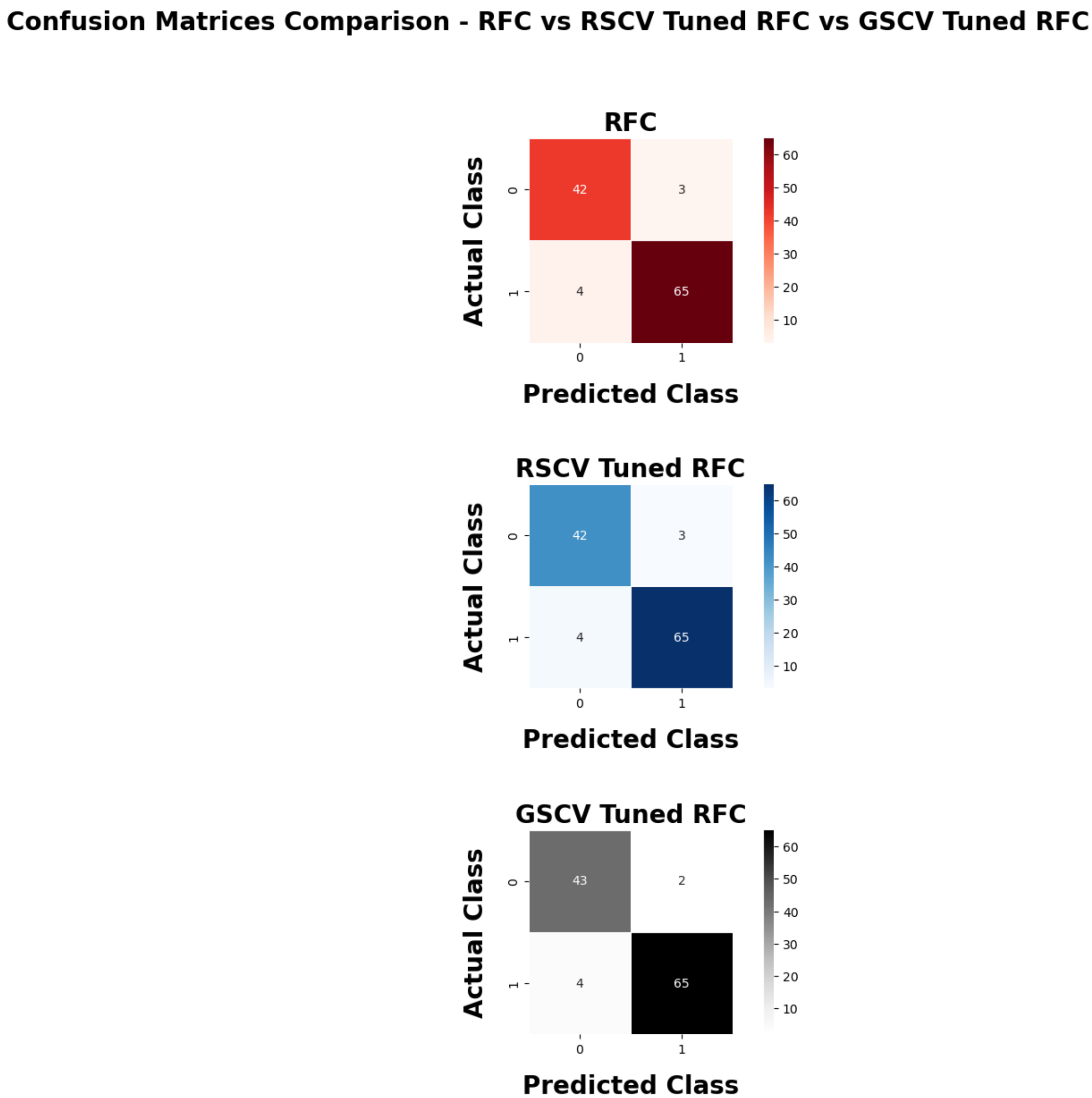
Comparing the confusion matrices for Non Tuned Random Forest Classifier, RSCV Tuned Random Forest Classifier and GSCV Tuned Random Forest Classifier

[46]:

```
fig, (plot1, plot2, plot3) = plt.subplots(3, 1, figsize = (8, 13))

sns.heatmap(
    confusion_matrix(ytest, rfc.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Reds",
    cbar = True,
```

```
    square = True,
    ax = plot1
)

sns.heatmap(
    confusion_matrix(ytest, rfc_rscv_best.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Blues",
    cbar = True,
    square = True,
    ax = plot2
)

sns.heatmap(
    confusion_matrix(ytest, rfc_gscv_best.predict(xtest)),
    annot = True,
    linewidths = 0.5,
    cmap = "Greys",
    cbar = True,
    square = True,
    ax = plot3
)

fig.suptitle("Confusion Matrices Comparison - RFC vs RSCV Tuned RFC vs GSCV␣
 ↪Tuned RFC", fontsize = 20, fontweight = "bold")

plot1.set_title("RFC", fontsize = 20, fontweight = "bold")
plot1.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot1.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot1.xaxis.labelpad = 15
plot1.yaxis.labelpad = 15

plot2.set_title("RSCV Tuned RFC", fontsize = 20, fontweight = "bold")
plot2.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot2.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot2.xaxis.labelpad = 15
plot2.yaxis.labelpad = 15

plot3.set_title("GSCV Tuned RFC", fontsize = 20, fontweight = "bold")
plot3.set_xlabel("Predicted Class", fontsize = 20, fontweight = "bold")
plot3.set_ylabel("Actual Class", fontsize = 20, fontweight = "bold")
plot3.xaxis.labelpad = 15
plot3.yaxis.labelpad = 15

fig.savefig("Confusion Matrix HeatMap between RFC, RSCV RFC and GSCV RFC.png")
```

```
plt.tight_layout(pad = 4.0)
plt.show()
```



[ ]:

```

```

23 Comparing Classification Performance Metrics for all 3 versions of Random Forest Classifier (Non Tuned, RSCV Tuned, GSCV Tuned) using BarPlot

Using a bar plot to visualize and interpret precision, recall and f1 score for all the three versions of the Random Forest Classifier after tuning

[47]:
```
rfcResults = calculatePerformanceMetrics(
    {
        "Random Forest Classifier" : rfc,
        "RSCV Random Forest Classifier" :  rfc_rscv_best,
        "GSCV Random Forest Classifier" : rfc_gscv_best
    },
    ytest,
    xtest,
    printResults = False
)
```

[48]:
```
rfcResultsFrame0, rfcResultsFrame1 = rfcResults[0], rfcResults[1]
```

[49]:
```
rfcResultsFrame0
```

[49]:

	Precision	Recall	F1 Score
Random Forest Classifier	0.913043	0.933333	0.923077
RSCV Random Forest Classifier	0.913043	0.933333	0.923077
GSCV Random Forest Classifier	0.914894	0.955556	0.934783

[50]:
```
rfcResultsFrame1
```

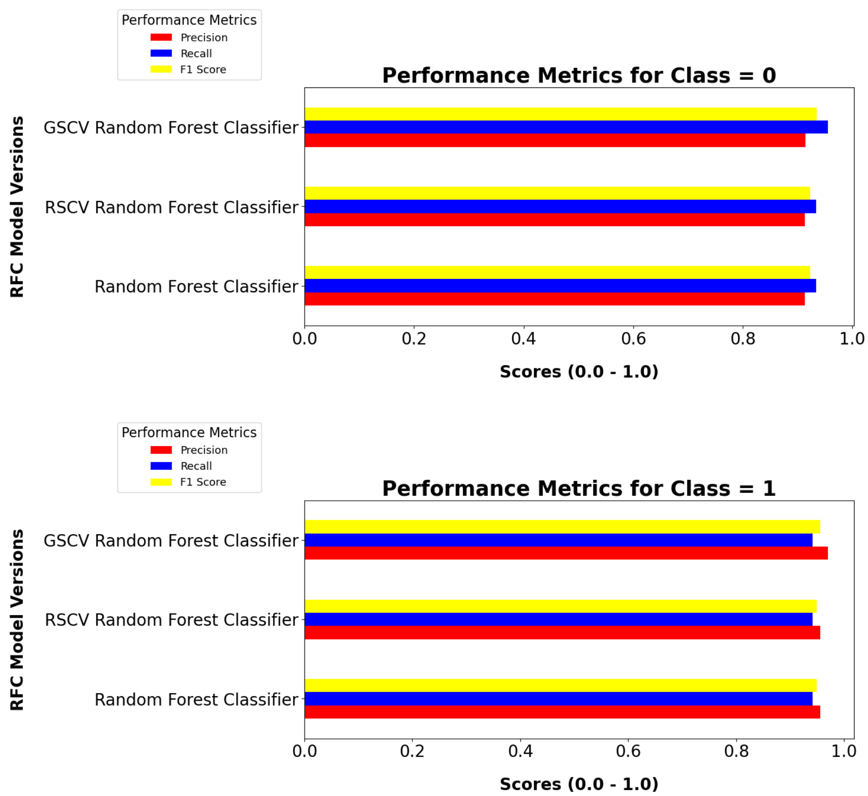
[50]:

	Precision	Recall	F1 Score
Random Forest Classifier	0.955882	0.942029	0.948905
RSCV Random Forest Classifier	0.955882	0.942029	0.948905
GSCV Random Forest Classifier	0.970149	0.942029	0.955882

[51]:
```
fig, (plot1, plot2) = plt.subplots(2, 1, figsize = (15, 15))

rfcResultsFrame0.plot(kind = "barh", ax = plot1, color = ["Red", "Blue",␣
↪"Yellow"])

fig.suptitle("Comparison of Performace Metrics for RFC, RSCV RFC and GSCV RFC",␣
↪fontsize = 35, fontweight = "bold", y = 1.02)

plot1.set_title("Performance Metrics for Class = 0", fontsize = 25, fontweight␣
↪= "bold")
plot1.set_ylabel("RFC Model Versions", fontsize = 20, fontweight = "bold")
plot1.set_xlabel("Scores (0.0 - 1.0)", fontsize = 20, fontweight = "bold")

plot1.xaxis.labelpad = 20
plot1.yaxis.labelpad = 20

plot1.tick_params(axis = "both", labelsize = 20)

plot1.legend(
    title = "Performance Metrics",
```

```
    fontsize = 13,
    title_fontsize = 16,
    loc = "upper left",
    bbox_to_anchor = (-0.35, 1.35)
)

rfcResultsFrame1.plot(kind = "barh", ax = plot2, color = ["Red", "Blue", 
 ↪"Yellow"])

plot2.set_title("Performance Metrics for Class = 1", fontsize = 25, fontweight 
 ↪= "bold")
plot2.set_ylabel("RFC Model Versions", fontsize = 20, fontweight = "bold")
plot2.set_xlabel("Scores (0.0 - 1.0)", fontsize = 20, fontweight = "bold")

plot2.xaxis.labelpad = 20
plot2.yaxis.labelpad = 20

plot2.tick_params(axis = "both", labelsize = 20)

plot2.legend(
    title = "Performance Metrics",
    fontsize = 13,
    title_fontsize = 16,
    loc = "upper left",
    bbox_to_anchor = (-0.35, 1.35)
)

fig.savefig("Performance Metrics BarPlot between RFC, RSCV RFC and GSCV RFC.
 ↪png")

plt.tight_layout(pad = 2.0)
plt.show()
```

Comparison of Performace Metrics for RFC, RSCV RFC and GSCV RFC



[ ]:

```

```

23.1 Saving the final model (after tuning) = GSCV Tuned Random Forest Classifier

After all operations, the GSCV Tuned version of Random Forest Classifier is chosen as the final and best version for this model (and also among the other two models, SVC and LR)

[52]:

```
best_model = rfc_gscv_best

dump(best_model, "best_model.joblib")
```

[52]:

```
['best_model.joblib']
```

[ ]:

```

```