Configuration Manual

MSc Research Project
Data Analytics

Temitope Oladimeji
Student ID: x23187204

School of Computing
National College of Ireland

Supervisor: Anderson Simiscuka

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Temitope Oladimeji
Student ID:	x23187204
Programme:	Data Analytics
Year:	2024
Module:	MSc Research Project
Supervisor:	Anderson Simiscuka
Submission Due Date:	16/09/2024
Project Title:	Configuration Manual
Word Count:	1415
Page Count:	16

Signature:	Temitope Oladimeji
Date:	14th September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Temitope Oladimeji
x23187204

1 Introduction

This manual is designed to provide a clear step-by-step instructions to help set up, configure, and run the machine learning models developed in this research. The primary goal of this project is to predict pregnancy-related risks using advanced machine learning techniques, specifically focusing on the implementation and evaluation of XGBoost, Random Forest, and Decision Tree model. It contains a detailed guidance on setting up the environment, preparing the dataset, running the model, and interpreting the result, structured to ensure users of different experience levels can replicate the experiment accurately and achieve similar results.

The following aspects of the project setup are covered, with each section building on the previous one from the initial setup to the final deployment:

- **System Requirements:** Detailed specifications for the hardware and software required to run the models efficiently.
- **Environment Setup:** Instructions for installing and configuring the necessary software packages and libraries.
- **Data Preparation:** Steps for collecting, cleaning, and pre-processing the dataset used in the study.
- **Model Execution:** Guidance on how to train, tune, and evaluate the machine learning models.
- **Results Interpretation:** Methods for interpreting the model outputs and visualizations to obtain any meaningful insights.
- **Deployment:** Information on exporting and deploying the trained models for practical use.

2 System Requirements

To successfully implement and run the models, the following hardware and software specifications were used:

2.1 Hardware Requirement

System	Specification
Processor	Quad-core 10th Gen Intel Core i5
RAM/Storage	8 GB/128 GB
OS	Windows 11 Home OS
GPU	Intel Iris Plus Graphics

Table 1: Minimum System Specifications

If a different operating system is to be used, an equally comparable processor and hardware features are recommended to offer a similar user experience. For example, the macOS - a MacBook Air (M1) with 8GB and 256GB SSD; Linux - a Linux OS such as Ubuntu that offers an Intel i5 processor with 8GB RAM and 128GB Storage; or Chrome OS - such as a Chromebook with an Intel i5 Processor, 8GB RAM, and 128GB SSD.

2.2 Software and Tools

Software/Tool	Version	Function
Python	3.7 or higher	Programming Language
Jupyter Notebook	Latest	Interactive Computing
NumPy	Latest	Numerical Computations
Pandas	Latest	Data Manipulation
Scikit-learn	Latest	Machine Learning Library
Matplotlib	Latest	Data Visualization
Seaborn	Latest	Data Visualization

Table 2: Software and Tools Required

3 Environment Setup

3.1 Python & Jupyter Notebook Installation

Below are the instructions for installing Python and the required software tool:

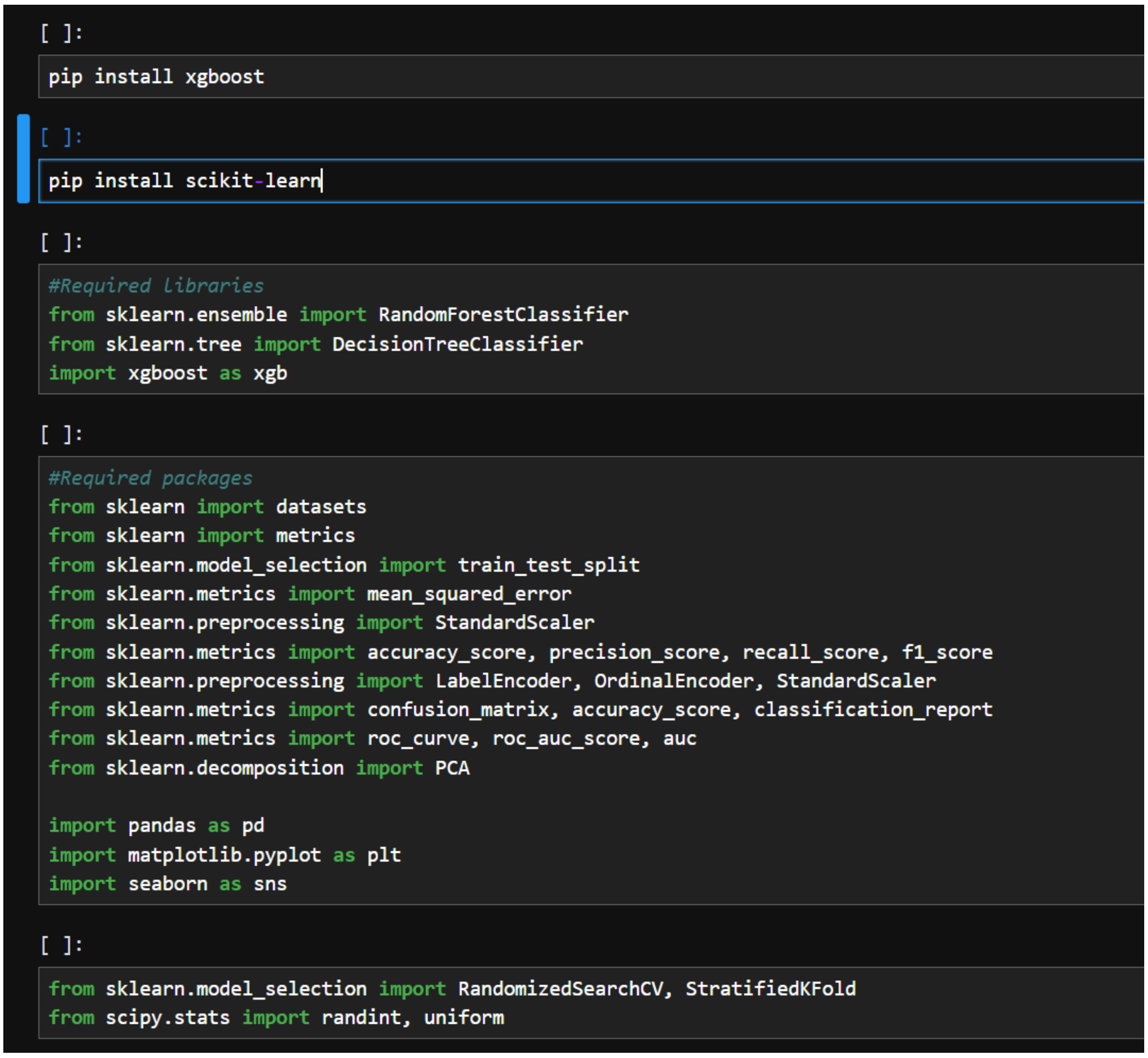
- Download and install the latest version of Python from the official website: Python Downloads.
- Follow the installation instructions for your operating system.
- Once installed, install Jupyter Notebook via pip:

```
pip install notebook
```

Once setup and installed, the Jupyter notebook can be launched by opening a terminal or command prompt and typing ”Jupyter Notebook”.

3.2 Libraries and Packages

Ensure the following Python libraries and packages are installed:



```
[ ]:
pip install xgboost

[ ]:
pip install scikit-learn

[ ]:
#Required libraries
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
import xgboost as xgb

[ ]:
#Required packages
from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder, StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
from sklearn.metrics import roc_curve, roc_auc_score, auc
from sklearn.decomposition import PCA

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

[ ]:
from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold
from scipy.stats import randint, uniform
```

Figure 1: Required Libraries

4 Data Preparation

4.1 Data Source

The dataset was sourced from the UCI Machine Learning Repository [1], consisting of 1014 records.

[1]https://archive.ics.uci.edu/dataset/863/maternal+health+risk

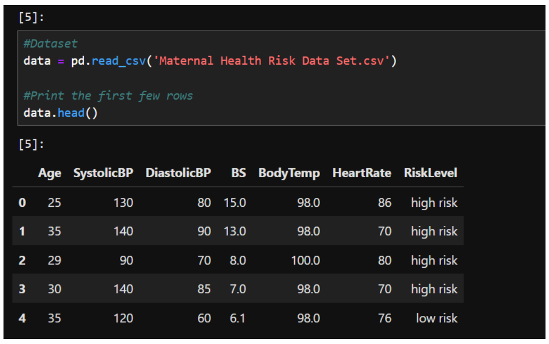
Attribute	Description
Age	The age of the mother in years.
Systolic BP	The systolic blood pressure measurement in mmHg.
Diastolic BP	The diastolic blood pressure measurement in mmHg.
Blood Sugar	Blood sugar level in mmol/L.
Temperature	Body temperature in degrees Fahrenheit.
Heart Rate	The heart rate in beats per minute.
Risk Level	The target variable, categorized into three levels (Low, Moderate, High) indicating the risk of maternal health complications.

Table 3: Description of Dataset Attributes

4.2 Data Preprocessing

4.2.1 Initial Data Exploration

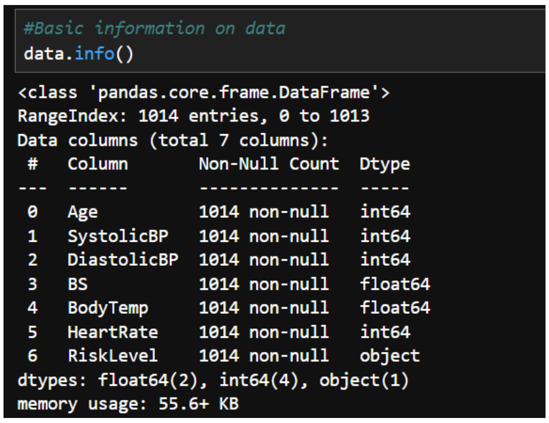
Understanding the content of the dataset is important before proceeding with any analysis. This can be done in many ways such as printing the first few rows for a quick observation (figure 2), checking the data type for each column (figure 3), or checking the summary statistics for a more detailed observation (figure 4).



```
[5]:
#Dataset
data = pd.read_csv('Maternal Health Risk Data Set.csv')

#Print the first few rows
data.head()

[5]:
```

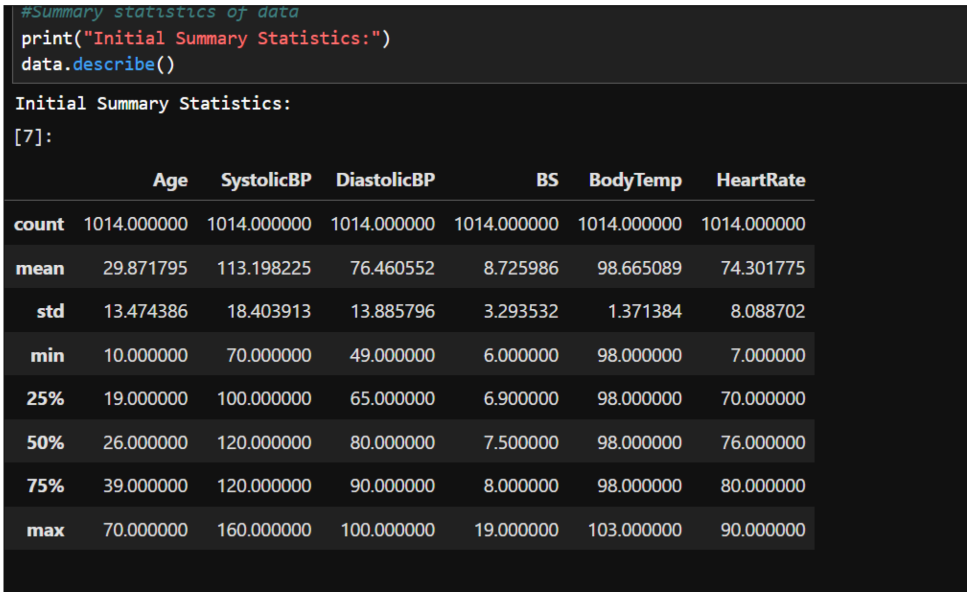
	Age	SystolicBP	DiastolicBP	BS	BodyTemp	HeartRate	RiskLevel
0	25	130	80	15.0	98.0	86	high risk
1	35	140	90	13.0	98.0	70	high risk
2	29	90	70	8.0	100.0	80	high risk
3	30	140	85	7.0	98.0	70	high risk
4	35	120	60	6.1	98.0	76	low risk

Figure 2: First few rows of dataset



```
#Basic information on data
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1014 entries, 0 to 1013
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Age          1014 non-null   int64  
 1   SystolicBP   1014 non-null   int64  
 2   DiastolicBP  1014 non-null   int64  
 3   BS           1014 non-null   float64
 4   BodyTemp     1014 non-null   float64
 5   HeartRate    1014 non-null   int64  
 6   RiskLevel    1014 non-null   object 
dtypes: float64(2), int64(4), object(1)
memory usage: 55.6+ KB
```

Figure 3: Info about dataset



```
#Summary statistics of data
print("Initial Summary Statistics:")
data.describe()
```

```
Initial Summary Statistics:
[7]:
```

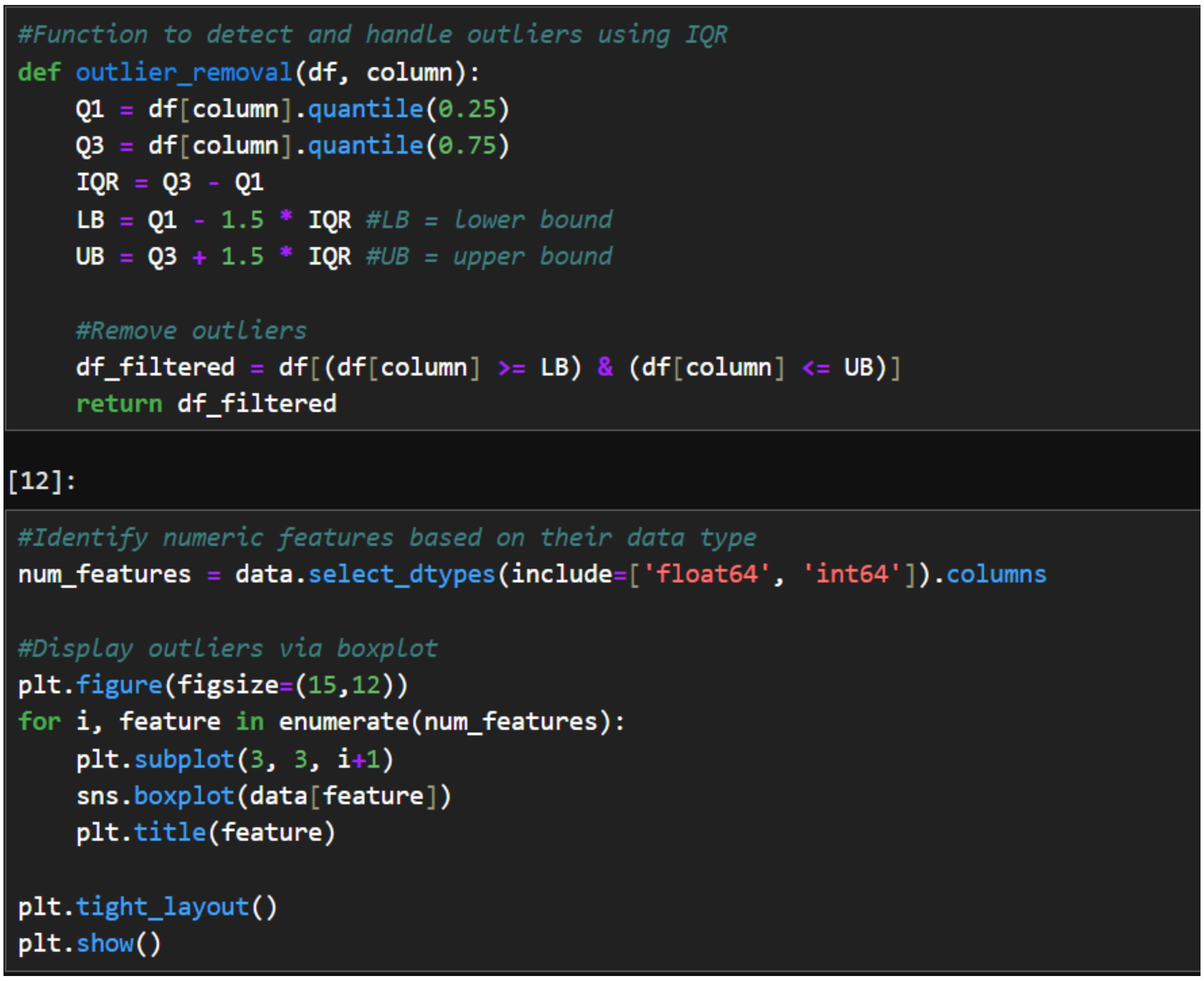
	Age	SystolicBP	DiastolicBP	BS	BodyTemp	HeartRate
count	1014.000000	1014.000000	1014.000000	1014.000000	1014.000000	1014.000000
mean	29.871795	113.198225	76.460552	8.725986	98.665089	74.301775
std	13.474386	18.403913	13.885796	3.293532	1.371384	8.088702
min	10.000000	70.000000	49.000000	6.000000	98.000000	7.000000
25%	19.000000	100.000000	65.000000	6.900000	98.000000	70.000000
50%	26.000000	120.000000	80.000000	7.500000	98.000000	76.000000
75%	39.000000	120.000000	90.000000	8.000000	98.000000	80.000000
max	70.000000	160.000000	100.000000	19.000000	103.000000	90.000000

Figure 4: Summary Statistics

4.2.2 Data Cleaning

Cleaning the data is also an essential pre-processing step. If present, missing values can be corrected in many ways e.g. imputation, column removal etc, depending on the context of the experiment and its potential implication. Tree-based models are less sensitive to outliers, however it is considered good practice to handle outliers because results can be skewed, giving inaccurate results.

In this research, a function (figure 5) was initially created to detect and handle outliers using the IQR (Inter-Quartile) method. Then the function was applied to the identified numeric features. Sometimes, outliers can affect the range of certain features, and if there is no observable distribution, the affected column can be dropped entirely, as was the case here.



```
#Function to detect and handle outliers using IQR
def outlier_removal(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    LB = Q1 - 1.5 * IQR #LB = lower bound
    UB = Q3 + 1.5 * IQR #UB = upper bound

    #Remove outliers
    df_filtered = df[(df[column] >= LB) & (df[column] <= UB)]
    return df_filtered
```

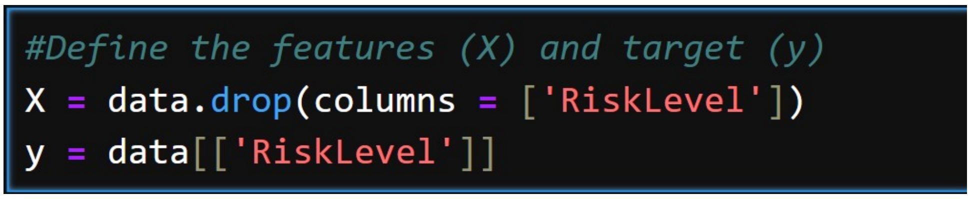
```
[12]:
```

```
#Identify numeric features based on their data type
num_features = data.select_dtypes(include=['float64', 'int64']).columns

#Display outliers via boxplot
plt.figure(figsize=(15,12))
for i, feature in enumerate(num_features):
    plt.subplot(3, 3, i+1)
    sns.boxplot(data[feature])
    plt.title(feature)

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

Figure 5: Function to remove outliers

4.2.3 Data Mining

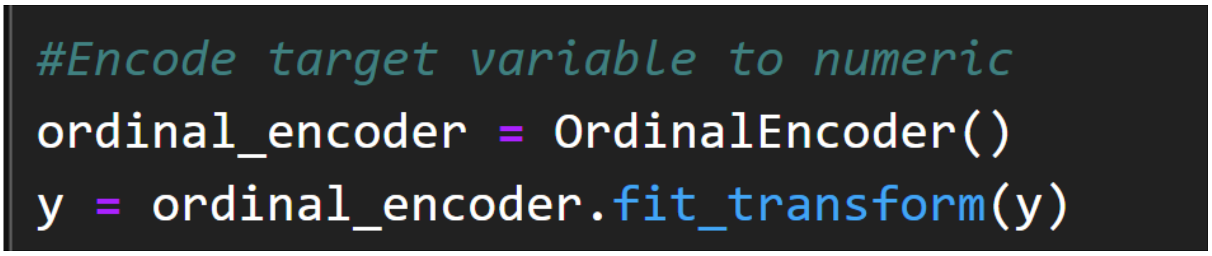
The features and target variable have to be identified and defined. Below is the script used:



```
#Define the features (X) and target (y)
X = data.drop(columns = ['RiskLevel'])
y = data[['RiskLevel']]
```

Figure 6: Defining features and target variable

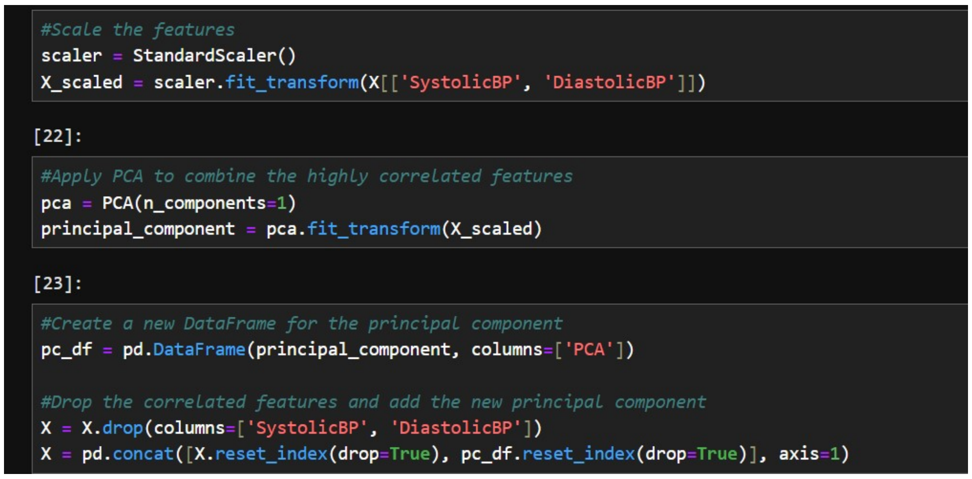
Machine Learning models usually require features to be numeric in order to performn any analysis. Encoding is a common method to convert categorical variables into numeric variables. There are different types of encoding such as Ordinal Encoding, which maintains the order, and Label Encoding, used when there is no inherent order. As the risk is categorized as high, low and mid, there is somewhat of an order there i.e. (0,1,2) so ordinal encoder was used. Since the features were already in numeric form, only the target variable was encoded using the `fit_transform` function in Python.



```
#Encode target variable to numeric
ordinal_encoder = OrdinalEncoder()
y = ordinal_encoder.fit_transform(y)
```

Figure 7: Function to encode

Feature collinearity is a common cause for biased result. A collinearity heatmap can be used to check for correlation between variables, with a reasonable threshold of 0.7 being generally considered to be too much. In this case, PCA (Principal Component Analysis) was applied to handle the multi-collinearity issue. Prior to that, however, the features were scaled, with a new dataframe created for the principal component while the correlated features were dropped.



```
#Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X[['SystolicBP', 'DiastolicBP']])

[22]:
#Apply PCA to combine the highly correlated features
pca = PCA(n_components=1)
principal_component = pca.fit_transform(X_scaled)

[23]:
#Create a new DataFrame for the principal component
pc_df = pd.DataFrame(principal_component, columns=['PCA'])

#Drop the correlated features and add the new principal component
X = X.drop(columns=['SystolicBP', 'DiastolicBP'])
X = pd.concat([X.reset_index(drop=True), pc_df.reset_index(drop=True)], axis=1)
```

Figure 8: Appplying PCA

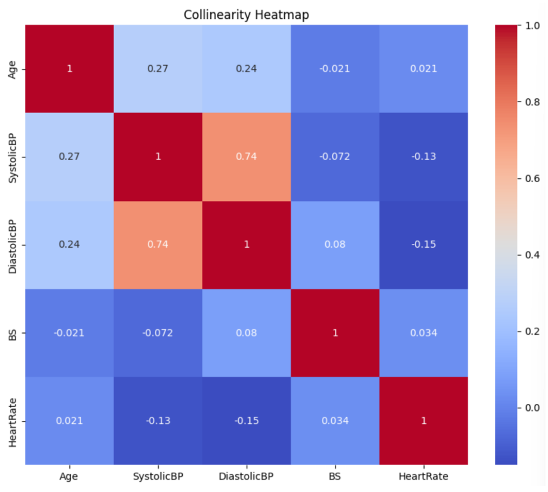


Figure 9: Heatmap before PCA

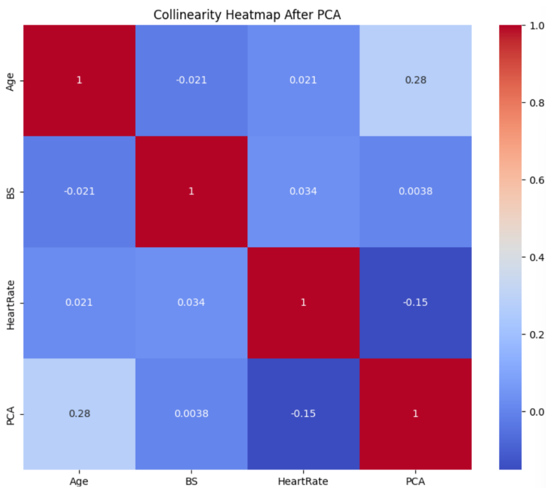
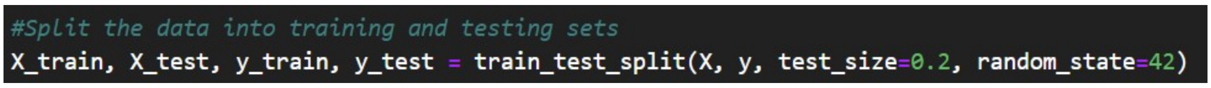


Figure 10: Heatmap after PCA

5 Model Execution

Section 5.1 and 5.2 both discuss the steps in training, testing, and evaluating the models with and without SMOTE, respectively.



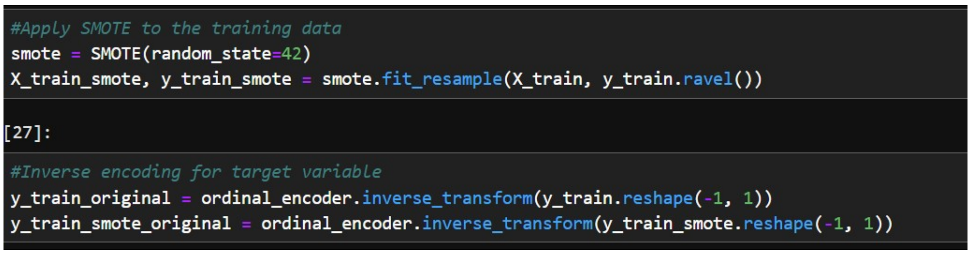
```
#Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 11: Splitting the data

Prior to any modeling, the data must be split into any ratio of training and testing subset of interest. Here, a 80/20 split was used where 80% of the data was used for analysis, while 20% was used for prediction.

5.1 Model Training with SMOTE

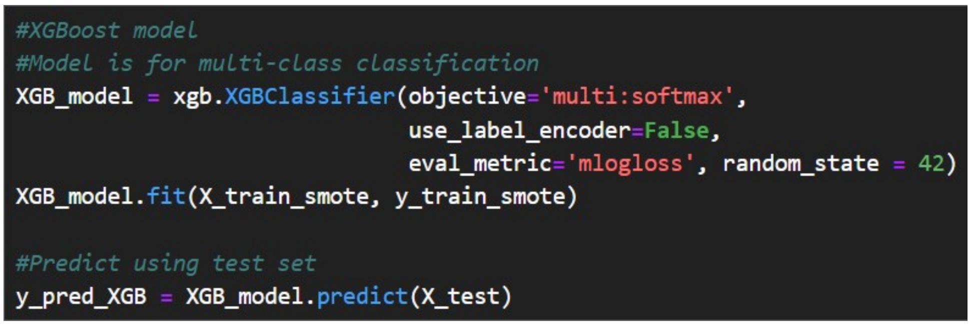
To balance the dataset, apply synthetic samples to the training data.



```
#Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train.ravel())

[27]:

#Inverse encoding for target variable
y_train_original = ordinal_encoder.inverse_transform(y_train.reshape(-1, 1))
y_train_smote_original = ordinal_encoder.inverse_transform(y_train_smote.reshape(-1, 1))
```

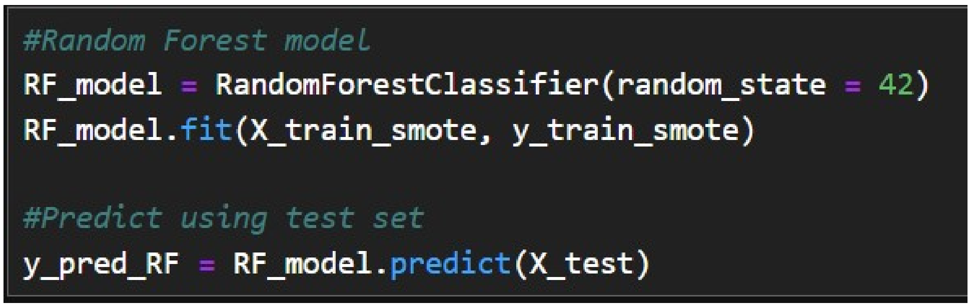
Figure 12: SMOTE

Define the models in separate cell blocks to make it more comprehensible. Then fit and predict the models using the training data and testing data, respectively.



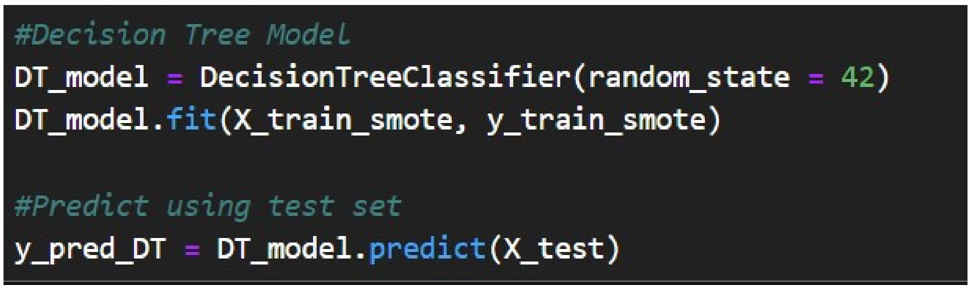
```
#XGBoost model
#Model is for multi-class classification
XGB_model = xgb.XGBClassifier(objective='multi:softmax',
                              use_label_encoder=False,
                              eval_metric='mlogloss', random_state = 42)
XGB_model.fit(X_train_smote, y_train_smote)

#Predict using test set
y_pred_XGB = XGB_model.predict(X_test)
```

Figure 13: XGBoost Classifier



```
#Random Forest model
RF_model = RandomForestClassifier(random_state = 42)
RF_model.fit(X_train_smote, y_train_smote)

#Predict using test set
y_pred_RF = RF_model.predict(X_test)
```

Figure 14: Random Forest Classifier

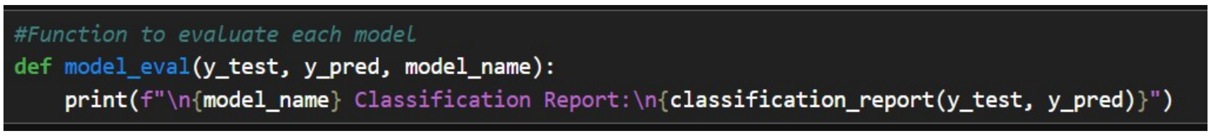


```
#Decision Tree Model
DT_model = DecisionTreeClassifier(random_state = 42)
DT_model.fit(X_train_smote, y_train_smote)

#Predict using test set
y_pred_DT = DT_model.predict(X_test)
```

Figure 15: Decision Tree Classifier

Then, reverse encode the target variable so it can be easily readable. In other words, instead of classifying the target variable as 0, 1, 2 etc, reverse encoding ensures they are classified as they originally are - high, low, and mid-risk.

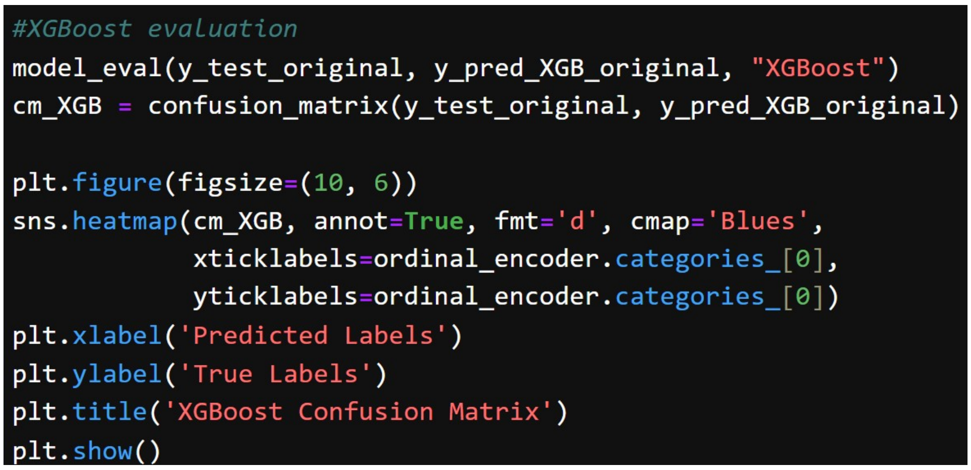
When it comes to evaluating the models, there are different ways to do so. Here, three methods are used. First is the classification report that shows the accuracy, precision, recall, and F1-score of each class and each model. The function used to do so is below:



```
#Function to evaluate each model
def model_eval(y_test, y_pred, model_name):
    print(f"\n{model_name} Classification Report:\n{classification_report(y_test, y_pred)}")
```

Figure 16: Classification Report

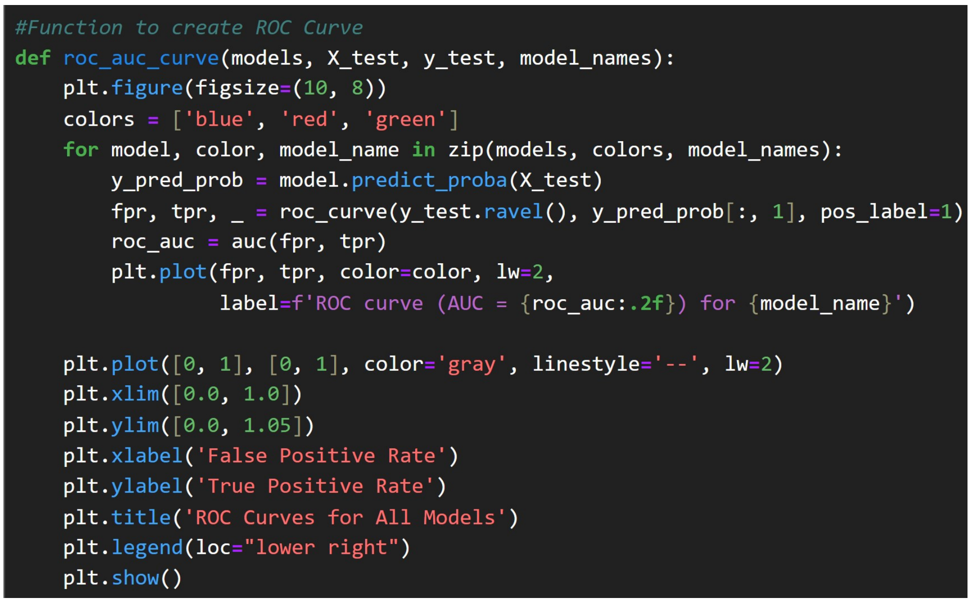
A confusion matrix is also generated for each model to highlight any true or false instances.



```
#XGBoost evaluation
model_eval(y_test_original, y_pred_XGB_original, "XGBoost")
cm_XGB = confusion_matrix(y_test_original, y_pred_XGB_original)

plt.figure(figsize=(10, 6))
sns.heatmap(cm_XGB, annot=True, fmt='d', cmap='Blues',
            xticklabels=ordinal_encoder.categories_[0],
            yticklabels=ordinal_encoder.categories_[0])
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('XGBoost Confusion Matrix')
plt.show()
```

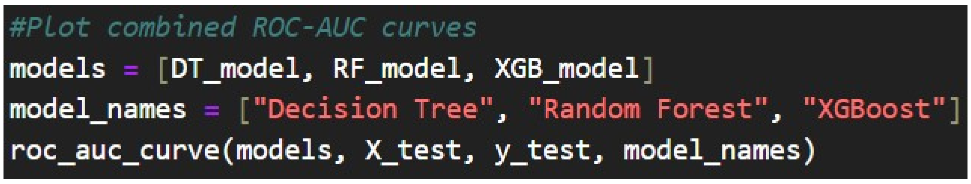
Figure 17: Confusion Matrix for XGBoost (code)

The figure above is the code snippet used to create a confusion matrix. It is the same way used to create a matrix for the other models, with just the name of the model being modified. An ROC curve is another metric to evaluate the reliability and performance of a model. Here, a multi-class classification ROC plot, which combined all three models on the scale of true positive against false positive, was generated. Figures 18 and 19 show the codes used to create the function and plot the curve, respectively.



```
#Function to create ROC Curve
def roc_auc_curve(models, X_test, y_test, model_names):
    plt.figure(figsize=(10, 8))
    colors = ['blue', 'red', 'green']
    for model, color, model_name in zip(models, colors, model_names):
        y_pred_prob = model.predict_proba(X_test)
        fpr, tpr, _ = roc_curve(y_test.ravel(), y_pred_prob[:, 1], pos_label=1)
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, color=color, lw=2,
                 label=f'ROC curve (AUC = {roc_auc:.2f}) for {model_name}')

    plt.plot([0, 1], [0, 1], color='gray', linestyle='--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curves for All Models')
    plt.legend(loc="lower right")
    plt.show()
```

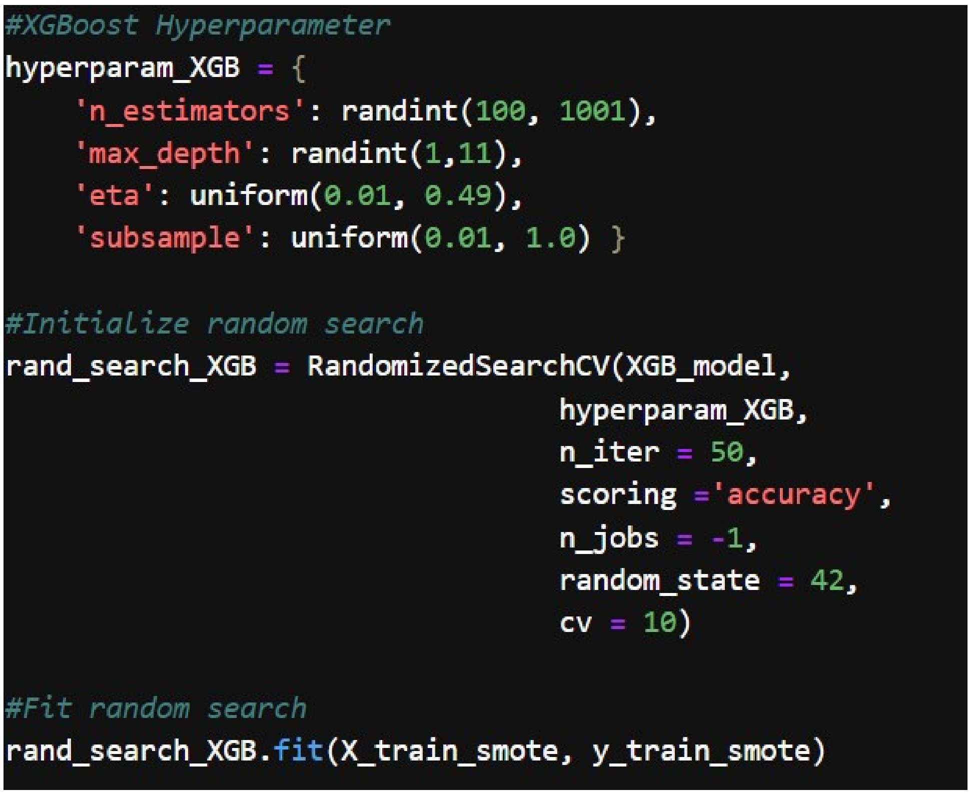
Figure 18: ROC function



```
#Plot combined ROC-AUC curves
models = [DT_model, RF_model, XGB_model]
model_names = ["Decision Tree", "Random Forest", "XGBoost"]
roc_auc_curve(models, X_test, y_test, model_names)
```

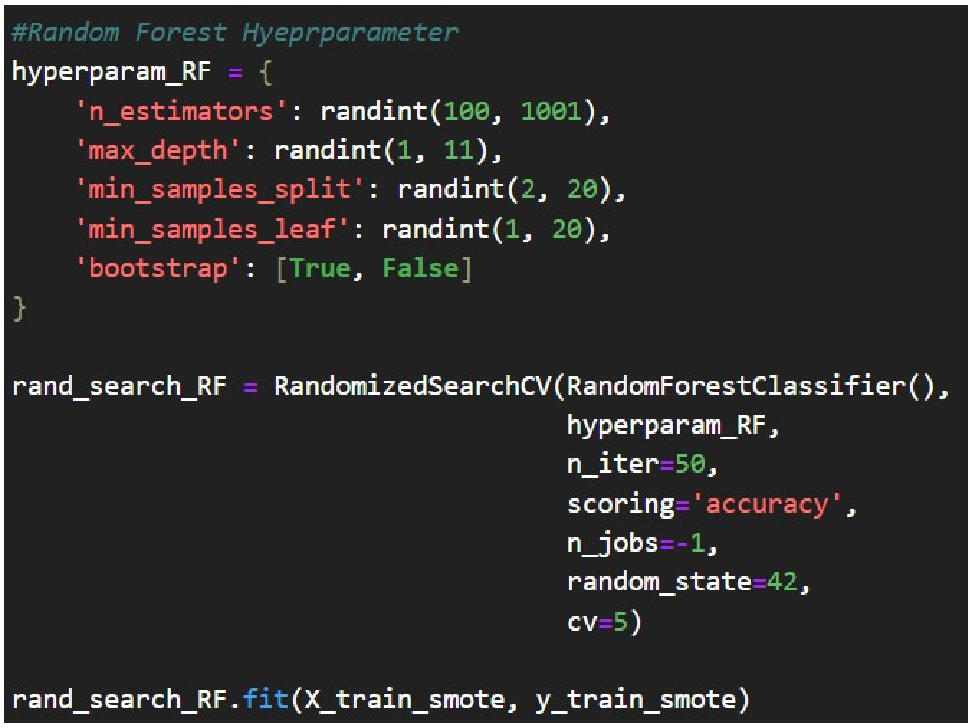
Figure 19: ROC plot

Though XGBoost proved to be the most accurate, it was evaluated on default parameters. Better results can be obtained by optimizing the model's parameters, however it does not always happen since overfitting can occur at this stage, especially with tree-based models where depth and tree numbers have to be carefully selected. To perform hyperparameter tuning, the hyperparameters have to be defined for each model.



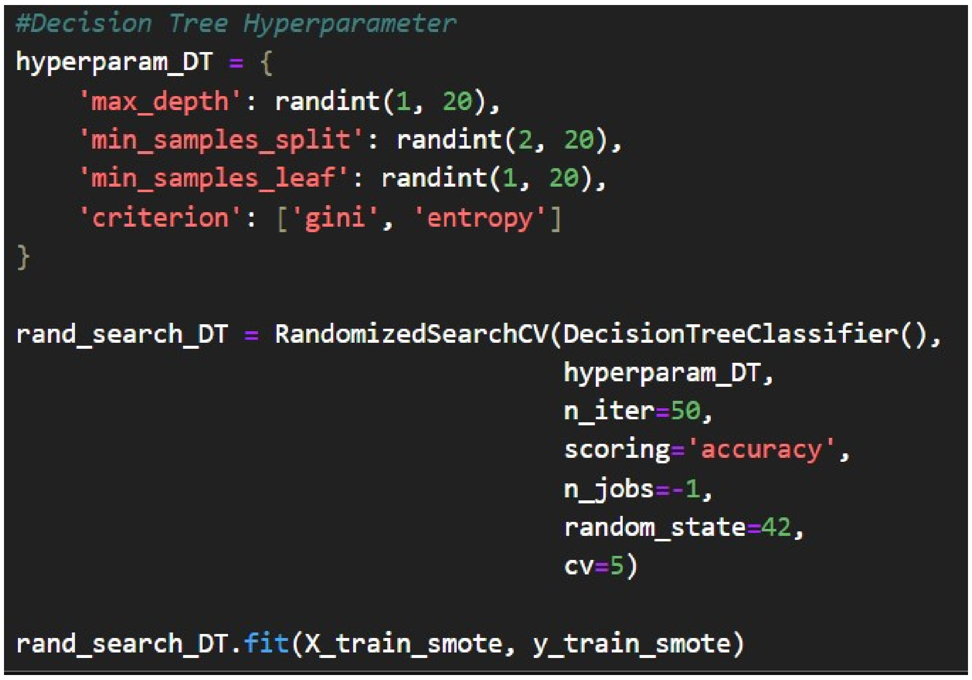
```
#XGBoost Hyperparameter
hyperparam_XGB = {
    'n_estimators': randint(100, 1001),
    'max_depth': randint(1,11),
    'eta': uniform(0.01, 0.49),
    'subsample': uniform(0.01, 1.0) }

#Initialize random search
rand_search_XGB = RandomizedSearchCV(XGB_model,
                                     hyperparam_XGB,
                                     n_iter = 50,
                                     scoring ='accuracy',
                                     n_jobs = -1,
                                     random_state = 42,
                                     cv = 10)

#Fit random search
rand_search_XGB.fit(X_train_smote, y_train_smote)
```

Figure 20: XGBoost Tuning



```python
#Random Forest Hyeprparameter
hyperparam_RF = {
    'n_estimators': randint(100, 1001),
    'max_depth': randint(1, 11),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 20),
    'bootstrap': [True, False]
}

rand_search_RF = RandomizedSearchCV(RandomForestClassifier(),
                                    hyperparam_RF,
                                    n_iter=50,
                                    scoring='accuracy',
                                    n_jobs=-1,
                                    random_state=42,
                                    cv=5)

rand_search_RF.fit(X_train_smote, y_train_smote)
```

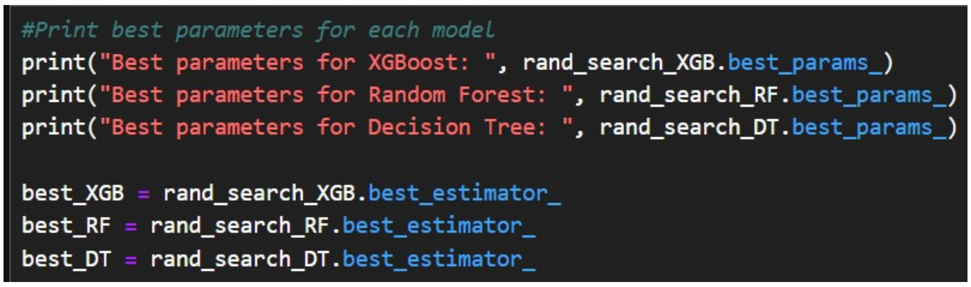
Figure 21: Random Forest Tuning



```python
#Decision Tree Hyperparameter
hyperparam_DT = {
    'max_depth': randint(1, 20),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 20),
    'criterion': ['gini', 'entropy']
}

rand_search_DT = RandomizedSearchCV(DecisionTreeClassifier(),
                                    hyperparam_DT,
                                    n_iter=50,
                                    scoring='accuracy',
                                    n_jobs=-1,
                                    random_state=42,
                                    cv=5)

rand_search_DT.fit(X_train_smote, y_train_smote)
```

Figure 22: Decision Tree Tuning

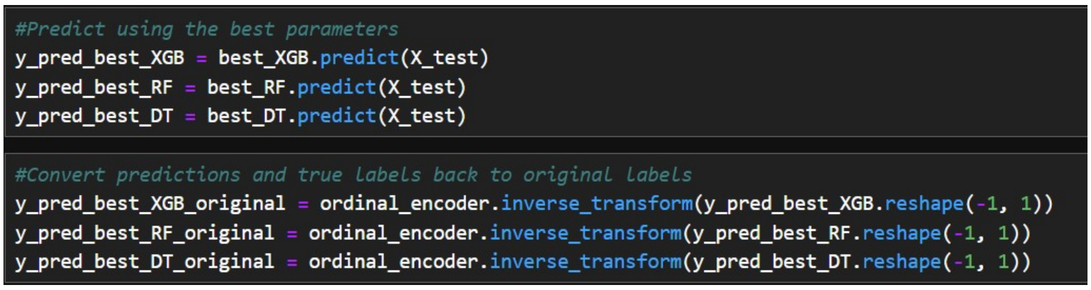
For each model, the best parameters are chosen:



```
#Print best parameters for each model
print("Best parameters for XGBoost: ", rand_search_XGB.best_params_)
print("Best parameters for Random Forest: ", rand_search_RF.best_params_)
print("Best parameters for Decision Tree: ", rand_search_DT.best_params_)

best_XGB = rand_search_XGB.best_estimator_
best_RF = rand_search_RF.best_estimator_
best_DT = rand_search_DT.best_estimator_
```

Figure 23: Best models for Hyperparameter Tuning

The predictions are done using these set of parameters, and as before, they are converted back to their original labels for easy interpretation:

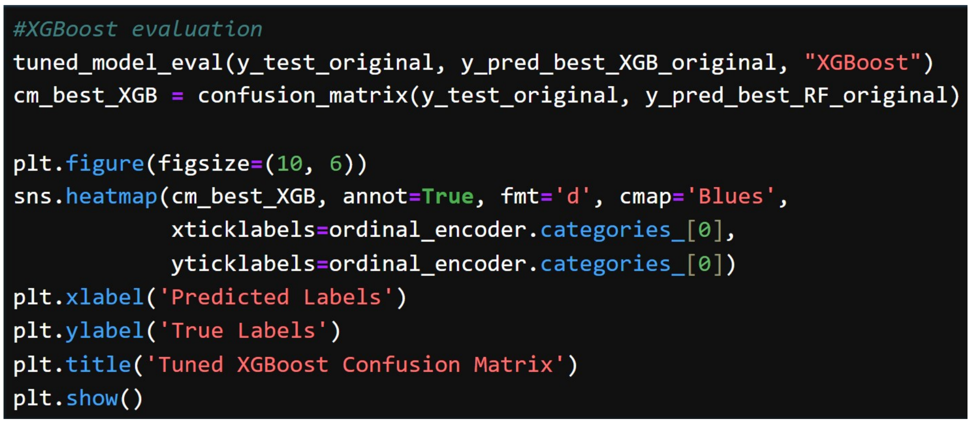


```
#Predict using the best parameters
y_pred_best_XGB = best_XGB.predict(X_test)
y_pred_best_RF = best_RF.predict(X_test)
y_pred_best_DT = best_DT.predict(X_test)
```

```
#Convert predictions and true labels back to original labels
y_pred_best_XGB_original = ordinal_encoder.inverse_transform(y_pred_best_XGB.reshape(-1, 1))
y_pred_best_RF_original = ordinal_encoder.inverse_transform(y_pred_best_RF.reshape(-1, 1))
y_pred_best_DT_original = ordinal_encoder.inverse_transform(y_pred_best_DT.reshape(-1, 1))
```

Figure 24: Encoding for Best Parameters

Using the exact steps as before hyperparameterization, with the new labels created, the classification report, confusion matrix, and ROC curves are generated. Example for the XGBoost:



```
#XGBoost evaluation
tuned_model_eval(y_test_original, y_pred_best_XGB_original, "XGBoost")
cm_best_XGB = confusion_matrix(y_test_original, y_pred_best_RF_original)

plt.figure(figsize=(10, 6))
sns.heatmap(cm_best_XGB, annot=True, fmt='d', cmap='Blues',
            xticklabels=ordinal_encoder.categories_[0],
            yticklabels=ordinal_encoder.categories_[0])
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Tuned XGBoost Confusion Matrix')
plt.show()
```

Figure 25: Tuned Classification for XGBoost

5.2 Model Training without SMOTE

The models were also evaluated without applying synthetic samples to the training data. The steps are largely the same with the exception of the 'X' and 'y' training data being left as it is instead of using the `'X_train_smote'` and `'y_train_smote'`. Hyperparameter Tuning was also done in a similar way than with the application of SMOTE. The table below shows a comparison of classification reports between both analyses -with and without SMOTE:

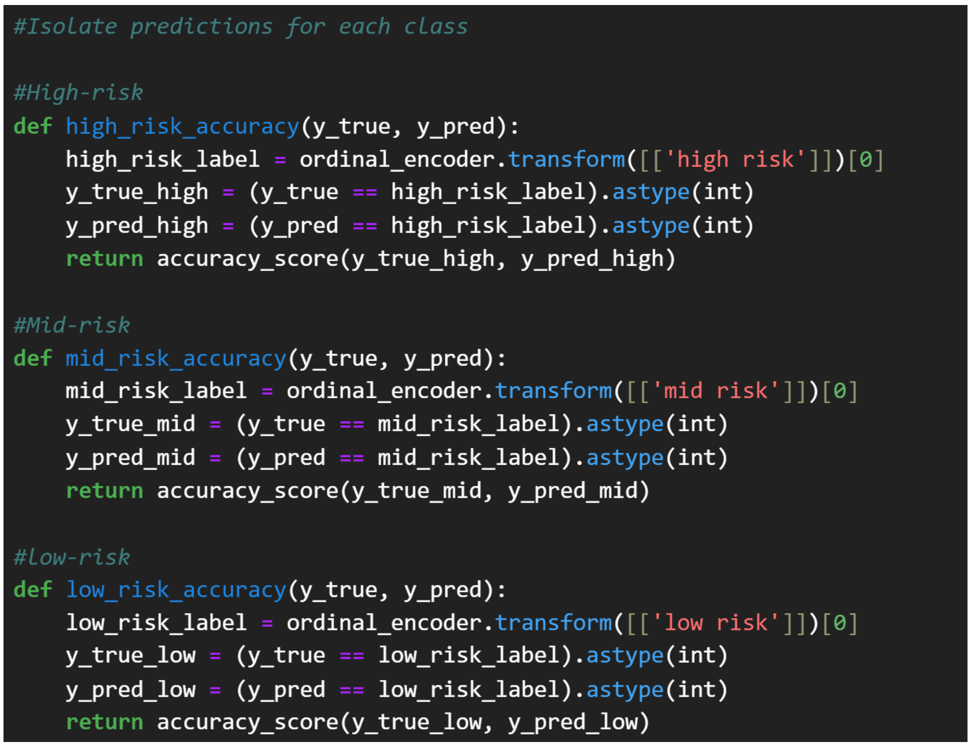
Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Pre Hyperparameter-Tuning with SMOTE				
XGBoost	0.82	0.84	0.83	0.83
Random Forest	0.80	0.81	0.81	0.81
Decision Tree	0.78	0.77	0.80	0.78
Pre Hyperparameter-Tuning without SMOTE				
XGBoost	0.80	0.82	0.80	0.81
Random Forest	0.81	0.83	0.81	0.82
Decision Tree	0.78	0.79	0.79	0.79

Table 4: Classification Report Summary: Pre Tuning

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Post Hyperparameter-Tuning with SMOTE				
XGBoost	0.81	0.82	0.82	0.82
Random Forest	0.75	0.75	0.75	0.75
Decision Tree	0.73	0.69	0.71	0.70
Post Hyperparameter-Tuning without SMOTE				
XGBoost	0.81	0.83	0.81	0.82
Random Forest	0.71	0.79	0.65	0.69
Decision Tree	0.68	0.67	0.67	0.67

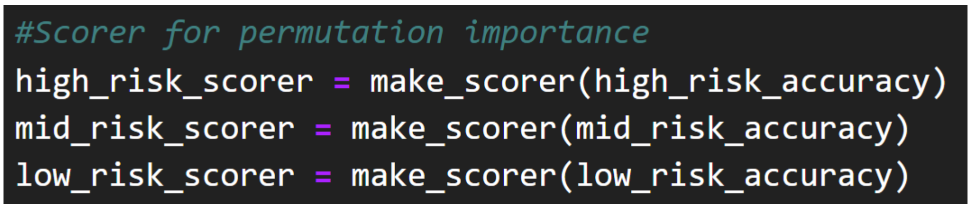
Table 5: Classification Report Summary: Post Tuning

The next step was to analyse the underlying issues related to high risk pregnancies. To do so, a feature importance plot was created based on permutation importance that examines the accuracy. The code used to generate the plot is below:



```python
#Isolate predictions for each class

#High-risk
def high_risk_accuracy(y_true, y_pred):
    high_risk_label = ordinal_encoder.transform([['high risk']])[0]
    y_true_high = (y_true == high_risk_label).astype(int)
    y_pred_high = (y_pred == high_risk_label).astype(int)
    return accuracy_score(y_true_high, y_pred_high)

#Mid-risk
def mid_risk_accuracy(y_true, y_pred):
    mid_risk_label = ordinal_encoder.transform([['mid risk']])[0]
    y_true_mid = (y_true == mid_risk_label).astype(int)
    y_pred_mid = (y_pred == mid_risk_label).astype(int)
    return accuracy_score(y_true_mid, y_pred_mid)

#Low-risk
def low_risk_accuracy(y_true, y_pred):
    low_risk_label = ordinal_encoder.transform([['low risk']])[0]
    y_true_low = (y_true == low_risk_label).astype(int)
    y_pred_low = (y_pred == low_risk_label).astype(int)
    return accuracy_score(y_true_low, y_pred_low)
```

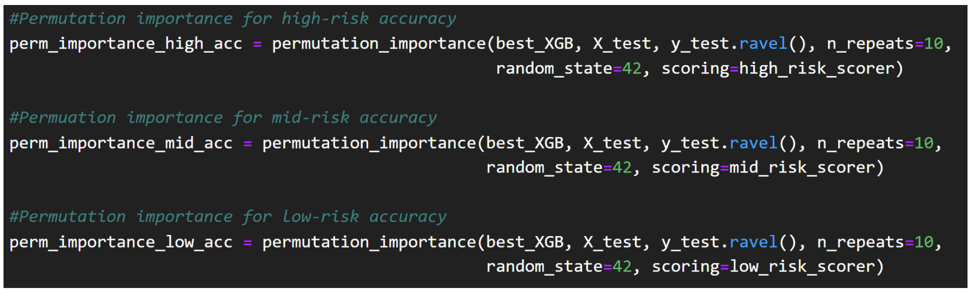
Figure 26: Feature Importance Code



```python
#Scorer for permutation importance
high_risk_scorer = make_scorer(high_risk_accuracy)
mid_risk_scorer = make_scorer(mid_risk_accuracy)
low_risk_scorer = make_scorer(low_risk_accuracy)
```

Figure 27: Scorer for permutation importance

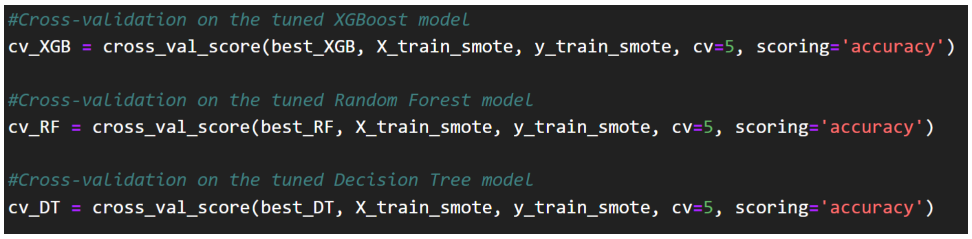
Feature Importance was generated for each class as seen below, however the focus was primarily on high-risk cases.



```python
#Permutation importance for high-risk accuracy
perm_importance_high_acc = permutation_importance(best_XGB, X_test, y_test.ravel(), n_repeats=10,
                                                  random_state=42, scoring=high_risk_scorer)

#Permuation importance for mid-risk accuracy
perm_importance_mid_acc = permutation_importance(best_XGB, X_test, y_test.ravel(), n_repeats=10,
                                                 random_state=42, scoring=mid_risk_scorer)

#Permutation importance for low-risk accuracy
perm_importance_low_acc = permutation_importance(best_XGB, X_test, y_test.ravel(), n_repeats=10,
                                                 random_state=42, scoring=low_risk_scorer)
```

Figure 28: Applying Permutation

Finally, to address any potential overfitting or underfitting, the analysis concluded with a cross-validation. Below is the code used for that:



```python
#Cross-validation on the tuned XGBoost model
cv_XGB = cross_val_score(best_XGB, X_train_smote, y_train_smote, cv=5, scoring='accuracy')

#Cross-validation on the tuned Random Forest model
cv_RF = cross_val_score(best_RF, X_train_smote, y_train_smote, cv=5, scoring='accuracy')

#Cross-validation on the tuned Decision Tree model
cv_DT = cross_val_score(best_DT, X_train_smote, y_train_smote, cv=5, scoring='accuracy')
```

Figure 29: Cross-validation