Stage 1: Study Replication

The performance metrics for each dataset and classifier developed by Doppala et al. are presented in the figure below (Doppala, Bhattacharyya, Janarthanan, & Baik, 2022).

TABLE 4: Achieved accuracies using benchmark classifiers.

Classification technique	Accuracy (%) achieved with the Cleveland dataset	Accuracy (%) achieved with the comprehensive dataset	Accuracy (%) achieved with the Mendeley dataset
Decision tree	77.86	82.56	95
Random forest	78.68	90.75	95.12
Naive Bayes	81.14	84.24	94.25
Logistic regression	81.96	84.03	95.25
Support vector machine	79.05	81.52	93.15
Gradient boosting	81.14	86.13	95.15
XGBoost	80.32	88.23	96.12

TABLE 5: Proposed model performance representation.

Classification technique	Accuracy (%) achieved with the Cleveland dataset	Accuracy (%) achieved with the comprehensive dataset	Accuracy (%) achieved with the Mendeley dataset
Proposed ensemble model	88.24	93.39	96.75

TABLE 6: Performance metrics of all the machine learning models.

Classification technique	Accuracy (%) achieved with the Cleveland dataset	Sensitivity	Specificity	Precision	Recall	F1-score	MCC
Decision tree	77.86	0.81	0.73	0.77	0.81	0.79	0.55
Random forest	78.68	0.78	0.77	0.80	0.78	0.79	0.55
Naive Bayes	81.14	0.87	0.73	0.79	0.87	0.83	0.62
Logistic regression	81.96	0.93	0.66	0.76	0.790.	0.84	0.63
Support vector machine	79.05	0.77	0.75	0.79	0.85	0.78	0.54
Gradient boosting	81.14	0.93	0.66	0.76	0.93	0.84	0.63
XGBoost	80.32	0.87	0.71	0.78	0.87	0.82	0.60
Proposed ensemble model	88.24	0.91	0.84	0.85	0.90	0.88	0.76
Classification technique	**Accuracy (%) achieved with the comprehensive dataset**	**Sensitivity**	**Specificity**	**Precision**	**Recall**	**F1-score**	**MCC**
Decision tree	82.56	0.79	0.85	0.83	0.79	0.81	0.65
Random forest	90.75	0.93	0.88	0.88	0.93	0.90	0.81
Naive Bayes	84.24	0.85	0.82	0.82	0.85	0.84	0.68
Logistic regression	84.03	0.87	0.80	0.81	0.87	0.84	0.68
Support vector machine	81.52	0.83	0.82	0.82	0.84	0.83	0.69
Gradient boosting	86.13	0.92	0.79	0.81	0.92	0.86	0.72
XGBoost	83.23	0.91	0.84	0.85	0.91	0.88	0.76
Proposed ensemble model	93.39	0.94	0.89	0.99	0.88	0.90	0.85
Classification technique	**Accuracy (%) achieved with the Mendeley dataset**	**Sensitivity**	**Specificity**	**Precision**	**Recall**	**F1-score**	**MCC**
Decision tree	95	0.95	0.94	0.96	0.95	0.95	0.88
Random forest	95.12	0.94	0.96	0.97	0.94	0.96	0.90
Naive Bayes	94.25	0.95	0.90	0.94	0.95	0.94	0.86
Logistic regression	95.25	0.97	0.95	0.97	0.97	0.97	0.92
Support vector machine	93.15	0.95	0.90	0.93	0.95	0.93	0.85
Gradient boosting	95.15	0.95	0.95	0.97	0.95	0.96	0.90
XGBoost	96.12	0.96	0.95	0.97	0.96	0.96	0.92
Proposed ensemble model	96.75	0.96	0.97	0.98	0.96	0.97	0.93

Bibliography

Doppala, B. P., Bhattacharyya, D., Janarthanan, M., & Baik, N. (2022, March 8). A Reliable Machine Intelligence Model for Accurate Identification of Cardiovascular Diseases Using Ensemble Techniques. Hindawi Journal of Healthcare Engineering, 2022. doi:https://doi.org/10.1155/2022/2585235

Dataset 1: Cleveland Dataset

Source: https://archive.ics.uci.edu/dataset/45/heart+disease

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import pandas as pd
path = "/content/drive/MyDrive/CIND 820/Heart_disease_cleveland_new.csv"
data_cleveland = pd.read_csv(path,encoding='utf-8-sig')
data_cleveland.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	0	145	233	1	2	150	0	2.3	2	0	2	0
1	67	1	3	160	286	0	2	108	1	1.5	1	3	1	1

```
#Identify duplicate rows in Cleveland Clinic Data
cleveland_dups = data_cleveland[data_cleveland.duplicated(keep=False)]

#Count number of duplicate rows in Cleveland Clinic Data
if not cleveland_dups.empty:
      cleveland_dup_num = cleveland_dups.shape[0]
else:
    print("No duplicates in Cleveland dataset")

#Check for null values in Cleveland dataset
null_cleveland =data_cleveland.isna().sum().sum()
print("There are " + str(null_cleveland) + " null values in the Cleveland dataset")
```

```
No duplicates in Cleveland dataset
There are 0 null values in the Cleveland dataset
```

```
#one-hot coding of Cleveland Data categorical independent variables

#The variables treated with one-hot encoding is unclear in replication paper, however 5 variables below are commonly encoded as in MIFH: A M
data_cleveland_coded = pd.get_dummies(data_cleveland, columns=['cp', 'restecg', 'slope','ca', 'thal'], prefix=['cp', 'restecg', 'slope', 'ca


#Output new column names as list for ease of use in test train split below
data_cleveland_coded.columns.values
```

```
array(['age', 'sex', 'trestbps', 'chol', 'fbs', 'thalach', 'exang',
       'oldpeak', 'target', 'cp_1', 'cp_2', 'cp_3', 'restecg_1',
       'restecg_2', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'ca_3',
       'thal_2', 'thal_3'], dtype=object)
```

```
#Test Train Split

#Import applicable scikit-learn libraries
from sklearn.model_selection import train_test_split

#Divide data into independent variables and dependent variable
independent = data_cleveland_coded.loc[:,['age', 'sex', 'trestbps', 'chol', 'fbs', 'thalach', 'exang','oldpeak', 'cp_1', 'cp_2', 'cp_3', 're
dependent = data_cleveland_coded.loc[:,['target']]

#Use a 60:40 test split as in CMTH642 Lab 7 and Lab 10. Assign a random_state of 0 for reproducibility of test-train split
x_train, x_test, y_train, y_test = train_test_split(independent, dependent, random_state=0, train_size = .60)


#Following advice of Jason Brownlee of https://machinelearningmastery.com/data-preparation-without-data-leakage/ and
#Data Preparation for Machine Learning Data Cleaning, Feature Selection, and Data Transforms in Python
#All data preparation must be fit on the training set only

#Data will be scaled using the minmaxscaler
from sklearn.preprocessing import MinMaxScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

#Initialize RobustScaler
scaler = MinMaxScaler()
#Fit on the training dataset to the RobustScaler instance. Fitting the data to the training set only prevents data leakage and the test set
scaler.fit(x_train[cleveland_numerical])

#Scale the training data using the RobustScaler instance
x_train[cleveland_numerical] = scaler.transform(x_train[cleveland_numerical])
#Scale the testing data using the RobustScaler instance
x_test[cleveland_numerical] = scaler.transform(x_test[cleveland_numerical])
```

```
#ca is the number of major vessels visible under fluorscopy. Upon additional literature review, it appears that this variable is commonly tr

#Use apply function and lambda to normalize the numeric columns using the normalization formula
cleveland_categorical = data_cleveland[['sex', 'cp','fbs','restecg','exang','slope','ca','thal','target']]
data_cleveland = data_cleveland.apply(lambda x: (x-min(x))/(max(x)-min(x)) if x.name not in cleveland_categorical else x)
```

```
#one-hot coding of Cleveland Data categorical independent variables

#The variables treated with one-hot encoding is unclear in replication paper, however 5 variables below are commonly encoded as in MIFH: A Ma
data_cleveland_coded = pd.get_dummies(data_cleveland, columns=['cp', 'restecg', 'slope','ca', 'thal'], prefix=['cp', 'restecg', 'slope', 'ca'
```

```
data_cleveland_coded.head()
```

```
#Output new column names as list for ease of use in test train split below
data_cleveland_coded.columns.values
```

```
#Test Train Split

#Import applicable scikit-learn libraries
from sklearn.model_selection import train_test_split

#Divide data into independent variables and dependent variable
independent = data_cleveland_coded.loc[:,['age', 'sex', 'trestbps', 'chol', 'fbs', 'thalach', 'exang','oldpeak', 'cp_1', 'cp_2', 'cp_3', 're
dependent = data_cleveland_coded.loc[:,['target']]

#Use a 60:40 test split as in CMTH642 Lab 7 and Lab 10. Assign a random_state of 0 for reproducibility of test-train split
x_train, x_test, y_train, y_test = train_test_split(independent, dependent, random_state=0, train_size = .60)
```

```
#Decision Tree

from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

#Initialize the decision tree classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#Fit the training data set to the decision tree model
cleveland_decision_tree.fit(x_train, y_train.values.ravel()).predict(x_test)

#Predict the presence of heart disese by inputting the test data into the cleveland_decision_tree model
target_pred_decision_tree = cleveland_decision_tree.predict(x_test)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1),3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1),3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1),3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree),3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0),3)

#Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree,precision_decision_tree,recall_decision_tre

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_decision_tree = performance_metrics

performance_metrics
```

```
#Random Forest

from sklearn.ensemble import RandomForestClassifier

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 0 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#Fit the training data set to the random forest classifier
cleveland_random_forest.fit(x_train, y_train.values.ravel()).predict(x_test)

#Predict the presence of heart disese by inputting the test data into the cleveland_random_forest model
target_pred_random_forest = cleveland_random_forest.predict(x_test)

accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

#Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest,specificity_random_forest,precision_random_forest,recall_random_forest

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_random_forest = performance_metrics

performance_metrics
```

```
#Naive Bayes

#The application of Naive Bayes in the paper is unclear. The dataset contains both categorical and numerical (i.e. continuous numerical) fea
#Here we will applying different Naive Bayes classifiers to the categorical and numerical features
from sklearn.naive_bayes import  CategoricalNB, GaussianNB

#Initialize the naive bayes models and assign to variable
cleveland_naive_numerical = GaussianNB()
cleveland_naive_categorical = CategoricalNB()

#List categorical and numerical feature names
cleveland_categorical_nb = ['sex', 'fbs', 'exang','cp_1', 'cp_2', 'cp_3', 'restecg_1','restecg_2', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'ca
cleveland_numerical_nb = ['age','trestbps','chol','thalach','oldpeak']

# Fit the categorical features in the training data set to the categorical naive bayes model
cleveland_naive_categorical.fit(x_train[cleveland_categorical_nb], y_train.values.ravel())

#Fit the numerical features in the training data set to the gaussian naive bayes model
cleveland_naive_numerical.fit(x_train[cleveland_numerical_nb], y_train.values.ravel())

# Predict probabilities for using categorical and numerical features
probability_categorical = cleveland_naive_categorical.predict_proba(x_test[cleveland_categorical_nb])
probability_numerical= cleveland_naive_numerical.predict_proba(x_test[cleveland_numerical_nb])

#Combine the probabilities using the product rule
total_probability = probability_categorical * probability_numerical

#We can use this code to select the class that has the greatest probability for a given row
import numpy as np
target_pred_naive = np.argmax(total_probability, axis=1)

#Calculate performance metrics
accuracy_naive =  round(accuracy_score(y_test, target_pred_naive),4)*100
precision_naive = round(precision_score(y_test, target_pred_naive, pos_label=1),3)
recall_naive = round(recall_score(y_test, target_pred_naive, pos_label=1),3)
f1_score_naive = round(f1_score(y_test, target_pred_naive, pos_label=1),3)
mcc_naive = round(matthews_corrcoef(y_test, target_pred_naive),3)
specificity_naive = round(recall_score(y_test, target_pred_naive, pos_label=0),3)

#Organize performance metrics into a list
performance_naive = [["Naive Bayes", accuracy_naive,specificity_naive,precision_naive,recall_naive,f1_score_naive,mcc_naive]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_naive, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', 'MCC'])
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_naive_bayes = performance_metrics

performance_metrics
```

```
#Logistic Regression

#Import logistic regression model from scikit-learn libraries
from sklearn.linear_model import LogisticRegression

#Initialize the logistic regression model and assign to variable wine_logistic_reg variable. Assign random_state 16 to reproduce results
cleveland_logistic_reg = LogisticRegression(random_state=42)

#Fit the training data set to the logistic model
cleveland_logistic_reg.fit(x_train, y_train.values.ravel())

#Predict the presence of heart disese by inputting the test data into the cleveland_logistic_reg
target_pred_logistic = cleveland_logistic_reg.predict(x_test)

accuracy_logistic =  round(accuracy_score(y_test, target_pred_logistic),4)*100
precision_logistic = round(precision_score(y_test, target_pred_logistic, pos_label=1),3)
recall_logistic = round(recall_score(y_test, target_pred_logistic, pos_label=1),3)
f1_score_logistic = round(f1_score(y_test, target_pred_logistic, pos_label=1),3)
mcc_logistic = round(matthews_corrcoef(y_test, target_pred_logistic),3)
specificity_logistic = round(recall_score(y_test, target_pred_logistic, pos_label=0),3)

#Organize performance metrics into a list
performance_logistic = [["Logistic Regression", accuracy_logistic,specificity_logistic,precision_logistic,recall_logistic,f1_score_logistic,

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_logistic, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_logistic_regression = performance_metrics

performance_metrics


#Import support vector machine from scikit-learn libraries
from sklearn import svm

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_support_vector = svm.SVC(kernel='linear', random_state=42)

#Fit the training data set to the support vector machine classifier
cleveland_support_vector.fit(x_train, y_train.values.ravel()).predict(x_test)

#Predict the presence of heart disese by inputting the test data into the cleveland_support_vector
target_pred_support_vector = cleveland_support_vector.predict(x_test)

accuracy_support_vector =  round(accuracy_score(y_test, target_pred_support_vector),4)*100
precision_support_vector = round(precision_score(y_test, target_pred_support_vector, pos_label=1),3)
recall_support_vector = round(recall_score(y_test, target_pred_support_vector, pos_label=1),3)
f1_score_support_vector = round(f1_score(y_test, target_pred_support_vector, pos_label=1),3)
mcc_support_vector = round(matthews_corrcoef(y_test, target_pred_support_vector),3)
specificity_support_vector = round(recall_score(y_test, target_pred_support_vector, pos_label=0),3)

#Organize performance metrics into a list
performance_support_vector = [["Support Vector", accuracy_support_vector,specificity_support_vector,precision_support_vector,recall_support_

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_support_vector, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_support_vector = performance_metrics

performance_metrics
```

```
#Gradient Boosting

#Import gradient boosting classifier from scikit-learn libraries
from sklearn.ensemble import GradientBoostingClassifier

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 42 to reproduce result
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train, y_train.values.ravel()).predict(x_test)

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics


#XGBoost

#Import xgboost
import xgboost as xgb

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

#Fit the training data set to the support vector machine classifier
cleveland_xgb.fit(x_train, y_train.values.ravel()).predict(x_test)

#Predict the presence of heart disese by inputting the test data into the cleveland_xgb
target_pred_xgb = cleveland_xgb.predict(x_test)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb,specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb= performance_metrics

performance_metrics
```

```
#Ensemble Model

#Import VotingClassifier to combine model predictions
from sklearn.ensemble import VotingClassifier

#Initialize a categorical naive bayes classifier and train using categorical training set
ensemble_categorical_nb = CategoricalNB()
ensemble_categorical_nb.fit(x_train[cleveland_categorical_nb], y_train.values.ravel())

#Initialize a gaussian naive bayes classifier and train using numerical (i.e continuous numerical features) training set
ensemble_numerical_nb = GaussianNB()
ensemble_numerical_nb.fit(x_train[cleveland_numerical_nb], y_train.values.ravel())

# Initialize remaining classifier types and fit the entire training setata
ensemble_random_forest = RandomForestClassifier(random_state = 42)
ensemble_random_forest.fit(x_train, y_train.values.ravel())

ensemble_svm = svm.SVC(probability=True, random_state = 42)
ensemble_svm.fit(x_train, y_train.values.ravel())

ensemble_gradient = GradientBoostingClassifier(random_state = 42)
ensemble_gradient.fit(x_train, y_train.values.ravel())

#Specify Algorithms and initialize ensemble model using a soft voting classifier
algorithms = [('ensemble_categorical_nb',ensemble_categorical_nb),('ensemble_numerical_nb',ensemble_numerical_nb),('ensemble_random_forest',
ensemble = VotingClassifier(estimators=algorithms, voting='soft')

#Now we can fit each model to voting classifier instance, selecting the categorical variables for naive bayes categorical models
ensemble.fit(
    np.column_stack([ensemble_categorical_nb.predict_proba(x_train[cleveland_categorical_nb]),
                     ensemble_numerical_nb.predict_proba(x_train[cleveland_numerical_nb]),
                     ensemble_random_forest.predict_proba(x_train),
                     ensemble_gradient.predict_proba(x_train),
                     ensemble_svm.predict_proba(x_train)]),
    y_train.values.ravel()
)

#If acting similarly, may be weakening the classifiers
#Check performance with parametric and non-parametric statistics based on results. Normally distributed results - use ANOVA
#Wilcoxon - for medians
#With the ensemble model, we can make predictions on the target values
target_pred_ensemble = ensemble.predict(
    np.column_stack([ensemble_categorical_nb.predict_proba(x_test[cleveland_categorical_nb]),
                     ensemble_numerical_nb.predict_proba(x_test[cleveland_numerical_nb]),
                     ensemble_random_forest.predict_proba(x_test),
                     ensemble_gradient.predict_proba(x_test),
                     ensemble_svm.predict_proba(x_test)])
)

accuracy_ensemble =  round(accuracy_score(y_test, target_pred_ensemble),4)*100
precision_ensemble = round(precision_score(y_test, target_pred_ensemble, pos_label=1),3)
recall_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=1),3)
f1_score_ensemble = round(f1_score(y_test, target_pred_ensemble, pos_label=1),3)
mcc_ensemble = round(matthews_corrcoef(y_test, target_pred_ensemble),3)
specificity_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=0),3)

#Organize performance metrics into a list
performance_ensemble = [["Ensemble", accuracy_ensemble,specificity_ensemble,precision_ensemble,recall_ensemble,f1_score_ensemble,mcc_ensembl

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_ensemble, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_ensemble = performance_metrics

performance_metrics
```

```
#Summary table - Stage 1 Basic Results

# Use concat to append DataFrames vertically
summary = pd.concat([st1_pm_decision_tree, st1_pm_random_forest, st1_pm_naive_bayes, st1_pm_logistic_regression, st1_pm_support_vector, st1_

#Remove row index values
summary.index = ["","","","","","","",""]
summary

#Although the same procedure was followed as the research paper, the accuracy of most models does not approximate that of the study except f
#The Ensemble Model, XGBoost and Naive Bayes show the greatest deviaton from the study paper. Notably, the classifiers that do not match the
```

Stage 1A: Changes to Approximate Study Accuracy

Hyperparameter tuning will be attempted to approximate the accuracy measure of the study paper for tree-based methods.

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the decision tree classifier
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers
param_dist = {'criterion': ['gini', 'entropy'],
              'max_depth': [None, 1,2,10, 20, 30],
              'min_samples_split': [2, 5, 10,15,18],
              'min_samples_leaf': [1, 2, 4]}

#Initialize the GridSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_dt = GridSearchCV(cleveland_decision_tree , param_dist, cv=10)
grid_cleveland_dt.fit(x_train, y_train)

#Output the best parameters, the model is optimized based on accuracy score
best_params_dt = grid_cleveland_dt.best_params_
print(best_params_dt)

#Fit the model using athe best parameters
cleveland_decision_tree = DecisionTreeClassifier(**best_params_dt, random_state=42)
cleveland_decision_tree.fit(x_train, y_train)

# Use the best model for predictions and recalculate metrics
target_pred_decision_tree = cleveland_decision_tree.predict(x_test)

#Calculate Performance Metrics
accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1), 3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1), 3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1), 3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree), 3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0), 3)

# Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree, precision_decision_tree, recall_decision_t

# Create a DataFrame of performance metrics
grid_dt_pm = pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_dt_pm.index = [""]

grid_dt_pm

#There are improvements to the metrics with the exception of specificity
```

```
{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 15}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
Decision Tree	76.23	0.833	0.811	0.694	0.748	0.531

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 42 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 75, 100, 125, 125, 175, 200, 300],
              'max_features': ['sqrt', 'log2'],
              'max_depth': [None, 1,2,10,20,30],
              'min_samples_split': [2,3,4,5,10,15,18,20],
              'min_samples_leaf': [1,2,4,5,6,7,8,9]}

#Initialize the GridSearchCV class using the random forest, the parameter grid and a 10-fold cross-validation
grid_cleveland_rf = GridSearchCV(cleveland_random_forest , param_dist, cv=10)
grid_cleveland_rf.fit(x_train, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_rf = grid_cleveland_rf.best_params_
print(best_params_rf)

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(**best_params_dt, random_state=42)
cleveland_random_forest.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for

# Create a DataFrame of performance metrics
grid_rf_pm = pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_rf_pm.index = [""]

grid_rf_pm

#Despite performing GridSearchCV, the accuracy remains unchanged. RandomSearchCV is attempted next
```

```
#Import RandomizedSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import RandomizedSearchCV

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 42 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 100, 200, 300],
              'max_features': ['auto', 'sqrt', 'log2'],
              'max_depth': [None, 1,2,10,20,30],
              'min_samples_split': [2,3,4,5,10,15,18,20],
              'min_samples_leaf': [1,2,4,5,6]}

#Initialize the RandomizedSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_rf = RandomizedSearchCV(cleveland_random_forest , param_dist, cv=10)
grid_cleveland_rf.fit(x_train, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_rf = grid_cleveland_rf.best_params_
print(best_params_rf)

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(**best_params_dt, random_state=42)
cleveland_random_forest.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_decision_tree = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for

# Create a DataFrame of performance metrics
grid_rf_pm = pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_rf_pm.index = [""]

grid_rf_pm

#Results are the same despite performing RandomSearchCV
```

```
#Import RandomizedSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import RandomizedSearchCV

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 42 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 100, 200, 300],
              'max_features': ['auto', 'sqrt', 'log2'],
              'max_depth': [None, 1,2,10,20,30],
              'min_samples_split': [2,3,4,5,10,15,18,20],
              'min_samples_leaf': [1,2,4,5,6]}

#Initialize the RandomizedSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_rf = RandomizedSearchCV(cleveland_random_forest , param_dist, cv=10)
grid_cleveland_rf.fit(x_train, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_rf = grid_cleveland_rf.best_params_
print(best_params_rf)

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(**best_params_dt, random_state=42)
cleveland_random_forest.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_decision_tree = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for

# Create a DataFrame of performance metrics
grid_rf_pm = pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_rf_pm.index = [""]

grid_rf_pm
```

```
#Create list of n_estimators to cycle through. The hyperparameter n_estimators is the number of decision trees in the random forest
n_estimators = [x for x in range(1, 100)]

#Create empty lists to store the calculated accuracy and random state
accuracy_list = []
random_states = []

#Create loop to calculate the accuracy for combinations of n_estimators and random_state
#Due to the small dataset, the accuracy fluctuated with differing random_states
for i in n_estimators:
    #Initilized a random forest classifier with default values for hyperparameters except for n_estimators and random_state
    cleveland_rf_opt = RandomForestClassifier(n_estimators=i, random_state =i)
    #Fit the initialized model to the training data, convert y_train values to a 1-D array. Predict the heart disease classes using x_test a
    cleveland_rf_opt.fit(x_train, y_train.values.ravel())
    target_pred_rf_opt = cleveland_rf_opt.predict(x_test)
    #Calculate the accuracy using y_test as the true values and the predicted targets
    model_accuracy = accuracy_score(y_test.values.ravel(), target_pred_rf_opt)
    accuracy_list.append(model_accuracy)
    random_states.append(i)

# Locate the index maximum accuracy, match to the list of n_estimators and random_states
max_estimators = n_estimators[accuracy_list.index(max(accuracy_list))]
max_random_state = random_states[accuracy_list.index(max(accuracy_list))]

#Output the maximum accuracy and the settings for the hyperparameters
print("The max accuracy is:", round(max(accuracy_list)*100,2))
print("The optimal n_estimators is: "+str(max_random_state)+". The optimal random_state is: "+str(max_random_state))

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(n_estimators=65, max_depth=None, min_samples_split=2, max_leaf_nodes=None, min_samples_leaf
cleveland_random_forest.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test,  target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test,  target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test,  target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test,  target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test,  target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for
#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]
performance_metrics

#This method of modifying only the number of trees and the random state provides the greatest accuracy and is closest to the research paper.
#The remaining performance metrics better approximate the results of the study paper with the exception of recall and F1 score.
```

```
#Gradient Boosting

#Initialize the gradient boosting classifier and assign to variable cleveland_gradient. Assign random_state 42 to reproduce results
cleveland_gradient = GradientBoostingClassifier(random_state=42)

#A parameter grid was created using selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 100, 200],
              'learning_rate': [0.01, 0.1, 0.2],
              'max_depth': [3, 4, 5],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1, 2, 4]}

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train, y_train.values.ravel()).predict(x_test)

#Initialize the GridSearchCV class using the gradient boosting, the parameter grid and a 10-fold cross-validation
grid_cleveland_gb = GridSearchCV(cleveland_gradient, param_dist, cv=10)
grid_cleveland_gb.fit(x_train, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_gb = grid_cleveland_gb.best_params_
print(best_params_gb)

#Fit the model using athe best parameters
cleveland_gradient= GradientBoostingClassifier(**best_params_gb, random_state=42)
cleveland_gradient.fit(x_train, y_train.values.ravel())

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

```
{'learning_rate': 0.1, 'max_depth': 4, 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 200}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
Gradient Boosting	75.41	0.783	0.776	0.726	0.75	0.51

```
#XGBoost

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(enable_categorical=True, seed= 42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0]
}

#Initialize the RandomizedSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_xgb = GridSearchCV(cleveland_xgb , param_dist, cv=10)
grid_cleveland_xgb.fit(x_train, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_xgb = grid_cleveland_xgb.best_params_
print(best_params_xgb)

#Fit the model using athe best parameters
cleveland_xgb = xgb.XGBClassifier(**best_params_xgb, seed=42)
cleveland_xgb.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_xgb.predict(x_test)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb,specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb= performance_metrics

performance_metrics
```

```
{'colsample_bytree': 0.9, 'learning_rate': 0.2, 'max_depth': 3, 'n_estimators': 50, 'subsample': 1.0}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
XGBoost	76.23	0.85	0.824	0.677	0.743	0.535

```
# Import necessary libraries
from sklearn.svm import SVC
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef, recall_score
import pandas as pd

# Initialize the svm classifier
cleveland_svm = SVC(probability=True, random_state=42)

# Define the parameter grid for hyperparameter tuning
param_dist = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': [2, 3, 4, 5],
    'gamma': ['scale', 'auto'] + [0.01, 0.1, 1, 10],
}

# Initialize the RandomizedSearchCV class for SVM
grid_cleveland_svm = RandomizedSearchCV(cleveland_svm, param_dist, cv=10)
grid_cleveland_svm.fit(x_train, y_train.values.ravel())

# Output the best parameters, the model is optimized based on accuracy score
best_params_svm = grid_cleveland_svm.best_params_
print(best_params_svm)

# Fit the SVM model using the best parameters
cleveland_svm = SVC(**best_params_svm, probability=True, random_state=42)
cleveland_svm.fit(x_train, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_svm = cleveland_svm.predict(x_test)

# Calculate Performance Metrics
accuracy_svm = round(accuracy_score(y_test, target_pred_svm), 4) * 100
precision_svm = round(precision_score(y_test, target_pred_svm, pos_label=1), 3)
recall_svm = round(recall_score(y_test, target_pred_svm, pos_label=1), 3)
f1_score_svm = round(f1_score(y_test, target_pred_svm, pos_label=1), 3)
mcc_svm = round(matthews_corrcoef(y_test, target_pred_svm), 3)
specificity_svm = round(recall_score(y_test, target_pred_svm, pos_label=0), 3)

# Organize performance metrics into a list
performance_svm = [["SVM", accuracy_svm, specificity_svm, precision_svm, recall_svm, f1_score_svm, mcc_svm]]

# Create a DataFrame of performance metrics
grid_svm_pm = pd.DataFrame(performance_svm, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_svm_pm.index = [""]
```

```
{'kernel': 'sigmoid', 'gamma': 0.1, 'degree': 2, 'C': 10}
```

```
#Ensemble Model

#Import VotingClassifier to combine model predictions
from sklearn.ensemble import VotingClassifier

#Initialize a categorical naive bayes classifier and train using categorical training set
ensemble_categorical_nb = CategoricalNB()
ensemble_categorical_nb.fit(x_train[cleveland_categorical_nb], y_train.values.ravel())

#Initialize a gaussian naive bayes classifier and train using numerical (i.e continuous numerical features) training set
ensemble_numerical_nb = GaussianNB()
ensemble_numerical_nb.fit(x_train[cleveland_numerical_nb], y_train.values.ravel())

# Initialize remaining classifier types and fit the entire training setata
ensemble_random_forest = RandomForestClassifier(random_state = 42, min_samples_split= 15, min_samples_leaf= 4, max_depth= 10, criterion = 'e
ensemble_random_forest.fit(x_train, y_train.values.ravel())

ensemble_svm = svm.SVC(probability=True, random_state = 42, kernel= 'sigmoid',gamma= 0.1, degree= 2, C= 10)
ensemble_svm.fit(x_train, y_train.values.ravel())

ensemble_gradient = GradientBoostingClassifier(random_state = 42, learning_rate=0.1, max_depth= 4, min_samples_leaf=4, min_samples_split= 2,
ensemble_gradient.fit(x_train, y_train.values.ravel())

#Specify Algorithms and initialize ensemble model using a soft voting classifier
algorithms = [('ensemble_categorical_nb',ensemble_categorical_nb),('ensemble_numerical_nb',ensemble_numerical_nb),('ensemble_random_forest',
ensemble = VotingClassifier(estimators=algorithms, voting='soft')

#Now we can fit each model to voting classifier instance, selecting the categorical variables for naive bayes categorical models
ensemble.fit(
    np.column_stack([ensemble_categorical_nb.predict_proba(x_train[cleveland_categorical_nb]),
                     ensemble_numerical_nb.predict_proba(x_train[cleveland_numerical_nb]),
                     ensemble_random_forest.predict_proba(x_train),
                     ensemble_gradient.predict_proba(x_train),
                     ensemble_svm.predict_proba(x_train)]),
    y_train.values.ravel()
)

#If acting similarly, may be weakening the classifiers
#Check performance with parametric and non-parametric statistics based on results. Normally distributed results - use ANOVA
#Wilcoxon - for medians
#With the ensemble model, we can make predictions on the target values
target_pred_ensemble = ensemble.predict(
    np.column_stack([ensemble_categorical_nb.predict_proba(x_test[cleveland_categorical_nb]),
                     ensemble_numerical_nb.predict_proba(x_test[cleveland_numerical_nb]),
                     ensemble_random_forest.predict_proba(x_test),
                     ensemble_gradient.predict_proba(x_test),
                     ensemble_svm.predict_proba(x_test)])
)

accuracy_ensemble =  round(accuracy_score(y_test, target_pred_ensemble),4)*100
precision_ensemble = round(precision_score(y_test, target_pred_ensemble, pos_label=1),3)
recall_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=1),3)
f1_score_ensemble = round(f1_score(y_test, target_pred_ensemble, pos_label=1),3)
mcc_ensemble = round(matthews_corrcoef(y_test, target_pred_ensemble),3)
specificity_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=0),3)

#Organize performance metrics into a list
performance_ensemble = [["Ensemble", accuracy_ensemble,specificity_ensemble,precision_ensemble,recall_ensemble,f1_score_ensemble,mcc_ensembl

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_ensemble, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_ensemble = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Ensemble	75.41	0.783	0.776	0.726	0.75	0.51

Stage 2: Improvements to Classifiers

```
import pandas as pd
path = "/content/drive/MyDrive/CIND 820/Heart_disease_cleveland_new.csv"
data_cleveland = pd.read_csv(path,encoding='utf-8-sig')
data_cleveland.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	0	145	233	1	2	150	0	2.3	2	0	2	0
1	67	1	3	160	286	0	2	108	1	1.5	1	3	1	1
2	67	1	3	120	229	0	2	129	1	2.6	1	2	3	1
3	37	1	2	130	250	0	0	187	0	3.5	2	0	1	0
4	41	0	1	130	204	0	2	172	0	1.4	0	0	1	0

```
#one-hot coding of Cleveland Data categorical independent variables

#The variables treated with one-hot encoding is unclear in replication paper, however 5 variables below are commonly encoded as in MIFH: A M
data_cleveland_coded = pd.get_dummies(data_cleveland, columns=['cp', 'restecg', 'slope','ca', 'thal'], prefix=['cp', 'restecg', 'slope', 'ca


#Output new column names as list for ease of use in test train split below
data_cleveland_coded.columns.values
```

```
array(['age', 'sex', 'trestbps', 'chol', 'fbs', 'thalach', 'exang',
       'oldpeak', 'target', 'cp_1', 'cp_2', 'cp_3', 'restecg_1',
       'restecg_2', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'ca_3',
       'thal_2', 'thal_3'], dtype=object)
```

```
#Test Train Split

#Import applicable scikit-learn libraries
from sklearn.model_selection import train_test_split

#Divide data into independent variables and dependent variable
independent = data_cleveland_coded.loc[:,['age', 'sex', 'trestbps', 'chol', 'fbs', 'thalach', 'exang','oldpeak', 'cp_1', 'cp_2', 'cp_3', 'res
dependent = data_cleveland_coded.loc[:,['target']]

#Use a 60:40 test split as in CMTH642 Lab 7 and Lab 10. Assign a random_state of 0 for reproducibility of test-train split
x_train, x_test, y_train, y_test = train_test_split(independent, dependent, random_state=0, train_size = .60)


#Following advice of Jason Brownlee of https://machinelearningmastery.com/data-preparation-without-data-leakage/ and
#Data Preparation for Machine Learning Data Cleaning, Feature Selection, and Data Transforms in Python
#All data preparation must be fit on the training set only

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit on the training dataset to the RobustScaler instance. Fitting the data to the training set only prevents data leakage and the test set w
scaler.fit(x_train[cleveland_numerical])

#Scale the training data using the RobustScaler instance
x_train[cleveland_numerical] = scaler.transform(x_train[cleveland_numerical])
#Scale the testing data using the RobustScaler instance
x_test[cleveland_numerical] = scaler.transform(x_test[cleveland_numerical])



#Feature Selection: Numerical Features - Binary Output

#Import Statistics module
from scipy.stats import ttest_ind

#Merge the dataframes to assist in separation of the healthy and diseased instances
cleveland_merge = x_train.join(y_train)

#Separate training set into diseased and healthy dataframes
disease = cleveland_merge[cleveland_merge['target'] == 1]
healthy = cleveland_merge[cleveland_merge['target']==0]

#Separate age features into diseased and healthy dataframes
```

```
#Separate age features into diseased and healthy dataframes
age_disease = disease['age']
age_healthy = healthy['age']

t_statistic, p_value = ttest_ind(age_disease, age_healthy)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [["age", t_statistic,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'T-Statistic','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	T-Statistic	P-Value	Accept?
	age	3.044418	0.002683	Y

```
#Feature Selection: Numerical Features - Binary Output

#Import Statistics module
from scipy.stats import ttest_ind

#Merge the dataframes to assist in separation of the healthy and diseased instances
cleveland_merge = x_train.join(y_train)

#Separate training set into diseased and healthy dataframes
disease = cleveland_merge[cleveland_merge['target'] == 1]
healthy = cleveland_merge[cleveland_merge['target']==0]

#Separate age features into diseased and healthy dataframes
age_disease = disease['trestbps']
age_healthy = healthy['trestbps']

t_statistic, p_value = ttest_ind(age_disease, age_healthy)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['trestbps', t_statistic,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'T-Statistic','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	T-Statistic	P-Value	Accept?
	trestbps	1.532062	0.127273	N

```
#Feature Selection: Numerical Features - Binary Output

#Import Statistics module
from scipy.stats import ttest_ind

#Merge the dataframes to assist in separation of the healthy and diseased instances
cleveland_merge = x_train.join(y_train)

#Separate training set into diseased and healthy dataframes
disease = cleveland_merge[cleveland_merge['target'] == 1]
healthy = cleveland_merge[cleveland_merge['target']==0]

#Separate age features into diseased and healthy dataframes
age_disease = disease['chol']
age_healthy = healthy['chol']

t_statistic, p_value = ttest_ind(age_disease, age_healthy)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['chol', t_statistic,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'T-Statistic','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

Feature	T-Statistic	P-Value	Accept?
chol	1.516038	0.131274	N

```
#Feature Selection: Numerical Features - Binary Output

#Import Statistics module
from scipy.stats import ttest_ind

#Merge the dataframes to assist in separation of the healthy and diseased instances
cleveland_merge = x_train.join(y_train)

#Separate training set into diseased and healthy dataframes
disease = cleveland_merge[cleveland_merge['target'] == 1]
healthy = cleveland_merge[cleveland_merge['target']==0]

#Separate age features into diseased and healthy dataframes
age_disease = disease['thalach']
age_healthy = healthy['thalach']

t_statistic, p_value = ttest_ind(age_disease, age_healthy)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['thalach', t_statistic,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'T-Statistic','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

Feature	T-Statistic	P-Value	Accept?
thalach	-7.385374	5.553374e-12	Y

```
#Feature Selection: Numerical Features - Binary Output

#Import Statistics module
from scipy.stats import ttest_ind

#Merge the dataframes to assist in separation of the healthy and diseased instances
cleveland_merge = x_train.join(y_train)

#Separate training set into diseased and healthy dataframes
disease = cleveland_merge[cleveland_merge['target'] == 1]
healthy = cleveland_merge[cleveland_merge['target']==0]

#Separate age features into diseased and healthy dataframes
age_disease = disease['oldpeak']
age_healthy = healthy['oldpeak']

t_statistic, p_value = ttest_ind(age_disease, age_healthy)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['oldpeak', t_statistic,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'T-Statistic','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	T-Statistic	P-Value	Accept?
	oldpeak	7.28923	9.644224e-12	Y

```
#Calculating odds ratios for categorical features with only 2 levels (i.e. binary features)

import pandas as pd
#Import fisher exact to perform odds ratio test
from scipy.stats import fisher_exact

#Construct a contingency table
cleveland_sex = pd.crosstab(x_train['sex'], y_train['target'])

# Calculate odds ratio and p-value for cleveland sex feature
odds_ratio, p_value = fisher_exact(cleveland_sex)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['sex', odds_ratio,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Odds Ratio','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Odds Ratio	P-Value	Accept?
	sex	2.913661	0.002142	Y

```
#Calculating odds ratios for categorical features with only 2 levels (i.e. binary features)

import pandas as pd
#Import fisher exact to perform odds ratio test
from scipy.stats import fisher_exact

#Construct a contingency table
cleveland_sex = pd.crosstab(x_train['sex'], y_train['target'])

# Calculate odds ratio and p-value for cleveland sex feature
odds_ratio, p_value = fisher_exact(cleveland_sex)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['sex', odds_ratio,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Odds Ratio','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

Feature	Odds Ratio	P-Value	Accept?
sex	2.913661	0.002142	Y

```
#Calculating odds ratios for categorical features with only 2 levels (i.e. binary features)

import pandas as pd
#Import fisher exact to perform odds ratio test
from scipy.stats import fisher_exact

#Construct a contingency table
cleveland_fbs = pd.crosstab(x_train['fbs'], y_train['target'])

# Calculate odds ratio and p-value for cleveland fasting blood sugar feature
odds_ratio, p_value = fisher_exact(cleveland_fbs)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['fbs', odds_ratio,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Odds Ratio','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

Feature	Odds Ratio	P-Value	Accept?
fbs	0.959488	1.0	N

```
#Calculating odds ratios for categorical features with only 2 levels (i.e. binary features)

import pandas as pd
#Import fisher exact to perform odds ratio test
from scipy.stats import fisher_exact

#Construct a contingency table
cleveland_exang = pd.crosstab(x_train['exang'], y_train['target'])

# Calculate odds ratio and p-value for cleveland fasting blood sugar feature
odds_ratio, p_value = fisher_exact(cleveland_exang)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['exang', odds_ratio,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Odds Ratio','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Odds Ratio	P-Value	Accept?
	exang	11.272727	1.369253e-11	Y

```
#Calculating chi-squared test for categorical features with more than 2 levels

#Import necessary classes
from scipy.stats import chi2_contingency
from scipy.stats import chi2

#Filter uncoded, raw data table to reflect rows from training set
cleveland_chi = data_cleveland.loc[x_train.index, :]

#Crosstabulate data in filtered, uncoded, raw data table
cleveland_cp = pd.crosstab(cleveland_chi['cp'], cleveland_chi['target'])

# Calculate chi_sq value and p-value for cleveland chest pain feature
chi_sq, p_value, dof, expected = chi2_contingency(cleveland_cp)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['cp', chi_sq,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Chi-Squared','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Chi-Squared	P-Value	Accept?
	cp	47.819745	2.326113e-10	Y

```
#Calculating chi-squared test for categorical features with more than 2 levels

#Import necessary classes
from scipy.stats import chi2_contingency
from scipy.stats import chi2

#Filter uncoded, raw data table to reflect rows from training set
cleveland_chi = data_cleveland.loc[x_train.index, :]

#Crosstabulate data in filtered, uncoded, raw data table
cleveland_cp = pd.crosstab(cleveland_chi['restecg'], cleveland_chi['target'])

# Calculate chi_sq value and p-value for cleveland restecg feature
chi_sq, p_value, dof, expected = chi2_contingency(cleveland_cp)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['restecg', chi_sq,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Chi-Squared','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Chi-Squared	P-Value	Accept?
	restecg	3.831708	0.147216	N

```
#Calculating chi-squared test for categorical features with more than 2 levels

#Import necessary classes
from scipy.stats import chi2_contingency
from scipy.stats import chi2

#Filter uncoded, raw data table to reflect rows from training set
cleveland_chi = data_cleveland.loc[x_train.index, :]

#Crosstabulate data in filtered, uncoded, raw data table
cleveland_cp = pd.crosstab(cleveland_chi['slope'], cleveland_chi['target'])

# Calculate chi_sq value and p-value for cleveland restecg feature
chi_sq, p_value, dof, expected = chi2_contingency(cleveland_cp)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['slope', chi_sq,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Chi-Squared','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Chi-Squared	P-Value	Accept?
	slope	42.934516	4.752129e-10	Y

```
#Calculating chi-squared test for categorical features with more than 2 levels

#Import necessary classes
from scipy.stats import chi2_contingency
from scipy.stats import chi2

#Filter uncoded, raw data table to reflect rows from training set
cleveland_chi = data_cleveland.loc[x_train.index, :]

#Crosstabulate data in filtered, uncoded, raw data table
cleveland_cp = pd.crosstab(cleveland_chi['ca'], cleveland_chi['target'])

# Calculate chi_sq value and p-value for cleveland restecg feature
chi_sq, p_value, dof, expected = chi2_contingency(cleveland_cp)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['ca', chi_sq,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Chi-Squared','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Chi-Squared	P-Value	Accept?
	ca	37.765747	3.168368e-08	Y

```
#Calculating chi-squared test for categorical features with more than 2 levels

#Import necessary classes
from scipy.stats import chi2_contingency
from scipy.stats import chi2

#Filter uncoded, raw data table to reflect rows from training set
cleveland_chi = data_cleveland.loc[x_train.index, :]

#Crosstabulate data in filtered, uncoded, raw data table
cleveland_cp = pd.crosstab(cleveland_chi['thal'], cleveland_chi['target'])

# Calculate chi_sq value and p-value for cleveland restecg feature
chi_sq, p_value, dof, expected = chi2_contingency(cleveland_cp)

#Test p-value
if p_value < 0.05:
    test = "Y"
else:
    test = "N"

#Organize performance metrics into a list
results_list = [['thal', chi_sq,p_value,test]]

#Create dataframe of performance metrics
simple_results= pd.DataFrame(results_list, columns=['Feature', 'Chi-Squared','P-Value', 'Accept?'])
simple_results.index = [""]

simple_results
```

	Feature	Chi-Squared	P-Value	Accept?
	thal	50.54244	1.058885e-11	Y

```
#Modify training and test records

#Features to drop based on simple filters
features_remove = ['fbs', 'restecg_1','restecg_2']

#Remove selected features from x_train and x_test to evaluate model based on hold-out method
x_train_simple = x_train.drop(columns=features_remove)
x_test_simple = x_test.drop(columns=features_remove)


#Modify consistent dataset for k-fold cross-validation

#Features to drop based on simple filters
features_remove = ['fbs', 'restecg_1','restecg_2']

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

#Initialize RobustScaler
scaler = RobustScaler()

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])
```

```
#Decision Tree

from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the decision tree classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_decision_tree.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the cleveland_decision_tree model
target_pred_decision_tree = cleveland_decision_tree.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1),3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1),3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1),3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree),3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0),3)

#Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree,precision_decision_tree,recall_decision_tre

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_decision_tree = performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
Decision Tree	71.31	0.767	0.745	0.661	0.701	0.43	0.010111	263.77

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the decision tree classifier
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers
param_grid = {'criterion': ['gini', 'entropy'],
              'max_depth': [None, 1,2,10, 20, 30],
              'min_samples_split': [2, 5, 10,15,18],
              'min_samples_leaf': [1, 2, 4]}

#Initialize the GridSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_dt = GridSearchCV(cleveland_decision_tree , param_grid, cv=10)
grid_cleveland_dt.fit(x_train_simple, y_train)

#Output the best parameters, the model is optimized based on accuracy score
best_params_dt = grid_cleveland_dt.best_params_
print(best_params_dt)

#Fit the model using athe best parameters
cleveland_decision_tree = DecisionTreeClassifier(**best_params_dt, random_state=42)
cleveland_decision_tree.fit(x_train_simple, y_train)

# Use the best model for predictions and recalculate metrics
target_pred_decision_tree = cleveland_decision_tree.predict(x_test_simple)

#Calculate Performance Metrics
accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1), 3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1), 3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1), 3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree), 3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0), 3)

# Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree, precision_decision_tree, recall_decision_t

# Create a DataFrame of performance metrics
grid_dt_pm = pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_dt_pm.index = [""]

grid_dt_pm

#There are improvements to the metrics with the exception of specificity
```

```
{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 15}
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Decision Tree	75.41	0.833	0.808	0.677	0.737	0.516

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_decision_tree = DecisionTreeClassifier(criterion = 'gini', max_depth = None, min_samples_leaf = 4, min_samples_split = 15, random_

# Initialize RepeatedStratifiedKFold (will complete 30 rounds in total)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_decision_tree.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_decision_tree = cleveland_decision_tree.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_decision_tree)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_decision_tree)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_decision_tree, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       74.91
Precision      76.23
Recall         66.43
F1 Score       70.55
MCC            49.71
Specificity    82.17
dtype: float64

Standard Deviation Metrics:
Accuracy        7.45
Precision       9.88
Recall         11.54
F1 Score        9.57
MCC            15.29
Specificity     8.41
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Random Forest

from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_random_forest.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the cleveland_random_forest model
target_pred_random_forest = cleveland_random_forest.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

#Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest,precision_random_forest,recall_random_fores

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_random_forest = performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
Random Forest	72.95	0.817	0.784	0.645	0.708	0.468	0.407271	201.67

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 42 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_grid = {'n_estimators': [25, 50, 100, 150],
              'max_features': ['sqrt', 'log2', None],
              'max_depth': [3, 6, 9],
              'max_leaf_nodes': [3, 6, 9]}

#Initialize the GridSearchCV class using the random forest, the parameter grid and a 10-fold cross-validation
grid_cleveland_rf = GridSearchCV(cleveland_random_forest , param_grid, cv=10)
grid_cleveland_rf.fit(x_train_simple, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_rf = grid_cleveland_rf.best_params_
print(best_params_rf)

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(**best_params_dt, random_state=42)
cleveland_random_forest.fit(x_train_simple, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test_simple)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for

# Create a DataFrame of performance metrics
grid_rf_pm = pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_rf_pm.index = [""]

grid_rf_pm
```

```
{'max_depth': 6, 'max_features': 'sqrt', 'max_leaf_nodes': 9, 'n_estimators': 150}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
Random Forest	74.59	0.833	0.804	0.661	0.726	0.501

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_random_forest = RandomForestClassifier(random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_random_forest.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_random_forest= cleveland_random_forest.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_random_forest)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_random_forest, pos_label=1),4))
    recall_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=1),4))
    f1_list.append(round(f1_score(y_test_fold, target_pred_random_forest, pos_label=1),4))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_random_forest),4))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=0),4))


#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       80.31
Precision      82.04
Recall         73.96
F1 Score       77.11
MCC            61.04
Specificity    85.86
dtype: float64

Standard Deviation Metrics:
Accuracy        6.24
Precision       8.20
Recall         12.72
F1 Score        8.29
MCC            12.87
Specificity     7.22
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Naive Bayes

#The application of Naive Bayes in the paper is unclear. The dataset contains both categorical and numerical (i.e. continuous numerical) fea
#Here we will applying different Naive Bayes classifiers to the categorical and numerical features
from sklearn.naive_bayes import  CategoricalNB, GaussianNB

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the naive bayes models and assign to variable
cleveland_naive_numerical = GaussianNB()
cleveland_naive_categorical = CategoricalNB()

#List categorical and numerical feature names
cleveland_categorical_nb = ['sex', 'exang','cp_1', 'cp_2', 'cp_3', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'ca_3','thal_2', 'thal_3']
cleveland_numerical_nb = ['age','trestbps','chol','thalach','oldpeak']

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

# Fit the categorical features in the training data set to the categorical naive bayes model
cleveland_naive_categorical.fit(x_train_simple[cleveland_categorical_nb], y_train.values.ravel())

#Fit the numerical features in the training data set to the gaussian naive bayes model
cleveland_naive_numerical.fit(x_train_simple[cleveland_numerical_nb], y_train.values.ravel())

# Predict probabilities for using categorical and numerical features
probability_categorical = cleveland_naive_categorical.predict_proba(x_test_simple[cleveland_categorical_nb])
probability_numerical= cleveland_naive_numerical.predict_proba(x_test_simple[cleveland_numerical_nb])

#Combine the probabilities using the product rule
total_probability = probability_categorical * probability_numerical

#We can use this code to select the class that has the greatest probability for a given row
import numpy as np
target_pred_naive = np.argmax(total_probability, axis=1)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

#Calculate performance metrics
accuracy_naive =  round(accuracy_score(y_test, target_pred_naive),4)*100
precision_naive = round(precision_score(y_test, target_pred_naive, pos_label=1),3)
recall_naive = round(recall_score(y_test, target_pred_naive, pos_label=1),3)
f1_score_naive = round(f1_score(y_test, target_pred_naive, pos_label=1),3)
mcc_naive = round(matthews_corrcoef(y_test, target_pred_naive),3)
specificity_naive = round(recall_score(y_test, target_pred_naive, pos_label=0),3)

#Organize performance metrics into a list
performance_naive = [["Naive Bayes", accuracy_naive,specificity_naive,precision_naive,recall_naive,f1_score_naive,mcc_naive, computational_t

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_naive, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', 'MCC', '
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_naive_bayes = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	Naive Bayes	75.41	0.817	0.796	0.694	0.741	0.514	0.046504	202.7

```
from sklearn.model_selection import StratifiedKFold
from sklearn.naive_bayes import CategoricalNB, GaussianNB

# List categorical and numerical feature names
cleveland_categorical_nb = ['sex', 'exang', 'cp_1', 'cp_2', 'cp_3', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'ca_3', 'thal_2', 'thal_3']
cleveland_numerical_nb = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

# Initialize Naive Bayes models and assign to variables
cleveland_naive_categorical = CategoricalNB()
cleveland_naive_numerical = GaussianNB()

# Combine categorical and numerical features
x_naive = x_simple[cleveland_categorical_nb + cleveland_numerical_nb]

# Initialize StratifiedKFold
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

# Iterate through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_naive, y_simple):
    # Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_naive.iloc[train_index], x_naive.iloc[test_index]
    # Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    # Fit the training data set to the categorical and numerical Naive Bayes models
    cleveland_naive_categorical.fit(x_train_fold[cleveland_categorical_nb], y_train_fold.values.ravel())
    cleveland_naive_numerical.fit(x_train_fold[cleveland_numerical_nb], y_train_fold.values.ravel())

    # Predict probabilities for using categorical and numerical features
    probability_categorical = cleveland_naive_categorical.predict_proba(x_test_fold[cleveland_categorical_nb])
    probability_numerical = cleveland_naive_numerical.predict_proba(x_test_fold[cleveland_numerical_nb])

    # Combine the probabilities using the product rule
    total_probability = probability_categorical * probability_numerical

    # Use this code to select the class that has the greatest probability for a given row
    target_pred_naive = np.argmax(total_probability, axis=1)

    # Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_naive) * 100, 2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_naive, pos_label=1) * 100, 2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_naive, pos_label=1) * 100, 2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_naive, pos_label=1) * 100, 2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_naive) * 100, 2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_naive, pos_label=0) * 100, 2))

# Create a DataFrame to store metrics for each fold
performance_metrics_naive = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

# Calculate mean and standard deviation across folds
mean_metrics_naive = round(performance_metrics_naive.mean(), 2)
std_metrics_naive = round(performance_metrics_naive.std(), 2)

# Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics_naive)
print('\nStandard Deviation Metrics:')
print(std_metrics_naive)

# performance_metrics_naive
```

```
Mean Metrics:
Accuracy       83.48
Precision      82.89
Recall         81.21
F1 Score       81.84
MCC            66.97
Specificity    85.33
dtype: float64

Standard Deviation Metrics:
Accuracy       4.72
Precision      7.32
Recall         6.45
F1 Score       5.33
MCC            9.65
Specificity    7.30
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Logistic Regression

from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_logistic_reg = LogisticRegression(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_logistic_reg.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the cleveland_logistic_reg model
target_pred_logistic_reg = cleveland_logistic_reg.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_logistic_reg =  round(accuracy_score(y_test, target_pred_logistic_reg),4)*100
precision_logistic_reg = round(precision_score(y_test, target_pred_logistic_reg, pos_label=1),3)
recall_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=1),3)
f1_score_logistic_reg = round(f1_score(y_test, target_pred_logistic_reg, pos_label=1),3)
mcc_logistic_reg = round(matthews_corrcoef(y_test, target_pred_logistic_reg),3)
specificity_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=0),3)

#Organize performance metrics into a list
performance_logistic_reg = [["Logistic Regression", accuracy_logistic_reg, specificity_logistic_reg,precision_logistic_reg,recall_logistic_r

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_logistic_reg, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', '
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_logistic_reg = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	Logistic Regression	81.97	0.85	0.845	0.79	0.817	0.641	0.013889	261.21

```
# Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the logistic regression classifier and assign it to the variable cleveland_logistic_reg. Assign random_state 42 to reproduce re
cleveland_logistic_reg = LogisticRegression(random_state=42)

# A parameter grid for logistic regression
param_grid = {'max_iter':[100,110,120,130,140],
              'C' : [1.0,1.5,2.0,2.5]}

# Initialize the GridSearchCV class using the logistic regression model, the parameter grid, and a 10-fold cross-validation
grid_cleveland_lr = GridSearchCV(cleveland_logistic_reg, param_grid, cv=10)
grid_cleveland_lr.fit(x_train_simple, y_train.values.ravel())

# Output the best parameters based on accuracy score
best_params_lr = grid_cleveland_lr.best_params_
print(best_params_lr)

#Fit the model using the best parameters
cleveland_logistic_reg = LogisticRegression(**best_params_lr, random_state=42)
cleveland_logistic_reg.fit(x_train_simple, y_train.values.ravel())

#Use the best model to calculate predictions
target_pred_logistic_reg = cleveland_logistic_reg.predict(x_test_simple)

# Calculate Performance Metrics
accuracy_logistic_reg = round(accuracy_score(y_test, target_pred_logistic_reg), 4) * 100
precision_logistic_reg = round(precision_score(y_test, target_pred_logistic_reg, pos_label=1), 3)
recall_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=1), 3)
f1_score_logistic_reg = round(f1_score(y_test, target_pred_logistic_reg, pos_label=1), 3)
mcc_logistic_reg = round(matthews_corrcoef(y_test, target_pred_logistic_reg), 3)
specificity_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=0), 3)

# Organize performance metrics into a list
performance_logistic_reg = [["Logistic Regression", accuracy_logistic_reg, specificity_logistic_reg, precision_logistic_reg, recall_logistic

# Create a DataFrame of performance metrics
grid_lr_pm = pd.DataFrame(performance_logistic_reg, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_lr_pm.index = [""]

grid_lr_pm
```

```
{'C': 1.0, 'max_iter': 100}
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Logistic Regression	78.69	0.817	0.81	0.758	0.783	0.575

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
import numpy as np

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_logistic_reg = LogisticRegression(C= 1.0, max_iter= 100, random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_logistic_reg.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_logistic_reg= cleveland_logistic_reg.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_logistic_reg)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_logistic_reg, pos_label=1),2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_logistic_reg, pos_label=1),2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_logistic_reg, pos_label=1),2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_logistic_reg),2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_logistic_reg, pos_label=0),2))


#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       85.36
Precision       0.86
Recall          0.82
F1 Score        0.83
MCC             0.71
Specificity     0.89
dtype: float64

Standard Deviation Metrics:
Accuracy       5.55
Precision      0.07
Recall         0.09
F1 Score       0.07
MCC            0.11
```

```
Specificity    0.07
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Support Vector Machine

from sklearn.model_selection import cross_val_score
from sklearn import svm

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_support_vector = svm.SVC(kernel='linear', random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_support_vector.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the model
target_pred_support_vector = cleveland_support_vector.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_support_vector =  round(accuracy_score(y_test, target_pred_support_vector),4)*100
precision_support_vector = round(precision_score(y_test, target_pred_support_vector, pos_label=1),3)
recall_support_vector = round(recall_score(y_test, target_pred_support_vector, pos_label=1),3)
f1_score_support_vector = round(f1_score(y_test, target_pred_support_vector, pos_label=1),3)
mcc_support_vector = round(matthews_corrcoef(y_test, target_pred_support_vector),3)
specificity_support_vector = round(recall_score(y_test, target_pred_support_vector, pos_label=0),3)

#Organize performance metrics into a list
performance_support_vector = [["Support Vector", accuracy_support_vector, specificity_support_vector,precision_support_vector,recall_support

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_support_vector, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_support_vector = performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
Support Vector	79.51	0.867	0.849	0.726	0.783	0.598	0.033995	244.57

```
# Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# Initialize the SVM classifier and assign it to the variable cleveland_svm. Assign random_state 42 to reproduce results
cleveland_svm = SVC(random_state=42)

# A parameter grid for SVM
param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'degree': [2, 3, 4],
    'gamma': ['scale', 'auto']
}

# Initialize the GridSearchCV class using the SVM model, the parameter grid, and a 10-fold cross-validation
grid_cleveland_svm = GridSearchCV(cleveland_svm, param_grid, cv=10)
grid_cleveland_svm.fit(x_train_simple, y_train.values.ravel())

# Output the best parameters based on accuracy score
best_params_svm = grid_cleveland_svm.best_params_
print(best_params_svm)

# Fit the model using the best parameters
cleveland_svm = SVC(**best_params_svm, random_state=42)
cleveland_svm.fit(x_train_simple, y_train.values.ravel())

# Use the best model to calculate predictions
target_pred_svm = cleveland_svm.predict(x_test_simple)

# Calculate Performance Metrics
accuracy_svm = round(accuracy_score(y_test, target_pred_svm), 4) * 100
precision_svm = round(precision_score(y_test, target_pred_svm, pos_label=1), 3)
recall_svm = round(recall_score(y_test, target_pred_svm, pos_label=1), 3)
f1_score_svm = round(f1_score(y_test, target_pred_svm, pos_label=1), 3)
mcc_svm = round(matthews_corrcoef(y_test, target_pred_svm), 3)
specificity_svm = round(recall_score(y_test, target_pred_svm, pos_label=0), 3)

# Organize performance metrics into a list
performance_svm = [["Support Vector Machine", accuracy_svm, specificity_svm, precision_svm, recall_svm, f1_score_svm, mcc_svm]]

# Create a DataFrame of performance metrics
grid_svm_pm = pd.DataFrame(performance_svm, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_svm_pm.index = [""]

grid_svm_pm
```

```
{'C': 100, 'degree': 2, 'gamma': 'auto', 'kernel': 'sigmoid'}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
Support Vector Machine	77.87	0.783	0.787	0.774	0.78	0.557

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn import svm
import numpy as np

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_support_vector = svm.SVC(kernel='linear', random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]
```

```
        #Fit the training data set to the classifier
        cleveland_support_vector.fit(x_train_fold, y_train_fold.values.ravel())

        #Predict the presence of heart disease by inputting the test data
        target_pred_support_vector= cleveland_support_vector.predict(x_test_fold)

        #Calculate performance metrics for the current fold
        accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_support_vector)*100,2))
        precision_list.append(round(precision_score(y_test_fold, target_pred_support_vector, pos_label=1)*100,2))
        recall_list.append(round(recall_score(y_test_fold, target_pred_support_vector, pos_label=1)*100,2))
        f1_list.append(round(f1_score(y_test_fold, target_pred_support_vector, pos_label=1)*100,2))
        mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_support_vector)*100,2))
        specificity_list.append(round(recall_score(y_test_fold, target_pred_support_vector, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       83.95
Precision      85.67
Recall         78.90
F1 Score       81.67
MCC            68.22
Specificity    88.30
dtype: float64

Standard Deviation Metrics:
Accuracy        5.19
Precision       7.90
Recall          9.54
F1 Score        6.37
MCC            10.57
Specificity     6.98
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Gradient Boosting

#Import gradient boosting classifier from scikit-learn libraries
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the support vector machine classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_gradient.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the model
```

```
target_pred_gradient = cleveland_gradient.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient", accuracy_gradient, specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mcc_gradien

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', 'MCC',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
Gradient	77.87	0.817	0.807	0.742	0.773	0.56	0.108624	203.93

```
#Gradient Boosting

#Initialize the gradient boosting classifier and assign to variable cleveland_gradient. Assign random_state 42 to reproduce results
cleveland_gradient = GradientBoostingClassifier(random_state=42)

#A parameter grid was created using selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 100, 200],
              'learning_rate': [0.01, 0.1, 0.2],
              'max_depth': [3, 4, 5],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1, 2, 4]}

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train, y_train.values.ravel()).predict(x_test)

#Initialize the GridSearchCV class using the gradient boosting, the parameter grid and a 10-fold cross-validation
grid_cleveland_gb = GridSearchCV(cleveland_gradient, param_dist, cv=10)
grid_cleveland_gb.fit(x_train_simple, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_gb = grid_cleveland_gb.best_params_
print(best_params_gb)

#Fit the model using athe best parameters
cleveland_gradient= GradientBoostingClassifier(**best_params_gb, random_state=42)
cleveland_gradient.fit(x_train_simple, y_train.values.ravel())

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test_simple)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

```
{'learning_rate': 0.2, 'max_depth': 3, 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 50}
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Gradient Boosting	72.95	0.783	0.764	0.677	0.718	0.463

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
import numpy as np

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       79.54
Precision      79.85
Recall         75.05
F1 Score       76.94
MCC            59.23
Specificity    83.38
dtype: float64

Standard Deviation Metrics:
Accuracy        5.77
Precision       8.30
Recall          9.70
F1 Score        6.80
MCC            11.87
Specificity     7.71
dtype: float64
```

```
#Repeat classifier and calculate performance metrics using only selected features

#XGBoost

#Import xgb boosting classifier from scikit-learn libraries
import xgboost as xgb
from sklearn.model_selection import cross_val_score

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorical=

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_xgb.fit(x_train_simple, y_train.values.ravel()).predict(x_test_simple)

#Predict the presence of heart disese by inputting the test data into the model
target_pred_xgb = cleveland_xgb.predict(x_test_simple)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb, specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb, computational_time, memory_usage]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score', 'MCC', 'Com
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	XGBoost	77.05	0.833	0.815	0.71	0.759	0.547	0.760564	222.71

```
#XGBoost

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(enable_categorical=True, seed= 42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0]
}

#Initialize the RandomizedSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_xgb = GridSearchCV(cleveland_xgb , param_grid, cv=10)
grid_cleveland_xgb.fit(x_train_simple, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_xgb = grid_cleveland_xgb.best_params_
print(best_params_xgb)

#Fit the model using athe best parameters
cleveland_xgb = xgb.XGBClassifier(**best_params_xgb, seed=42)
cleveland_xgb.fit(x_train_simple, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_xgb.predict(x_test_simple)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb,specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb= performance_metrics

performance_metrics
```

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
import xgboost as xgb
import numpy as np

#Initialize the classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       79.54
Precision      79.85
Recall         75.05
F1 Score       76.94
MCC            59.23
Specificity    83.38
dtype: float64

Standard Deviation Metrics:
Accuracy        5.77
Precision       8.30
Recall          9.70
F1 Score        6.80
MCC            11.87
Specificity     7.71
dtype: float64
```

```
#Ensemble Model

#Import VotingClassifier to combine model predictions
from sklearn.ensemble import VotingClassifier


# Initialize remaining classifier types and fit the entire training setata
ensemble_random_forest = RandomForestClassifier(random_state = 42)
ensemble_random_forest.fit(x_train_simple, y_train.values.ravel())


ensemble_gradient = GradientBoostingClassifier(random_state = 42)
ensemble_gradient.fit(x_train_simple, y_train.values.ravel())

#Specify Algorithms and initialize ensemble model using a soft voting classifier
algorithms = [('ensemble_random_forest',ensemble_random_forest),('ensemble_gradient',ensemble_gradient)]
ensemble = VotingClassifier(estimators=algorithms, voting='soft')

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Now we can fit each model to voting classifier instance, selecting the categorical variables for naive bayes categorical models
ensemble.fit(
    np.column_stack([ensemble_gradient.predict_proba(x_train_simple),
                     ensemble_random_forest.predict_proba(x_train_simple)]),
    y_train.values.ravel()
)

#With the ensemble model, we can make predictions on the target values
target_pred_ensemble = ensemble.predict(
    np.column_stack([ensemble_gradient.predict_proba(x_test_simple),
                     ensemble_random_forest.predict_proba(x_test_simple)])
)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

accuracy_ensemble =  round(accuracy_score(y_test, target_pred_ensemble),4)*100
precision_ensemble = round(precision_score(y_test, target_pred_ensemble, pos_label=1),3)
recall_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=1),3)
f1_score_ensemble = round(f1_score(y_test, target_pred_ensemble, pos_label=1),3)
mcc_ensemble = round(matthews_corrcoef(y_test, target_pred_ensemble),3)
specificity_ensemble = round(recall_score(y_test, target_pred_ensemble, pos_label=0),3)

#Organize performance metrics into a list
performance_ensemble = [["Ensemble", accuracy_ensemble,specificity_ensemble,precision_ensemble,recall_ensemble,f1_score_ensemble,mcc_ensembl

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_ensemble, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_ensemble = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	Ensemble	73.77	0.8	0.778	0.677	0.724	0.481	0.882053	261.97

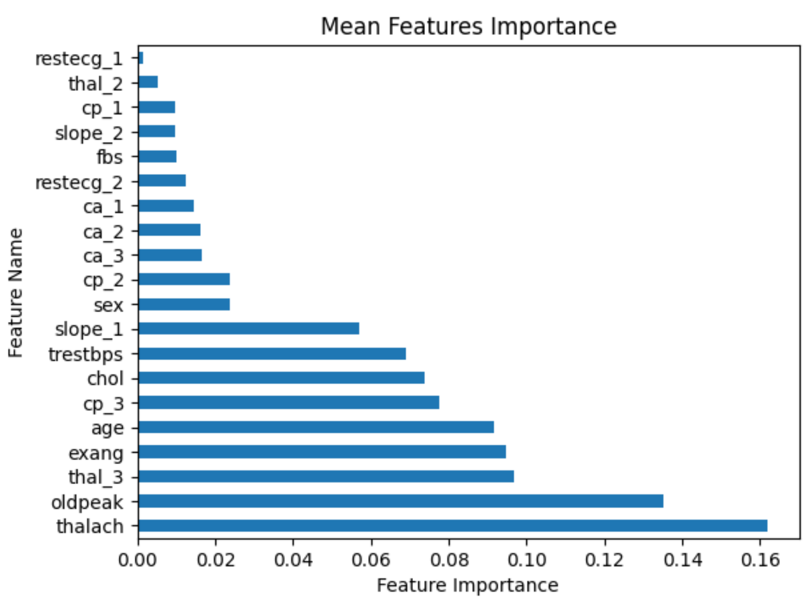
```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.ensemble import VotingClassifier
from sklearn.naive_bayes import CategoricalNB, GaussianNB
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef

# Initialize classifiers

ensemble_random_forest = RandomForestClassifier(random_state=42)
ensemble_gradient = GradientBoostingClassifier(random_state=42)
```

```
ensemble_gradient = GradientBoostingClassifier(random_state=42)

# Specify Algorithms and initialize ensemble model using a soft voting classifier
algorithms = [('ensemble_random_forest', ensemble_random_forest),('ensemble_gradient', ensemble_gradient)]

ensemble = VotingClassifier(estimators=algorithms, voting='soft')

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

# We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    # Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    # Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]


    # Fit the entire training set
    ensemble_random_forest.fit(x_train_fold, y_train_fold.values.ravel())
    ensemble_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Specify Algorithms and initialize ensemble model using a soft voting classifier
    algorithms = [('ensemble_random_forest',ensemble_random_forest),('ensemble_gradient',ensemble_gradient)]
    ensemble = VotingClassifier(estimators=algorithms, voting='soft')

    #Now we can fit each model to voting classifier instance, selecting the categorical variables for naive bayes categorical models
    ensemble.fit(np.column_stack([ensemble_random_forest.predict_proba(x_train_fold),ensemble_gradient.predict_proba(x_train_fold),]),y_train_

    #With the ensemble model, we can make predictions on the target values
    target_pred_ensemble = ensemble.predict(np.column_stack([ensemble_random_forest.predict_proba(x_test_fold), ensemble_gradient.predict_prol

    #Calculate performance metrics for the current fold
    accuracy_list.append(accuracy_score(y_test_fold, target_pred_ensemble))
    precision_list.append(precision_score(y_test_fold, target_pred_ensemble, pos_label=1))
    recall_list.append(recall_score(y_test_fold, target_pred_ensemble, pos_label=1))
    f1_list.append(f1_score(y_test_fold, target_pred_ensemble, pos_label=1))
    mcc_list.append(matthews_corrcoef(y_test_fold, target_pred_ensemble))
    specificity_list.append(recall_score(y_test_fold, target_pred_ensemble, pos_label=0))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = performance_metrics_fold.mean()
std_metrics = performance_metrics_fold.std()

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       0.809713
Precision      0.805417
Recall         0.777839
F1 Score       0.786964
MCC            0.621872
Specificity    0.838235
```

```
dtype: float64

Standard Deviation Metrics:
Accuracy       0.068794
Precision      0.082283
Recall         0.117434
F1 Score       0.083133
MCC            0.140508
Specificity    0.075333
dtype: float64
```

```
#Feature Selection using Embedded Methods

#Random Forest Feature Importance Plot
from sklearn.ensemble import RandomForestClassifier


#Initialize the random forest classifier. Assign random_state 42 to reproduce results
random_forest_embedded = RandomForestClassifier(random_state=42)

#Fit the training data set to the random forest classifier
random_forest_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Extract feature importance plot
(pd.Series(random_forest_embedded.feature_importances_, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

import matplotlib.pyplot as plt
# Add plot labels
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Mean Features Importance')
plt.show()

#Based on feature importance plot, all features below slope_1 appear to be most relevant

#Drop least releavnt features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['restecg_1', 'thal_2','cp_1','slope_2','fbs','restecg_2','ca_1','ca_2','ca_3','cp_2','sex']

#Remove selected features from x_train and x_test
x_train_embedded = x_train.drop(columns=features_remove)
x_test_embedded = x_test.drop(columns=features_remove)
```



```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])


#Repeat Random Forest Classifier and Output Statistics
#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 0 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#Fit the training data set to the random forest classifier
cleveland_random_forest.fit(x_train_embedded, y_train.values.ravel()).predict(x_test_embedded)

#Predict the presence of heart disese by inputting the test data into the cleveland_random_forest model
target_pred_random_forest = cleveland_random_forest.predict(x_test_embedded)

accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

#Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest,specificity_random_forest,precision_random_forest,recall_random_forest

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_random_forest = performance_metrics

performance_metrics
```

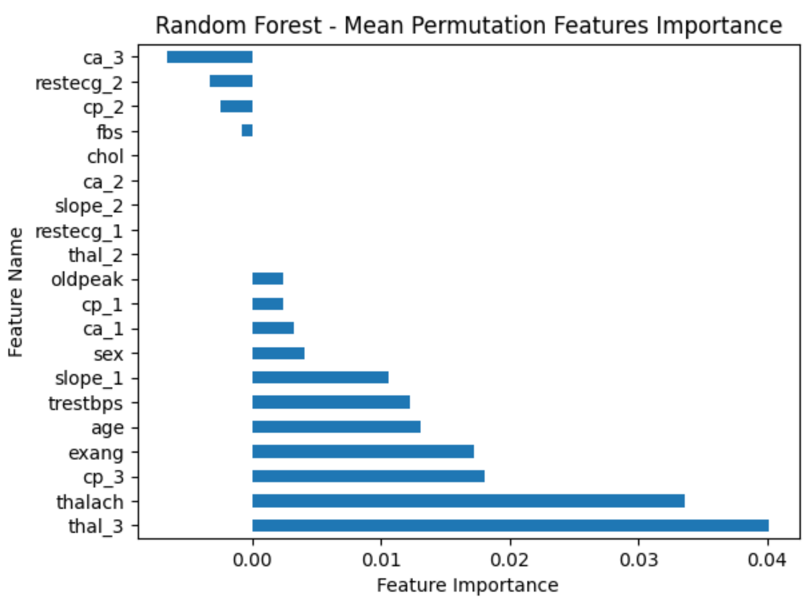
	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Random Forest	71.31	0.783	0.755	0.645	0.696	0.432

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 42 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_grid = {'n_estimators': [25, 50, 100, 150],
              'max_features': ['sqrt', 'log2', None],
              'max_depth': [3, 6, 9],
              'max_leaf_nodes': [3, 6, 9]}

#Initialize the GridSearchCV class using the random forest, the parameter grid and a 10-fold cross-validation
grid_cleveland_rf = GridSearchCV(cleveland_random_forest , param_grid, cv=10)
grid_cleveland_rf.fit(x_train_embedded, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_rf = grid_cleveland_rf.best_params_
print(best_params_rf)

#Fit the model using athe best parameters
cleveland_random_forest = RandomForestClassifier(**best_params_dt, random_state=42)
cleveland_random_forest.fit(x_train_embedded, y_train.values.ravel())

# Use the best model for predictions and recalculate metrics
target_pred_random_forest = cleveland_random_forest.predict(x_test_embedded)

#Calculate Performance Metrics
accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

# Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest, precision_random_forest, recall_random_for

# Create a DataFrame of performance metrics
grid_rf_pm = pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_rf_pm.index = [""]

grid_rf_pm
```

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_random_forest = RandomForestClassifier(random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_random_forest.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_random_forest= cleveland_random_forest.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_random_forest)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_random_forest)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=0)*100,2))


#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       77.00
Precision      76.90
Recall         72.44
F1 Score       74.07
MCC            54.23
Specificity    81.02
dtype: float64

Standard Deviation Metrics:
Accuracy        5.19
Precision       7.59
Recall         10.07
F1 Score        6.50
MCC            10.53
Specificity     8.06
dtype: float64
```

```
#Feature Selection using Embedded Methods

#Random Forest Feature Importance Plot
from sklearn.ensemble import RandomForestClassifier
from sklearn.inspection import permutation_importance #Neccessary class for permutation importances

#Initialize the random forest classifier. Assign random_state 42 to reproduce results
random_forest_embedded = RandomForestClassifier(random_state=42)

#Fit the training data set to the random forest classifier
random_forest_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Calculate the feature importance based on the permutation method
random_permutation= permutation_importance(random_forest_embedded, x_test, y_test, n_repeats=10, random_state=42)

#Extract feature importance plot
(pd.Series(random_permutation.importances_mean, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

import matplotlib.pyplot as plt
# Add plot labels
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Random Forest - Mean Permutation Features Importance')
plt.show()

#Based on feature importance plot, all features below ca_1 appear to be most relevant

#Drop least releavnt features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['restecg_2','chol','cp_2','fbs','ca_2','slope_2','thal_2','restecg_1','cp_1','sex','oldpeak']

#Remove selected features from x_train and x_test
x_train_perm = x_train.drop(columns=features_remove)
x_test_perm = x_test.drop(columns=features_remove)
```



```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'thalach']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])


#Repeat Random Forest Classifier and Output Statistics
#Initialize the random forest classifier and assign to variable cleveland_random_forest. Assign random_state 0 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the random forest classifier
cleveland_random_forest.fit(x_train_perm, y_train.values.ravel()).predict(x_test_perm)

#Predict the presence of heart disese by inputting the test data into the cleveland_random_forest model
target_pred_random_forest = cleveland_random_forest.predict(x_test_perm)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

#Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest,specificity_random_forest,precision_random_forest,recall_random_forest

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_random_forest = performance_metrics

performance_metrics
```

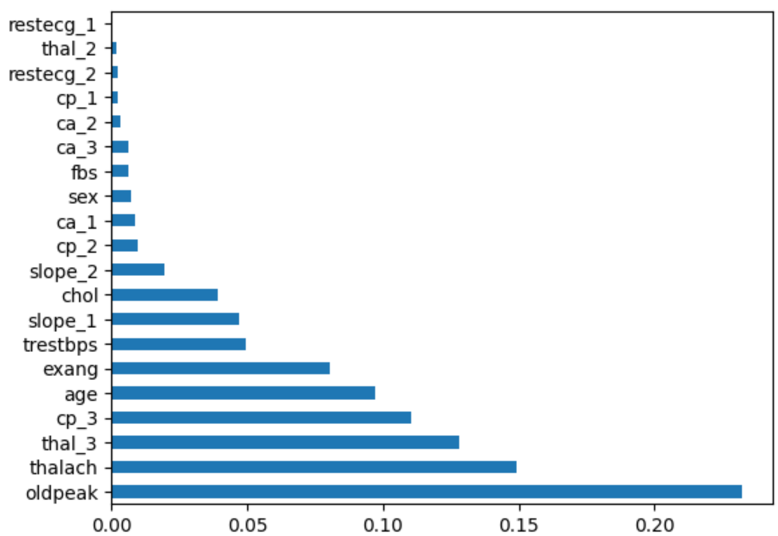
	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computation Speed	Memory Usage
	Random Forest	76.23	0.8	0.789	0.726	0.756	0.527	0.334595	272.76

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_random_forest = RandomForestClassifier(random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_random_forest.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_random_forest= cleveland_random_forest.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_random_forest)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_random_forest)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=0)*100,2))


#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       78.88
Precision      81.57
Recall         71.06
F1 Score       75.23
MCC            58.29
Specificity    85.66
dtype: float64

Standard Deviation Metrics:
Accuracy        6.24
Precision       9.58
Recall         11.91
F1 Score        7.96
MCC            12.95
```

```
Specificity     8.67
dtype: float64
```

```
#Feature Selection using Embedded Methods

#Gradient Boosting Feature Importance Plot
from sklearn.ensemble import GradientBoostingClassifier

#Initialize the random forest classifier. Assign random_state 42 to reproduce results
gradient_embedded =GradientBoostingClassifier(random_state=42)

#Fit the training data set to the random forest classifier
gradient_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Extract feature importance plot
(pd.Series(gradient_embedded.feature_importances_, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

#Based on feature importance plot, all features below slope_1 appear to be most relevant

#Drop least releavnt features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['restecg_1', 'thal_2','restecg_2', 'cp_1','ca_2','ca_3','fbs','sex', 'ca_1','cp_2','slope_2']

#Remove selected features from x_train and x_test
x_train_embedded = x_train.drop(columns=features_remove)
x_test_embedded = x_test.drop(columns=features_remove)
```




```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])
```

```
#Gradient Boosting

#Import gradient boosting classifier from scikit-learn libraries
from sklearn.ensemble import GradientBoostingClassifier

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 42 to reproduce result
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train_embedded, y_train.values.ravel()).predict(x_test_embedded)

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test_embedded)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

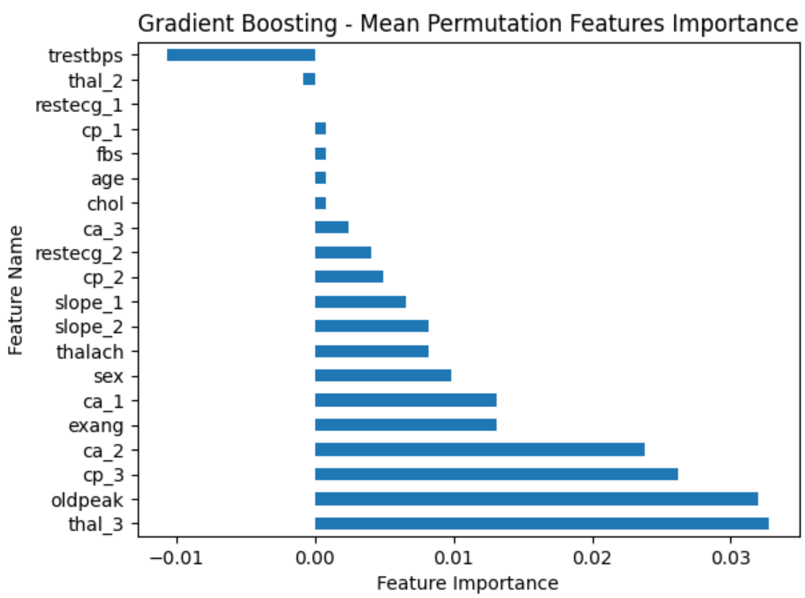
	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
	Gradient Boosting	70.49	0.733	0.724	0.677	0.7	0.411

```
#Gradient Boosting

#Initialize the gradient boosting classifier and assign to variable cleveland_gradient. Assign random_state 42 to reproduce results
cleveland_gradient = GradientBoostingClassifier(random_state=42)

#A parameter grid was created using selected integers to cycle through in order to optimize the accuracy
#These features were selected based on the values available in the sklearn documentation
param_dist = {'n_estimators': [50, 100, 200],
              'learning_rate': [0.01, 0.1, 0.2],
              'max_depth': [3, 4, 5],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1, 2, 4]}

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train, y_train.values.ravel()).predict(x_test)

#Initialize the GridSearchCV class using the gradient boosting, the parameter grid and a 10-fold cross-validation
grid_cleveland_gb = GridSearchCV(cleveland_gradient, param_dist, cv=10)
grid_cleveland_gb.fit(x_train_embedded, y_train.values.ravel())

#Output the best parameters, the model is optimized based on accuracy score
best_params_gb = grid_cleveland_gb.best_params_
print(best_params_gb)

#Fit the model using athe best parameters
cleveland_gradient= GradientBoostingClassifier(**best_params_gb, random_state=42)
cleveland_gradient.fit(x_train_embedded, y_train.values.ravel())

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test_embedded)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
import numpy as np

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       76.80
Precision      75.39
Recall         73.66
F1 Score       73.95
MCC            53.82
Specificity    79.55
dtype: float64

Standard Deviation Metrics:
Accuracy        7.27
Precision       8.01
Recall         13.91
F1 Score        9.50
MCC            15.06
Specificity     7.67
dtype: float64
```

```
#Feature Selection using Embedded Methods

#Gradient Boosting Feature Importance Plot
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.inspection import permutation_importance #Neccessary class for permutation importances

#Initialize the random forest classifier. Assign random_state 42 to reproduce results
gradient_embedded =GradientBoostingClassifier(random_state=42)

#Fit the training data set to the random forest classifier
gradient_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Calculate the feature importance based on the permutation method
gradient_permutation= permutation_importance(gradient_embedded, x_test, y_test, n_repeats=10, random_state=42)

#Extract feature importance plot
(pd.Series(gradient_permutation.importances_mean, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

import matplotlib.pyplot as plt
# Add plot labels
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Gradient Boosting - Mean Permutation Features Importance')
plt.show()

#Based on feature importance plot, all features below ca_1 appear to be most relevant

#Drop least releavnt features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['thal_2','restecg_1','cp_1','age','chol','fbs','ca_3','restecg_2','cp_2','slope_1']

#Remove selected features from x_train and x_test
x_train_perm = x_train.drop(columns=features_remove)
x_test_perm = x_test.drop(columns=features_remove)
```



```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['trestbps', 'thalach', 'oldpeak']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])


#Gradient Boosting

#Import gradient boosting classifier from scikit-learn libraries
from sklearn.ensemble import GradientBoostingClassifier

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 42 to reproduce result
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the support vector machine classifier
cleveland_gradient.fit(x_train_perm, y_train.values.ravel()).predict(x_test_perm)

#Predict the presence of heart disese by inputting the test data into the cleveland_gradient
target_pred_gradient = cleveland_gradient.predict(x_test_perm)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

accuracy_gradient =  round(accuracy_score(y_test, target_pred_gradient),4)*100
precision_gradient = round(precision_score(y_test, target_pred_gradient, pos_label=1),3)
recall_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=1),3)
f1_score_gradient = round(f1_score(y_test, target_pred_gradient, pos_label=1),3)
mcc_gradient = round(matthews_corrcoef(y_test, target_pred_gradient),3)
specificity_gradient = round(recall_score(y_test, target_pred_gradient, pos_label=0),3)

#Organize performance metrics into a list
performance_gradient = [["Gradient Boosting", accuracy_gradient,specificity_gradient,precision_gradient,recall_gradient,f1_score_gradient,mc

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_gradient, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_gradient = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	Gradient Boosting	77.87	0.883	0.857	0.677	0.757	0.572	0.256542	273.79

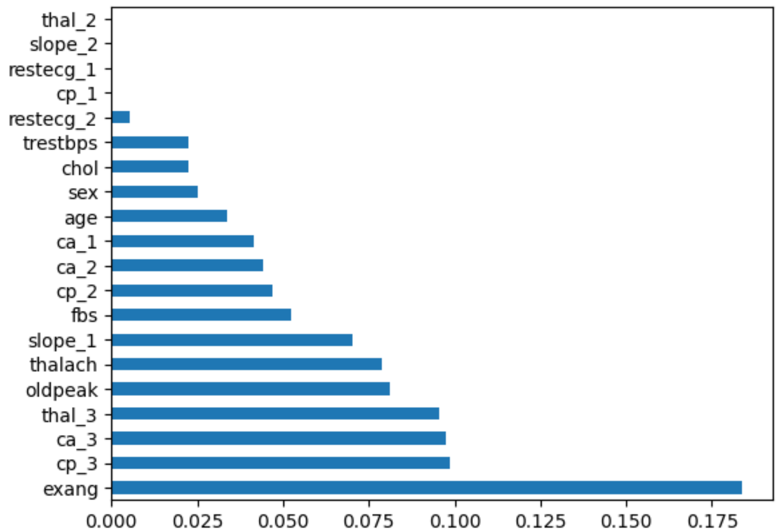
```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
import numpy as np

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_gradient = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth=1, random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

```
#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1),2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1),2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1),2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient),2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0),2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       76.57
Precision       0.77
Recall          0.71
F1 Score        0.73
MCC             0.53
Specificity     0.82
dtype: float64

Standard Deviation Metrics:
Accuracy       7.31
Precision      0.10
Recall         0.12
F1 Score       0.09
MCC            0.15
Specificity    0.10
dtype: float64
```

```
#Feature Selection using Embedded Methods

#XGBoost Feature Importance Plot
import xgboost as xgb

#Initialize the random forest classifier. Assign random_state 42 to reproduce results
xgboost_embedded =xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica
```

```
#Fit the training data set to the random forest classifier
xgboost_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Extract feature importance plot
(pd.Series(xgboost_embedded.feature_importances_, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

#Based on feature importance plot, all features below slope_1 appear to be most relevant

#Drop least releavnt features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['thal_2','slope_2','restecg_1','cp_1','restecg_2','trestbps','chol','sex','age']

#Remove selected features from x_train and x_test
x_train_embedded = x_train.drop(columns=features_remove)
x_test_embedded = x_test.drop(columns=features_remove)
```




```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['thalach', 'oldpeak']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])
```

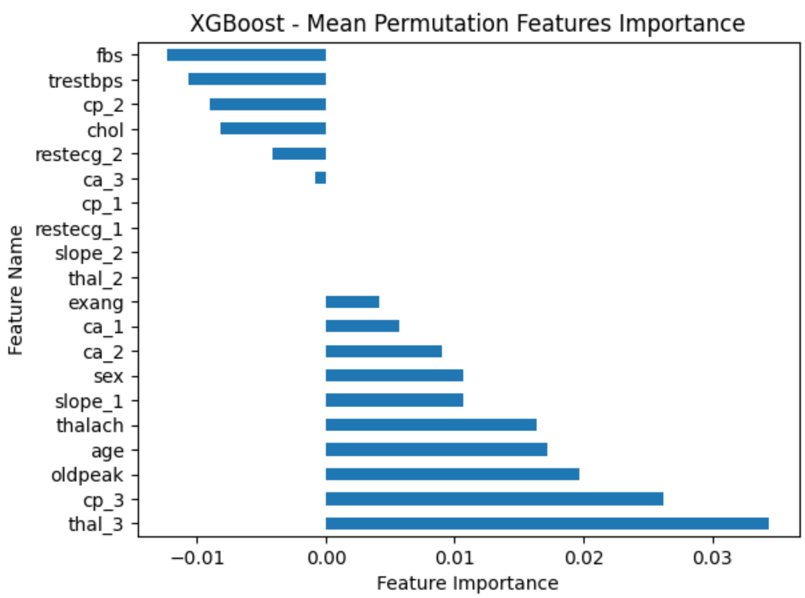
```
#XGBoost

#Import xgboost
import xgboost as xgb

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

#Fit the training data set to the support vector machine classifier
cleveland_xgb.fit(x_train_embedded, y_train.values.ravel()).predict(x_test_embedded)

#Predict the presence of heart disese by inputting the test data into the cleveland_xgb
target_pred_xgb = cleveland_xgb.predict(x_test_embedded)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb,specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb= performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
XGBoost	73.77	0.8	0.778	0.677	0.724	0.481

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
import xgboost as xgb
import numpy as np

#Initialize the classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       80.76
Precision      80.82
Recall         76.76
F1 Score       78.31
MCC            61.69
Specificity    84.22
dtype: float64

Standard Deviation Metrics:
Accuracy        7.40
Precision       9.47
Recall         11.65
F1 Score        9.14
MCC            15.03
Specificity     8.70
dtype: float64
```

```
#Feature Selection using Embedded Methods

#XGBoost Feature Importance Plot
import xgboost as xgb
from sklearn.inspection import permutation_importance

#Initialize the random forest classifier. Assign random_state 42 to reproduce results
xgboost_embedded =xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

#Fit the training data set to the random forest classifier
xgboost_embedded.fit(x_train, y_train.values.ravel()).predict(x_test)

#Calculate the feature importance based on the permutation method
xgboost_permutation= permutation_importance(xgboost_embedded, x_test, y_test, n_repeats=10, random_state=42)

#Extract feature importance plot
(pd.Series(xgboost_permutation.importances_mean, index=x_train.columns)
   .nlargest(20)
   .plot(kind='barh'))

import matplotlib.pyplot as plt
# Add plot labels
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('XGBoost - Mean Permutation Features Importance')
plt.show()

#Based on feature importance plot, all features below ca_1 appear to be most relevant

#Drop least relevant features from x_train and x_test

#Features to drop based on importance chart
features_remove = ['restecg_2','ca_3', 'cp_1','thal_2' 'slope_2','restecg_1','cp_1','exang','ca_1']

#Remove selected features from x_train and x_test
x_train_perm = x_train.drop(columns=features_remove)
x_test_perm = x_test.drop(columns=features_remove)
```



```
#Remove features from consistent dataset

#Modify consistent dataset for k-fold cross-validation

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'thalach']

#Separate target column
x_simple = data_cleveland_coded.drop(columns=['target'] + features_remove)
y_simple = data_cleveland_coded['target']

#Initialize RobustScaler
scaler = RobustScaler()
#Fit the consistent dataset to the RobustScaler instance.
x_simple[cleveland_numerical] = scaler.fit_transform(x_simple[cleveland_numerical])


#XGBoost

#Import xgboost
import xgboost as xgb

#Initialize the support vector machine classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()


#Fit the training data set to the support vector machine classifier
cleveland_xgb.fit(x_train_perm, y_train.values.ravel()).predict(x_test_perm)

#Predict the presence of heart disese by inputting the test data into the cleveland_xgb
target_pred_xgb = cleveland_xgb.predict(x_test_perm)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

accuracy_xgb =  round(accuracy_score(y_test, target_pred_xgb),4)*100
precision_xgb = round(precision_score(y_test, target_pred_xgb, pos_label=1),3)
recall_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=1),3)
f1_score_xgb = round(f1_score(y_test, target_pred_xgb, pos_label=1),3)
mcc_xgb = round(matthews_corrcoef(y_test, target_pred_xgb),3)
specificity_xgb = round(recall_score(y_test, target_pred_xgb, pos_label=0),3)

#Organize performance metrics into a list
performance_xgb = [["XGBoost", accuracy_xgb,specificity_xgb,precision_xgb,recall_xgb,f1_score_xgb,mcc_xgb,computational_time,memory_usage]]

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_xgb, columns=['Model', 'Accuracy','Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC','Com
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_xgb= performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
XGBoost	72.95	0.717	0.73	0.742	0.736	0.459	0.143458	273.79

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
import xgboost as xgb
import numpy as np

#Initialize the classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_xgb = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic', tree_method='hist', eta=0.3, max_depth=3, enable_categorica

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_simple, y_simple.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_simple.iloc[train_index], x_simple.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_simple.iloc[train_index], y_simple.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_gradient.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_gradient= cleveland_gradient.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_gradient)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_gradient, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_gradient)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_gradient, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       76.57
Precision      77.39
Recall         70.82
F1 Score       73.22
MCC            53.46
Specificity    81.56
dtype: float64

Standard Deviation Metrics:
Accuracy        7.31
Precision      10.25
Recall         12.19
F1 Score        8.83
MCC            14.83
Specificity     9.90
dtype: float64
```

Wrapper Methods of Feature Selection: Backwards Elimination

```
pip install mlxtend
```

```
Requirement already satisfied: mlxtend in /usr/local/lib/python3.10/dist-packages (0.22.0)
Requirement already satisfied: scipy>=1.2.1 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.11.3)
Requirement already satisfied: numpy>=1.16.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.23.5)
Requirement already satisfied: pandas>=0.24.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.5.3)
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.2.2)
Requirement already satisfied: matplotlib>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (3.7.1)
Requirement already satisfied: joblib>=0.13.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.3.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from mlxtend) (67.7.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (4.44.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (23.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.24.2->mlxtend) (2023.3.post1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.2->mlxtend) (3.2.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib>=3.0.0->mlxten
```

Backward Elimination: Decision Tree Classifier

```
#Import sequential feature selector class from mlxtend.feature_selection library
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

#Instantiate decision tree classifier and set random state to ensure reproducibility of the results
decision_backward =DecisionTreeClassifier(random_state = 42)

#Instantiate backward elimination feature selection
backward_elim = SFS(estimator=decision_backward,
          k_features=(1, 20), #Indicate that any number of features between 0 and 20 may be chosen to maximize the score
          forward=False,
          floating=False,
          scoring='accuracy', #Metric to maximized
          cv=5) #5-fold cross-validation to be used during feature selection

#Fit backward elimination feature selection using the training data
backward_elim = backward_elim.fit(x_train, y_train)

#Access average accuracy scores and feature names
results = pd.DataFrame.from_dict(backward_elim.get_metric_dict()).T
print(results[['avg_score', 'feature_names']])

#Identify the maximum average score calculate and print the corresponding feature indices
max_features = results[results['avg_score'] == results['avg_score'].max()]
print(max_features['feature_idx'].values[0])
print(max_features['feature_names'].values[0])
```

```
   avg_score                                     feature_names
20  0.701652  (age, sex, trestbps, chol, fbs, thalach, exang...
19  0.756907  (age, sex, trestbps, chol, fbs, exang, oldpeak...
18  0.795796  (age, sex, trestbps, chol, fbs, exang, oldpeak...
17   0.79009  (age, sex, chol, fbs, exang, oldpeak, cp_1, cp...
16  0.834234  (age, sex, chol, fbs, exang, oldpeak, cp_1, cp...
15  0.834084  (age, sex, chol, fbs, exang, oldpeak, cp_1, cp...
14   0.83964  (age, sex, fbs, exang, oldpeak, cp_1, cp_2, cp...
13   0.83964  (age, sex, fbs, exang, oldpeak, cp_1, cp_2, cp...
12  0.828829  (age, sex, fbs, exang, oldpeak, cp_1, cp_2, cp...
11  0.823423  (age, sex, fbs, exang, oldpeak, cp_2, cp_3, sl...
10  0.828829  (age, fbs, exang, oldpeak, cp_2, cp_3, slope_2...
9   0.856306  (age, fbs, exang, oldpeak, cp_3, slope_2, ca_3...
8   0.828829  (age, fbs, oldpeak, cp_3, slope_2, ca_3, thal_...
7   0.828829  (age, fbs, oldpeak, cp_3, slope_2, thal_2, tha...
6   0.806607         (age, fbs, oldpeak, cp_3, slope_2, thal_3)
5   0.795646              (fbs, oldpeak, cp_3, slope_2, thal_3)
4    0.79009                       (fbs, cp_3, slope_2, thal_3)
3   0.778829                                (fbs, cp_3, thal_3)
2   0.756907                                      (fbs, thal_3)
1   0.756907                                          (thal_3,)
(0, 4, 6, 7, 10, 14, 17, 18, 19)
('age', 'fbs', 'exang', 'oldpeak', 'cp_3', 'slope_2', 'ca_3', 'thal_2', 'thal_3')
```

```
#Select columns from x_train and x_test identified by backward elimination
selected_features = list(max_features['feature_idx'].values[0])
x_train_backward = x_train.iloc[:,selected_features]
x_test_backward = x_test.iloc[:,selected_features]


#Modify consistent dataset for k-fold cross-validation

#Features to drop based on simple filters
features_select= list(max_features['feature_names'].values[0])

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age','oldpeak']

#Initialize RobustScaler
scaler = RobustScaler()

#Separate target column
x_backward = data_cleveland_coded.drop(columns=['target'])
y_backward = data_cleveland_coded['target']

#Choose variables
x_backward = x_backward[x_backward.columns.intersection(features_select)]

#Fit the consistent dataset to the RobustScaler instance.
x_backward[cleveland_numerical] = scaler.fit_transform(x_backward[cleveland_numerical])

x_backward.head()
```

	age	fbs	exang	oldpeak	cp_3	slope_2	ca_3	thal_2	thal_3
0	0.538462	1	0	0.9375	0	1	0	1	0
1	0.846154	0	1	0.4375	1	0	1	0	0
2	0.846154	0	1	1.1250	1	0	0	0	1
3	-1.461538	0	0	1.6875	0	1	0	0	0
4	-1.153846	0	0	0.3750	0	0	0	0	0

```
#Decision Tree

from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the decision tree classifier and assign to variable cleveland_decision_tree. Assign random_state 0 to reproduce results
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_decision_tree.fit(x_train_backward, y_train.values.ravel()).predict(x_test_backward)

#Predict the presence of heart disese by inputting the test data into the cleveland_decision_tree model
target_pred_decision_tree = cleveland_decision_tree.predict(x_test_backward)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1),3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1),3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1),3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree),3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0),3)

#Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree,precision_decision_tree,recall_decision_tre

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_decision_tree = performance_metrics

performance_metrics
```

	Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
	Decision Tree	66.39	0.75	0.706	0.581	0.637	0.335	0.033554	273.79

```
#Import GridSearchCV class from sklearn library for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

#Initialize the decision tree classifier
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

#A parameter grid was created using the defaults and selected integers
param_grid = {'criterion': ['gini', 'entropy'],
              'max_depth': [None, 1,2,10, 20, 30],
              'min_samples_split': [2, 5, 10,15,18],
              'min_samples_leaf': [1, 2, 4]}

#Initialize the GridSearchCV class using the decision model, the parameter grid and a 10-fold cross-validation
grid_cleveland_dt = GridSearchCV(cleveland_decision_tree , param_grid, cv=10)
grid_cleveland_dt.fit(x_train_backward, y_train)

#Output the best parameters, the model is optimized based on accuracy score
best_params_dt = grid_cleveland_dt.best_params_
print(best_params_dt)

#Fit the model using athe best parameters
cleveland_decision_tree = DecisionTreeClassifier(**best_params_dt, random_state=42)
cleveland_decision_tree.fit(x_train_backward, y_train)

# Use the best model for predictions and recalculate metrics
target_pred_decision_tree = cleveland_decision_tree.predict(x_test_backward)

#Calculate Performance Metrics
accuracy_decision_tree =  round(accuracy_score(y_test, target_pred_decision_tree),4)*100
precision_decision_tree = round(precision_score(y_test, target_pred_decision_tree, pos_label=1), 3)
recall_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=1), 3)
f1_score_decision_tree = round(f1_score(y_test, target_pred_decision_tree, pos_label=1), 3)
mcc_decision_tree = round(matthews_corrcoef(y_test, target_pred_decision_tree), 3)
specificity_decision_tree = round(recall_score(y_test, target_pred_decision_tree, pos_label=0), 3)

# Organize performance metrics into a list
performance_decision_tree = [["Decision Tree", accuracy_decision_tree, specificity_decision_tree, precision_decision_tree, recall_decision_t

# Create a DataFrame of performance metrics
grid_dt_pm = pd.DataFrame(performance_decision_tree, columns=['Model', 'Accuracy', 'Specificity', 'Precision', 'Recall', 'F1 Score', 'MCC'])
grid_dt_pm.index = [""]

grid_dt_pm

#There are improvements to the metrics with the exception of specificity
```

```
{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC
Decision Tree	66.39	0.75	0.706	0.581	0.637	0.335

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_decision_tree = DecisionTreeClassifier(random_state=42)

# Initialize RepeatedStratifiedKFold (will complete 30 rounds in total)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_backward, y_backward.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_backward.iloc[train_index], x_backward.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_backward.iloc[train_index], y_backward.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_decision_tree.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_decision_tree = cleveland_decision_tree.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_decision_tree)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_decision_tree, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_decision_tree)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_decision_tree, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       70.42
Precision      69.54
Recall         65.92
F1 Score       67.12
MCC            40.90
Specificity    74.24
dtype: float64

Standard Deviation Metrics:
Accuracy        8.42
Precision      12.54
Recall         10.60
F1 Score        9.62
MCC            17.26
```

```
Specificity    12.61
dtype: float64
```

Backward Elimination: Random Forest

```
#Import sequential feature selector class from mlxtend.feature_selection library
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

#Instantiate classifier and set random state to ensure reproducibility of the results
random_backward =RandomForestClassifier(random_state=42)

#Instantiate backward elimination feature selection
backward_elim = SFS(estimator=random_backward,
           k_features=(1, 20), #Indicate that any number of features between 0 and 20 may be chosen to maximize the score
           forward=False,
           floating=False,
           scoring='accuracy', #Metric to maximized
           cv=5) #5-fold cross-validation to be used during feature selection

#Fit backward elimination feature selection using the training data
backward_elim = backward_elim.fit(x_train, y_train.values.ravel())

#Access average accuracy scores and feature names
results = pd.DataFrame.from_dict(backward_elim.get_metric_dict()).T
print(results[['avg_score', 'feature_names']])

#Identify the maximum average score calculate and print the corresponding feature indices
max_features = results[results['avg_score'] == results['avg_score'].max()]
print(max_features['feature_idx'].values[0])
print(max_features['feature_names'].values[0])
```

```
    avg_score                                   feature_names
20  0.834084  (age, sex, trestbps, chol, fbs, thalach, exang...
19  0.845345  (age, sex, trestbps, chol, fbs, thalach, exang...
18  0.850751  (age, sex, trestbps, chol, fbs, thalach, exang...
17   0.83979  (age, sex, trestbps, chol, fbs, thalach, exang...
16   0.83994  (age, sex, trestbps, chol, fbs, thalach, exang...
15   0.83979  (age, trestbps, chol, fbs, thalach, exang, old...
14   0.83964  (age, trestbps, chol, fbs, thalach, exang, old...
13   0.83994  (age, trestbps, chol, fbs, exang, oldpeak, cp_...
12  0.845345  (age, trestbps, chol, fbs, exang, oldpeak, cp_...
11   0.83979  (age, trestbps, chol, fbs, exang, oldpeak, cp_...
10  0.845495  (age, trestbps, chol, fbs, exang, oldpeak, cp_...
9    0.83994  (age, trestbps, chol, exang, oldpeak, cp_1, cp...
8   0.850901  (age, trestbps, chol, exang, oldpeak, cp_2, sl...
7   0.834535   (age, chol, exang, oldpeak, cp_2, slope_1, ca_3)
6    0.83964            (age, chol, exang, oldpeak, cp_2, ca_3)
5    0.83994                  (age, chol, exang, oldpeak, cp_2)
4   0.834535                        (age, chol, exang, oldpeak)
3    0.79024                              (age, exang, oldpeak)
2   0.740691                                       (age, exang)
1   0.756757                                          (exang,)
(0, 2, 3, 6, 7, 9, 13, 17)
('age', 'trestbps', 'chol', 'exang', 'oldpeak', 'cp_2', 'slope_1', 'ca_3')
```

```
#Select columns from x_train and x_test identified by backward elimination
selected_features = list(max_features['feature_idx'].values[0])
x_train_backward = x_train.iloc[:,selected_features]
x_test_backward = x_test.iloc[:,selected_features]
```

```
#Modify consistent dataset for k-fold cross-validation

#Features to drop based on simple filters
features_select= list(max_features['feature_names'].values[0])

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age', 'trestbps', 'chol',  'oldpeak']

#Initialize RobustScaler
scaler = RobustScaler()

#Separate target column
x_backward = data_cleveland_coded.drop(columns=['target'])
y_backward = data_cleveland_coded['target']

#Choose variables
x_backward = x_backward[x_backward.columns.intersection(features_select)]

#Fit the consistent dataset to the RobustScaler instance.
x_backward[cleveland_numerical] = scaler.fit_transform(x_backward[cleveland_numerical])

x_backward.head()
```

	age	trestbps	chol	exang	oldpeak	cp_2	slope_1	ca_3
0	0.538462	0.75	-0.125000	0	0.9375	0	0	0
1	0.846154	1.50	0.703125	1	0.4375	0	1	1
2	0.846154	-0.50	-0.187500	1	1.1250	0	1	0
3	-1.461538	0.00	0.140625	0	1.6875	1	0	0
4	-1.153846	0.00	-0.578125	0	0.3750	0	0	0

```
#Repeat classifier and calculate performance metrics using only selected features

#Random Forest

from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_random_forest = RandomForestClassifier(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_random_forest.fit(x_train_backward, y_train.values.ravel()).predict(x_test_backward)

#Predict the presence of heart disese by inputting the test data into the cleveland_random_forest model
target_pred_random_forest = cleveland_random_forest.predict(x_test_backward)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_random_forest =  round(accuracy_score(y_test, target_pred_random_forest),4)*100
precision_random_forest = round(precision_score(y_test, target_pred_random_forest, pos_label=1),3)
recall_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=1),3)
f1_score_random_forest = round(f1_score(y_test, target_pred_random_forest, pos_label=1),3)
mcc_random_forest = round(matthews_corrcoef(y_test, target_pred_random_forest),3)
specificity_random_forest = round(recall_score(y_test, target_pred_random_forest, pos_label=0),3)

#Organize performance metrics into a list
performance_random_forest = [["Random Forest", accuracy_random_forest, specificity_random_forest,precision_random_forest,recall_random_fores

#Create dataframe of performance metrics
performance_metrics= pd.DataFrame(performance_random_forest, columns=['Model', 'Accuracy', 'Specificity','Precision', 'Recall', 'F1 Score',
performance_metrics.index = [""]

#Create copy to append to a summary table
st1_pm_random_forest = performance_metrics

performance_metrics
```

Model	Accuracy	Specificity	Precision	Recall	F1 Score	MCC	Computational Speed	Memory Usage
Random Forest	68.03	0.733	0.709	0.629	0.667	0.364	0.304591	260.94

```
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Initialize the classifier and assign it to the variable. Assign random_state 42 to ensure reproducibility of results
cleveland_random_forest = RandomForestClassifier(random_state=42)

# Initialize RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Create empty lists to store performance metrics
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
mcc_list = []
specificity_list = []

#We will begin by iterating through the folds created by the Stratified K-fold
for train_index, test_index in cv.split(x_backward, y_backward.values.ravel()):
    #Start by grouping training and testing features based on training and testing row indices within the fold
    x_train_fold, x_test_fold = x_backward.iloc[train_index], x_backward.iloc[test_index]
    #Group labels based on training and testing row indices within the fold
    y_train_fold, y_test_fold = y_backward.iloc[train_index], y_backward.iloc[test_index]

    #Fit the training data set to the classifier
    cleveland_random_forest.fit(x_train_fold, y_train_fold.values.ravel())

    #Predict the presence of heart disease by inputting the test data
    target_pred_random_forest= cleveland_random_forest.predict(x_test_fold)

    #Calculate performance metrics for the current fold
    accuracy_list.append(round(accuracy_score(y_test_fold, target_pred_random_forest)*100,2))
    precision_list.append(round(precision_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    recall_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    f1_list.append(round(f1_score(y_test_fold, target_pred_random_forest, pos_label=1)*100,2))
    mcc_list.append(round(matthews_corrcoef(y_test_fold, target_pred_random_forest)*100,2))
    specificity_list.append(round(recall_score(y_test_fold, target_pred_random_forest, pos_label=0)*100,2))

#Create a DataFrame to store metrics for each fold and repeat
performance_metrics_fold = pd.DataFrame({
    'Accuracy': accuracy_list,
    'Precision': precision_list,
    'Recall': recall_list,
    'F1 Score': f1_list,
    'MCC': mcc_list,
    'Specificity': specificity_list
})

#Calculate mean and standard deviation across folds and repeats
mean_metrics = round(performance_metrics_fold.mean(),2)
std_metrics = round(performance_metrics_fold.std(),2)

#Display mean and standard deviation
print('Mean Metrics:')
print(mean_metrics)
print('\nStandard Deviation Metrics:')
print(std_metrics)

#performance_metrics_fold
```

```
Mean Metrics:
Accuracy       73.37
Precision      72.72
Recall         68.33
F1 Score       69.95
MCC            46.79
Specificity    77.71
dtype: float64

Standard Deviation Metrics:
Accuracy        7.90
Precision      10.06
Recall         11.97
F1 Score        9.18
MCC            16.13
Specificity     9.66
dtype: float64
```

Backward Elimination: Logistic Regression

```
#Import sequential feature selector class from mlxtend.feature_selection library
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

#Instantiate classifier and set random state to ensure reproducibility of the results
logistic_backward =LogisticRegression(random_state=42)

#Instantiate backward elimination feature selection
backward_elim = SFS(estimator=logistic_backward,
           k_features=(1, 20), #Indicate that any number of features between 0 and 20 may be chosen to maximize the score
           forward=False,
           floating=False,
           scoring='accuracy', #Metric to maximized
           cv=5) #5-fold cross-validation to be used during feature selection

#Fit backward elimination feature selection using the training data
backward_elim = backward_elim.fit(x_train, y_train.values.ravel())

#Access average accuracy scores and feature names
results = pd.DataFrame.from_dict(backward_elim.get_metric_dict()).T
print(results[['avg_score', 'feature_names']])

#Identify the maximum average score calculate and print the corresponding feature indices
max_features = results[results['avg_score'] == results['avg_score'].max()]
print(max_features['feature_idx'].values[0])
print(max_features['feature_names'].values[0])
```

```
   avg_score                                      feature_names
20  0.856456  (age, sex, trestbps, chol, fbs, thalach, exang...
19  0.867417  (age, sex, trestbps, chol, fbs, exang, oldpeak...
18  0.878679  (age, sex, trestbps, chol, fbs, exang, oldpeak...
17  0.878679  (age, sex, trestbps, chol, fbs, exang, oldpeak...
16  0.878679  (age, sex, chol, fbs, exang, oldpeak, cp_1, cp...
15  0.884234  (age, sex, chol, exang, oldpeak, cp_1, cp_2, c...
14  0.878679  (age, sex, chol, exang, oldpeak, cp_1, cp_2, c...
13  0.878679  (age, sex, chol, exang, oldpeak, cp_1, cp_2, c...
12  0.884084  (age, sex, exang, oldpeak, cp_1, cp_2, cp_3, r...
11  0.872973  (age, sex, exang, oldpeak, cp_1, cp_2, cp_3, s...
10  0.872973  (age, sex, exang, oldpeak, cp_2, cp_3, slope_1...
9   0.867568  (sex, exang, oldpeak, cp_2, cp_3, slope_1, ca_...
8   0.862012  (sex, exang, oldpeak, cp_2, slope_1, ca_1, ca_...
7   0.867568  (exang, oldpeak, cp_2, slope_1, ca_1, ca_2, th...
6   0.867718      (exang, oldpeak, cp_2, slope_1, ca_2, thal_3)
5   0.856456            (exang, oldpeak, slope_1, ca_2, thal_3)
4   0.850901                  (exang, oldpeak, slope_1, thal_3)
3   0.828979                         (oldpeak, slope_1, thal_3)
2   0.795495                                 (oldpeak, slope_1)
1   0.728829                                         (slope_1,)
(0, 1, 3, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19)
('age', 'sex', 'chol', 'exang', 'oldpeak', 'cp_1', 'cp_2', 'cp_3', 'restecg_2', 'slope_1', 'slope_2', 'ca_1', 'ca_2', 'thal_2', 'thal_3'
```

```
#Select columns from x_train and x_test identified by backward elimination
selected_features = list(max_features['feature_idx'].values[0])
x_train_backward = x_train.iloc[:,selected_features]
x_test_backward = x_test.iloc[:,selected_features]
```

```
#Modify consistent dataset for k-fold cross-validation

#Features to drop based on simple filters
features_select= list(max_features['feature_names'].values[0])

#Data will be scaled using the robust scaler that is less susceptible to outliers
from sklearn.preprocessing import RobustScaler

#Subset of numerical features
cleveland_numerical =['age',  'chol',  'oldpeak']

#Initialize RobustScaler
scaler = RobustScaler()

#Separate target column
x_backward = data_cleveland_coded.drop(columns=['target'])
y_backward = data_cleveland_coded['target']

#Choose variables
```

```
#Repeat classifier and calculate performance metrics using only selected features

#Logistic Regression

from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

#Import classes to calculate memory consumption and runtime
import timeit
import psutil

#Initialize the classifier and assign to variable. Assign random_state 0 to reproduce results
cleveland_logistic_reg = LogisticRegression(random_state=42)

#Begin timing of fitting and prediction process
start_time = timeit.default_timer()

#Fit the training data set to the decision tree model
cleveland_logistic_reg.fit(x_train_backward, y_train.values.ravel()).predict(x_test_backward)

#Predict the presence of heart disese by inputting the test data into the cleveland_logistic_reg model
target_pred_logistic_reg = cleveland_logistic_reg.predict(x_test_backward)

#Stop timing of fitting and prediction process
end_time = timeit.default_timer()

#Calculate total time
computational_time = end_time - start_time
#Calculate memory usage in megabytes
memory_usage = round(psutil.Process().memory_info().rss/ (1024 * 1024),2)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef

accuracy_logistic_reg =  round(accuracy_score(y_test, target_pred_logistic_reg),4)*100
precision_logistic_reg = round(precision_score(y_test, target_pred_logistic_reg, pos_label=1),3)
recall_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=1),3)
f1_score_logistic_reg = round(f1_score(y_test, target_pred_logistic_reg, pos_label=1),3)
mcc_logistic_reg = round(matthews_corrcoef(y_test, target_pred_logistic_reg),3)
specificity_logistic_reg = round(recall_score(y_test, target_pred_logistic_reg, pos_label=0),3)
```