REFINING HEART DISEASE PREDICTION ACCURACY USING HYBRID MACHINE LEARNING TECHNIQUES WITH NOVEL METAHEURISTIC ALGORITHMS

Report submitted to the SASTRA Deemed to be University
as the requirement for the course

CSE300 - MINI PROJECT

Submitted by

SANJAY M
(Reg no.: 126003230, B.Tech - COMPUTER SCIENCE & ENGINEERING)
SHREERAAM J
(Reg no.: 126003247, B.Tech - COMPUTER SCIENCE & ENGINEERING)
SIVAHARI D
(Reg no.: 126003252, B.Tech - COMPUTER SCIENCE & ENGINEERING)

May 2025

SCHOOL OF COMPUTING

THANJAVUR, TAMIL NADU, INDIA – 613 401

SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

THANJAVUR | KUMBAKONAM | CHENNAI

SCHOOL OF COMPUTING

THANJAVUR – 613 401

Bonafide Certificate

This is to certify that the report titled "**Refining Heart Disease Prediction Accuracy Using Hybrid Machine Learning Techniques With Novel Metaheuristic Algorithms**" submitted as a requirement for the course, CSE300: **MINI PROJECT** for B.Tech. is a bonafide record of the work done by **Mr. SANJAY M (Reg no.: 126003230, B.Tech COMPUTER SCIENCE & ENGINEERING), Mr. SHREERAAM J (Reg no.: 126003247, B.Tech COMPUTER SCIENCE & ENGINEERING) & Mr. SIVAHARI D (Reg no.: 126003252, B.Tech COMPUTER SCIENCE & ENGINEERING)** during the academic year 2024-25, in the School of Computing, under my supervision.

Signature of Project Supervisor :

Name with Affiliation : **Dr. Venkatesan R**, Asst. Professor - III, SoC

Date :

Mini Project *Viva voc*e held on _____

Examiner 1 **Examiner 2**

Acknowledgements

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan,** Dean**,** Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli,** Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr. V. S. Shankar Sriram**, Dean, School of Computing, **Dr. R. Muthaiah**, Associate Dean, Research, **Dr. K.Ramkumar**, Associate Dean, Academics, **Dr. D. Manivannan**, Associate Dean, Infrastructure, **Dr. R. Alageswaran**, Associate Dean, Students Welfare

Our guide **Dr. Venkatesan R**, Asst. Professor - III, School of Computing was the driving force behind this whole idea from the start. His deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing us an opportunity to showcase our skills through this project.

List of Figures

Abbreviations

DTC	Decision Tree Classifier
FOA	Forest Optimization Algorithm
GAO	Giant Armadillo Optimization
KNNC	K-Nearest Neighbor Classifier
LRC	Logistic Regression Classifier
ML	Machine Learning
PFA	Pathfinder Algorithm
RFC	Random Forest Classifier
SMA	Slime Mold Algorithm
XGBC	eXtreme Gradient Boosting Classifier

Abstract

Heart disease is a major global health issue, and early detection is vital. This project improves heart disease prediction by combining machine learning models-like XGBoost, Random Forest, and Decision Tree-with advanced optimization algorithms inspired by nature. The process starts by selecting the most important features from patient data using cross-validation and other methods to ensure the models focus on what matters most. After training five different models, the top three are further optimized to boost their accuracy. The best results come from the hybrid XGBoost + Giant Armadillo Optimization model, which achieves high accuracy and reliability. This approach can help doctors predict heart disease more effectively and support better patient care.

Keywords: Metaheuristic, Hyperparameter, Optimization, Fitness Function

Table of Contents

CHAPTER 1

SUMMARY OF THE BASE PAPER

Title **:** Refining Heart Disease Prediction Accuracy Using Hybrid Machine Learning Techniques with Novel Metaheuristic Algorithms

Authors **:** Haifeng Zhang, Rui Mu

Publisher/Journal **:** International Journal Of Cardiology

Year **:** 2024

Indexing **:** SCIE

Content

The main motive of this paper is to make early detection of heart disease more accurate and reliable, so that doctors can help patients sooner and save more lives. The objective is to build a smarter prediction system by combining machine learning models with advanced, nature-inspired optimization techniques, ensuring the system focuses on the most important patient information and delivers the best possible results.

In this study, researchers used a dataset containing 13 key features about patients, such as age, cholesterol, and blood pressure, to predict the likelihood of heart disease. They began by selecting the most relevant features using cross-validation and other methods, which helped the models focus only on the data that truly matters. Five machine learning models-XGBoost, Random Forest, Decision Tree, K-Nearest Neighbors, and Logistic Regression-were trained and tested. The top three models were then further improved using four metaheuristic optimization algorithms inspired by natural behaviors, like how armadillos or slime molds search for food. The best performance was achieved by combining XGBoost with Giant Armadillo Optimization, resulting in a highly accurate model . This hybrid approach can help doctors predict heart disease more effectively, leading to better and faster patient care.

Key Contributions:

1. **Hybrid Approach:**
    - Integrates feature selection (e.g., PCA, mutual information) with ML models (XGBoost, Random Forest, Decision Tree) and optimizes them using novel metaheuristic algorithms (Giant Armadillo Optimization, Slime Mold Algorithm).
    - This combination is unique and addresses the limitations of standalone ML models.
2. **Novel Optimization Techniques:**
    - Introduces four understudied metaheuristic algorithms (Giant Armadillo, Forest, Pathfinder, Slime Mold) to fine-tune model parameters, improving accuracy and reducing overfitting.
3. **High Accuracy:**
    - The hybrid RFC + Giant Armadillo Optimization outperforms traditional models like Logistic Regression and K-Nearest Neighbors.

Research Problem & Proposed Solution

- **Problem:** Existing ML models for heart disease prediction often lack robustness due to irrelevant features, imbalanced data, and suboptimal parameter tuning.
- **Solution:**
    1. **Feature Selection:** Uses statistical methods (F-test, correlation) and PCA to identify critical features (e.g., cholesterol, exercise-induced angina).
    2. **Hybrid Optimization:** Combines top ML models with metaheuristic algorithms to enhance performance.
    3. **Validation:** Evaluates models using metrics like F1-score, MCC, and cross-validation to ensure reliability.

Architecture & Algorithm

Workflow:

1. **Data Preprocessing:**
    - Uses the UCI heart disease dataset with 13 features (age, cholesterol, blood pressure, etc.).
    - Using feature selection methods like Filter method ,Wrapper method , Embedded methods and Dimensionality reduction using Principle Component analysis

2. **Model Training:**
    - Tests five ML models: XGBoost, Random Forest, Decision Tree, K-Nearest Neighbors, Logistic Regression.
3. **Optimization:**
    - Applies four metaheuristic algorithms to the top three models (XGBoost, Random Forest, Decision Tree).
    - Example: Giant Armadillo Optimization mimics armadillo foraging behavior to adjust model parameters.
4. **Evaluation:**
    - Metrics: Accuracy, precision, recall, F1-score, Matthews Correlation Coefficient (MCC).
    - Results: XGGA achieves **97.2% accuracy** with minimal prediction errors (≤5.5% for alive patients, ≤1.2% for deceased).

Algorithm Correctness:

- Validated through **5-fold cross-validation** and comparison with existing methods
- The hybrid approach reduces computational complexity while maintaining high generalizability.

Key Takeaways

- **Innovation:** First study to combine feature selection, XGBoost, and nature-inspired optimization for heart disease prediction.
- **Impact:** Provides a scalable tool for early diagnosis, enabling timely medical interventions.
- **Limitations:** Dataset size (303 samples) and lack of genetic/socioeconomic features.

CHAPTER 2

MERITS AND DEMERITS OF THE BASE PAPER

MERITS

- **High Prediction Accuracy**:The hybrid model (XGBoost + Giant Armadillo Optimization) achieved excellent accuracy, outperforming many traditional models. This means it can predict heart disease reliably and with minimal error.

- **Smart Use of Data**:The study carefully selected the most important features from patient data using methods like cross-validation and PCA. This helps the models focus on what matters most and reduces unnecessary complexity.

- **Advanced Optimization**:By using metaheuristic algorithms (inspired by nature, like how armadillos or slime molds search for food), the models were further improved. These optimizers help find the best settings for the models, boosting their performance.

- **Comprehensive Evaluation**: The models were tested using multiple metrics (accuracy, precision, recall, F1-score, MCC) and validated with both cross-validation and train-test splits, making the results trustworthy and robust.

- **Real-World Impact**: The approach is practical and can be used in hospitals to help doctors make faster and more accurate decisions about heart disease, potentially saving lives.

DEMERITS

- **Limited Dataset Size:** The dataset used is not very large (about 300 samples). This might limit the model's ability to generalize to new or different populations, especially those not represented in the data.

- **Lack of External Validation:** The model was not tested on an external or real-world dataset outside the original data, which is important to prove its reliability in different settings.
- **Complexity for Deployment:** Combining machine learning with multiple optimization algorithms increases the system's complexity. This can make it harder to implement and maintain in real hospital environments, especially where resources are limited.

- **Interpretability:** While the hybrid model is highly accurate, it may be difficult for doctors to understand exactly how it makes decisions. This "black box" nature can reduce trust and make it harder to explain predictions to patients.

- **Potential Overfitting:** Using advanced optimization and many parameters on a small dataset can sometimes cause the model to "memorize" the training data rather than learn patterns that generalize well (overfitting)

CHAPTER 3

SOURCE CODE

1. DATA PRE-PROCESSING MODULE

1.1. FILTER METHODS

```
import pandas as pd
import numpy as np
from sklearn.feature_selection import mutual_info_classif, f_classif
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Load data and prepare features
data = pd.read_csv('data.csv')
X = data.iloc[:, :-1]
y = data.iloc[:, -1]

# Calculate all metrics at once
metrics = {
   'Mutual Information': mutual_info_classif(X, y),
   'F-test Score': f_classif(X, y)[0],
   'Correlation': data.corrwith(y).abs().drop(y.name).values
}

# Create unified DataFrame
feature_importance = pd.DataFrame({'Feature': X.columns})
for name, values in metrics.items():
   feature_importance[name] = values

# Generic plotting function
def plot_metric(metric, title):
   df = feature_importance.sort_values(metric, ascending=False)
   plt.figure(figsize=(10, 6))
   plt.bar(df['Feature'], df[metric])
   plt.title(f'Feature Importance: {title}')
   plt.xticks(rotation=90)
   plt.tight_layout()
   plt.show()
   return df

# Plot and display results for each metric
for metric, title in [('Mutual Information', 'Mutual Information'),
                      ('F-test Score', 'F-test Scores'),
                      ('Correlation', 'Correlation')]:
```

```
    sorted_df = plot_metric(metric, title)
    print(f"\n{title}:\n", sorted_df[['Feature', metric]])

# Combined comparison plot
scaler = MinMaxScaler()
scaled = pd.DataFrame(scaler.fit_transform(feature_importance.iloc[:, 1:]),
                      columns=[f"{col} (Scaled)" for col in metrics],
                      index=feature_importance.Feature)

ax = scaled.plot.bar(figsize=(15, 6), width=0.8)
plt.title('Scaled Feature Importance Comparison')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

1.2. DIMENSIONALITY REDUCTION

```
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Load your data
data = pd.read_csv('data.csv')

# Standardize features
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data.drop('target',axis=1))

# Apply PCA
pca = PCA(n_components=2)  # Example with 2 components
principal_components = pca.fit_transform(standardized_data)

# Get the loadings (coefficients) for each feature in each component
loadings = pca.components_

# Select features based on loadings
# For example, select features with the highest absolute loading in the first component
feature_importances = np.abs(loadings[0])
top_features = np.argsort(feature_importances)[::-1][:8]  # Select top 5 features

# Use these original features for your model
selected_features = data.iloc[:, top_features]
```

1.3. WRAPPER METHODS

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE
```

```
# Load the dataset
data = pd.read_csv('data.csv')

# Separate features and target
X = data.drop(columns=['target'])
y = data['target']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize a Random Forest Classifier
model = RandomForestClassifier(random_state=42)

# Use Recursive Feature Elimination (RFE) for Wrapper Method
rfe = RFE(estimator=model, n_features_to_select=6)  # Selecting top 6 features
rfe.fit(X_train, y_train)

# Get the selected features
selected_features_wrapper = X.columns[rfe.support_]
print("Selected features using Wrapper Method:", selected_features_wrapper.tolist())
```

1.4. EMBEDDED METHODS

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel

# Initialize a Logistic Regression model
logistic_model = LogisticRegression(random_state=42, max_iter=1000)

# Use SelectFromModel for Embedded Method
sfm = SelectFromModel(estimator=logistic_model)
sfm.fit(X_train, y_train)

# Get the selected features
selected_features_embedded = X.columns[sfm.get_support()]
print("Selected features using Embedded Method:", selected_features_embedded.tolist())

# Plot feature importance (absolute values of coefficients)
importance = abs(sfm.estimator_.coef_[0])  # Get absolute values of coefficients
plt.figure(figsize=(12, 6))
plt.bar(X.columns, importance, color='lightcoral')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance using Embedded Method (Logistic Regression)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

2. MODEL IMPLEMENTATION

2.1. LRC

```
# Load data
data = pd.read_csv("data.csv")

# Prepare data
X = data.iloc[:, :-1]  # Features
y = data.iloc[:, -1]   # Target variable

# Initialize k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Initialize K-Fold with k=5
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize logistic regression model
model = LogisticRegression(solver='liblinear', max_iter=1000, random_state=42)

# Store results
accuracies = []

# Perform 5-fold cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Predict and evaluate
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

    print(f"Fold {fold + 1}:")
    print(f"Model Accuracy: {accuracy:.3f}")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("\n" + "="*50 + "\n")

# Calculate average accuracy across 5 folds
print(f"Average Accuracy Across {kf.get_n_splits()} Folds:
```

```
{sum(accuracies)/len(accuracies):.3f}")
```

2.2. DTC

```
# Initialize model
model = DecisionTreeClassifier(random_state=42)

# Store results
accuracies = []

# Perform k-fold cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train and predict
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Calculate metrics
    fold_accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(fold_accuracy)

    # Print fold results
    print(f"Fold {fold + 1}:")
    print(f"Accuracy: {fold_accuracy:.3f}")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("-" * 50 + "\n")

# Print final summary
print(f"Average Accuracy: {sum(accuracies)/len(accuracies):.3f}")
```

2.3. RFC

```
# Initialize model
model = RandomForestClassifier(n_estimators=500, bootstrap=True,
random_state=42)

# Perform k-fold cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    model.fit(X_train, y_train)
```

```
    # Predict and evaluate
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

    print(f"Fold {fold + 1}:")
    print(f"Model Accuracy: {accuracy:.3f}")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("\n")
# Print final summary
print(f"Average Accuracy: {sum(accuracies)/len(accuracies):.3f}")
```

2.4. KNNC

```
# Initialize model and scaler
model = KNeighborsClassifier(n_neighbors=5)
scaler = StandardScaler()

# Store results
accuracies = []

# Perform k-fold cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Scale features
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Train the model
    model.fit(X_train_scaled, y_train)

    # Predict and evaluate
    y_pred = model.predict(X_test_scaled)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

    print(f"Fold {fold + 1}:")
    print(f"Accuracy: {accuracy:.4f}")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("\n" + "="*50 + "\n")
```

```
# Calculate average accuracy
print(f"Average Accuracy Across {kf.get_n_splits()} Folds:
{sum(accuracies)/len(accuracies):.4f}")
```

2.5. XGBC

```
# Initialize model
model = XGBClassifier(
      objective='binary:logistic',
      max_depth=5,  # Increased depth to capture more complex patterns
      learning_rate=0.05,  # Lower learning rate for better performance
      n_estimators=200,  # Increased number of trees
      subsample=0.8,  # Introduced randomness to prevent overfitting
      colsample_bytree=0.8,
   tree_method='hist'
   )
# Perform k-fold cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
   X_train, X_test = X.iloc[train_index], X.iloc[test_index]
   y_train, y_test = y.iloc[train_index], y.iloc[test_index]

   # Train the model
   model.fit(X_train, y_train)

   # Predict and evaluate
   y_pred = model.predict(X_test)
   accuracy = accuracy_score(y_test, y_pred)
   accuracies.append(accuracy)
   print(f"Fold {fold + 1}:")
   print(f"Model Accuracy: {accuracy:.3f}")
   print("Classification Report:")
   print(classification_report(y_test, y_pred))
   print("\n")
 # Print final summary
print(f"Average Accuracy: {sum(accuracies)/len(accuracies):.3f}")
```

3. HYPER-PARAMETER OPTIMIZATION

3.1. RFC

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold,
cross_val_score
```

```
from sklearn.metrics import accuracy_score, classification_report, roc_curve,
roc_auc_score

import matplotlib.pyplot as plt

# 1. Load data
df = pd.read_csv('heart-disease.csv')

# 2. Prepare features and target (drop columns as in your reference)
X = df.drop(['target', 'restecg', 'sex', 'chol', 'fbs', 'trestbps', 'slope', 'exang'], axis=1)
y = df['target'].astype(int)

# 3. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# 4. RFC parameter search space
rfc_space = {
    'n_estimators': (50, 200),
    'max_depth': (3, 12),
    'min_samples_split': (2, 10),
    'min_samples_leaf': (1, 5),
    'max_features': (0.5, 1.0),  # fraction of features
    'bootstrap': (0, 1),         # 0=False, 1=True
    'criterion': (0, 1),         # 0='gini', 1='entropy'
    'class_weight': (0, 1)       # 0=None, 1='balanced'
}

# 5. Helper: convert vector to RFC params
def convert_params(params):
    return {
        'n_estimators': int(round(params[0])),
        'max_depth': int(round(params[1])),
        'min_samples_split': int(round(params[2])),
        'min_samples_leaf': int(round(params[3])),
        'max_features': params[4],
        'bootstrap': [False, True][int(round(params[5]))],
        'criterion': ['gini', 'entropy'][int(round(params[6]))],
        'class_weight': [None, 'balanced'][int(round(params[7]))],
        'random_state': 42
    }

# 6. Objective function for optimization
```

```
def objective(params):
    clf = RandomForestClassifier(**convert_params(params))
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    return cross_val_score(clf, X_train, y_train, cv=cv, scoring='accuracy').mean()

# 7. Metaheuristic optimizers

def SMA_optimization(search_space, max_iter=20):
    pop_size = 5
    population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                           for _ in range(pop_size)])
    for iteration in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])
        sorted_idx = np.argsort(-fitness)
        population = population[sorted_idx]
        a = np.arctanh(1 - (iteration+1)/max_iter)
        b = 1 - (iteration+1)/max_iter
        weights = 1 + np.random.rand() * np.log10((fitness[sorted_idx] - fitness.min())
/
                 (fitness.max() - fitness.min() + 1e-10) + 1)
        new_population = []
        for i in range(pop_size):
            if i < pop_size//2:
                new_pos = population[i] + a * (np.random.rand() * (population[0] -
population[i]))
            else:
                new_pos = population[i] + b * (np.random.rand() * (population[i] -
population[i-1]))
            new_population.append(np.clip(new_pos, [v[0] for v in
search_space.values()],
                                          [v[1] for v in search_space.values()]))
        population = np.array(new_population)
    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

def FOA_optimization(search_space, max_iter=20):
    num_trees = 5
    trees = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(num_trees)])
    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in trees])
        best_idx = np.argmax(fitness)
```

```
        new_trees = []
        for tree in trees:
            if objective(tree) < np.median(fitness):
                new_trees.append(tree + np.random.normal(0, 0.1,
size=len(search_space)))
            else:
                new_trees.append(tree)
        new_trees.append(trees[best_idx] + np.random.normal(0, 0.05,
size=len(search_space)))
        trees = np.clip(new_trees, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])
    best_idx = np.argmax([objective(ind) for ind in trees])
    return trees[best_idx]

def PFA_optimization(search_space, max_iter=20):
    group_size = 5
    group = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(group_size)])
    leader = group[np.argmax([objective(ind) for ind in group])]
    for _ in range(max_iter):
        new_group = []
        for member in group:
            if np.array_equal(member, leader):
                new_pos = member + np.random.rand() * (leader - member)
            else:
                new_pos = member + 2*np.random.rand()*(leader - member) + \
                          np.random.rand()*(member - leader)
            new_group.append(new_pos)
        group = np.clip(new_group, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])
        current_leader = group[np.argmax([objective(ind) for ind in group])]
        if objective(current_leader) > objective(leader):
            leader = current_leader
    return leader

def GAO_optimization(search_space, max_iter=20):
    pop_size = 5
    population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                           for _ in range(pop_size)])
    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])
        for i in range(pop_size):
```

```
            if np.random.rand() < 0.5:
                population[i] += np.random.normal(0, 0.1, size=len(search_space))
        best_idx = np.argmax(fitness)
        for i in range(pop_size):
            if i != best_idx:
                population[i] += np.random.rand() * (population[best_idx] -
population[i])
        population = np.clip(population, [v[0] for v in search_space.values()],
                             [v[1] for v in search_space.values()])
    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

# 8. Run all optimizations and evaluate
optimizers = {
    'SMA': SMA_optimization,
    'FOA': FOA_optimization,
    'PFA': PFA_optimization,
    'GAO': GAO_optimization
}

results = {}
plt.figure(figsize=(10, 8))
for idx, (name, optimizer) in enumerate(optimizers.items()):
    print(f"Optimizing RFC with {name}...")
    best_params = optimizer(rfc_space)
    rfc = RandomForestClassifier(**convert_params(best_params))
    rfc.fit(X_train, y_train)
    y_pred = rfc.predict(X_test)
    y_proba = rfc.predict_proba(X_test)[:, 1]
    report = classification_report(y_test, y_pred, output_dict=True)
    auc = roc_auc_score(y_test, y_proba)
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    results[name] = {
        'params': convert_params(best_params),
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': report['1']['precision'],
        'recall': report['1']['recall'],
        'f1_score': report['1']['f1-score'],
        'auc': auc
    }
    plt.plot(fpr, tpr, label=f'{name} (AUC={auc:.3f})')

plt.plot([0, 1], [0, 1], 'k--', label='No Skill')
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for RFC (All Optimizers)')
plt.legend()
plt.show()

# 9. Plot results
metrics = ['accuracy', 'precision', 'recall', 'f1_score']
optimizer_names = list(results.keys())
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
axs = axs.flatten()
for i, metric in enumerate(metrics):
    values = [results[name][metric] for name in optimizer_names]
    bars = axs[i].bar(optimizer_names, values, color=['#1f77b4', '#ff7f0e', '#2ca02c',
'#d62728'])
    axs[i].set_title(f'RFC: {metric.capitalize()}')
    axs[i].set_ylim(0.5, 1.0)
    for bar in bars:
        height = bar.get_height()
        axs[i].text(bar.get_x() + bar.get_width()/2., height + 0.01,
                f'{height:.4f}', ha='center', va='bottom')
plt.tight_layout()
plt.suptitle('RFC Optimization Comparison', fontsize=16)
plt.subplots_adjust(top=0.9)
plt.show()

# 10. Print summary table
print("Optimization Results Summary:")
print("-" * 80)
print(f"{'Optimizer':<10} | {'Accuracy':<10} | {'Precision':<10} | {'Recall':<10} |
{'F1-score':<10} | Parameters")
print("-" * 80)
for name in optimizer_names:
    params_str = str(results[name]['params'])
    print(f"{name:<10} | {results[name]['accuracy']:<10.4f} | "
          f"{results[name]['precision']:<10.4f} | {results[name]['recall']:<10.4f} | "
          f"{results[name]['f1_score']:<10.4f} | {params_str}")
```

3.2. XGBC

```
import numpy as np
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold,
cross_val_score
```

```
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load and preprocess data
df = pd.read_csv('heart-disease.csv')

# Drop columns as in the reference
X = df.drop(['target', 'restecg', 'sex', 'chol', 'fbs', 'trestbps', 'slope', 'exang'], axis=1)
y = df['target'].astype(int)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

# Define search space for XGBoost parameters
xgb_space = {
   'max_depth': (3, 10),
   'learning_rate': (0.1, 0.3),
   'n_estimators': (50, 200),
   'alpha': (0, 10)
}

# Objective function to maximize
def objective(params):
   model = XGBClassifier(
      max_depth=int(params[0]),
      learning_rate=params[1],
      n_estimators=int(params[2]),
      alpha=params[3],
      objective='binary:logistic',
      eval_metric='logloss'
   )
   cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
   return cross_val_score(model, X_train, y_train, cv=cv,
scoring='accuracy').mean()

# 1. Slime Mold Algorithm (SMA)
def SMA_optimization(search_space, max_iter=20):
   population_size = 5
   population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                          for _ in range(population_size)])
```

```
    for iteration in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])
        sorted_idx = np.argsort(-fitness)
        population = population[sorted_idx]

        # Update weights
        a = np.arctanh(1 - (iteration+1)/max_iter)
        b = 1 - (iteration+1)/max_iter
        weights = 1 + np.random.rand() * np.log10((fitness[sorted_idx] - fitness.min())
/
                (fitness.max() - fitness.min() + 1e-10) + 1)

        # Update positions
        new_population = []
        for i in range(population_size):
            if i < population_size//2:
                new_pos = population[i] + a * (np.random.rand() *
                        (population[0] - population[i]))
            else:
                new_pos = population[i] + b * (np.random.rand() *
                        (population[i] - population[i-1]))
            new_population.append(np.clip(new_pos, [v[0] for v in
search_space.values()],
                                  [v[1] for v in search_space.values()]))

        population = np.array(new_population)

    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

# 2. Forest Optimization Algorithm (FOA)
def FOA_optimization(search_space, max_iter=20):
    num_trees = 5
    trees = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(num_trees)])

    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in trees])
        best_idx = np.argmax(fitness)

        # Local seeding
        new_trees = []
        for tree in trees:
```

```
            if objective(tree) < np.median(fitness):
                new_trees.append(tree + np.random.normal(0, 0.1,
size=len(search_space)))
            else:
                new_trees.append(tree)

        # Global seeding
        new_trees.append(trees[best_idx] + np.random.normal(0, 0.05,
size=len(search_space)))

        trees = np.clip(new_trees, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])

    best_idx = np.argmax([objective(ind) for ind in trees])
    return trees[best_idx]

# 3. Pathfinder Algorithm (PFA)
def PFA_optimization(search_space, max_iter=20):
    group_size = 5
    group = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(group_size)])
    leader = group[np.argmax([objective(ind) for ind in group])]

    for _ in range(max_iter):
        # Update positions
        new_group = []
        for member in group:
            if np.array_equal(member, leader):
                new_pos = member + np.random.rand() * (leader - member)
            else:
                new_pos = member + 2*np.random.rand()*(leader - member) + \
                          np.random.rand()*(member - leader)
            new_group.append(new_pos)

        group = np.clip(new_group, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])

        # Update leader
        current_leader = group[np.argmax([objective(ind) for ind in group])]
        if objective(current_leader) > objective(leader):
            leader = current_leader

    return leader
```

```
# 4. Giant Armadillo Optimization (GAO)
def GAO_optimization(search_space, max_iter=20):
    population_size = 5
    population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                           for _ in range(population_size)])

    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])

        # Excavation phase
        for i in range(population_size):
            if np.random.rand() < 0.5:
                population[i] += np.random.normal(0, 0.1, size=len(search_space))

        # Foraging phase
        best_idx = np.argmax(fitness)
        for i in range(population_size):
            if i != best_idx:
                population[i] += np.random.rand() * (population[best_idx] -
population[i])

        population = np.clip(population, [v[0] for v in search_space.values()],
                             [v[1] for v in search_space.values()])

    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

# Run all four optimizations for XGBoost
optimizers = {
    'SMA': SMA_optimization,
    'FOA': FOA_optimization,
    'PFA': PFA_optimization,
    'GAO': GAO_optimization
}

results = {}

plt.figure(figsize=(8, 6))  # For ROC curves

for name, optimizer in optimizers.items():
    print(f"Optimizing XGBoost with {name}...")
    best_params = optimizer(xgb_space)
```

```
# Create model with optimized parameters
model = XGBClassifier(
    max_depth=int(best_params[0]),
    learning_rate=best_params[1],
    n_estimators=int(best_params[2]),
    alpha=best_params[3],
    objective='binary:logistic',
    eval_metric='logloss'
)

# Train and predict
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]  # <--- ADD THIS LINE

# Calculate metrics
report = classification_report(y_test, y_pred, output_dict=True)
accuracy = accuracy_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_proba)        # <--- ADD THIS LINE
fpr, tpr, _ = roc_curve(y_test, y_proba)    # <--- ADD THIS LINE

# Store results
results[name] = {
    'params': {
        'max_depth': int(best_params[0]),
        'learning_rate': best_params[1],
        'n_estimators': int(best_params[2]),
        'alpha': best_params[3]
    },
    'accuracy': accuracy,
    'precision': report['1']['precision'],
    'recall': report['1']['recall'],
    'f1_score': report['1']['f1-score'],
    'auc': auc
}

print(f"  Accuracy: {accuracy:.4f}")
print(f"  Precision: {report['1']['precision']:.4f}")
print(f"  Recall: {report['1']['recall']:.4f}")
print(f"  F1-score: {report['1']['f1-score']:.4f}")
print(f"  AUC: {auc:.4f}")               # <--- ADD THIS LINE
print(f"  Parameters: {results[name]['params']}")
print("")
```

```
    # --- Plot ROC curve for this optimizer ---
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC={auc:.3f})')

# --- Finalize ROC plot ---
plt.plot([0, 1], [0, 1], 'k--', label='Random guess')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for XGBoost (All Optimizers)')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# Plot results
metrics = ['accuracy', 'precision', 'recall', 'f1_score']
optimizer_names = list(results.keys())

fig, axs = plt.subplots(2, 2, figsize=(12, 8))
axs = axs.flatten()

for i, metric in enumerate(metrics):
    values = [results[name][metric] for name in optimizer_names]
    bars = axs[i].bar(optimizer_names, values, color=['#1f77b4', '#ff7f0e', '#2ca02c',
'#d62728'])
    axs[i].set_title(f'XGBoost: {metric.capitalize()}')
    axs[i].set_ylim(0.5, 1.0)

    # Add value labels
    for bar in bars:
        height = bar.get_height()
        axs[i].text(bar.get_x() + bar.get_width()/2., height + 0.01,
                f'{height:.4f}', ha='center', va='bottom')

plt.tight_layout()
plt.suptitle('XGBoost Optimization Comparison', fontsize=16)
plt.subplots_adjust(top=0.9)
plt.show()

# Create a comparison table
print("Optimization Results Summary:")
print("-" * 80)
print(f"{'Optimizer':<10} | {'Accuracy':<10} | {'Precision':<10} | {'Recall':<10} |
{'F1-score':<10} | Parameters")
print("-" * 80)
```

```
for name in optimizer_names:
    params_str = f"depth={results[name]['params']['max_depth']}, " + \
                 f"lr={results[name]['params']['learning_rate']:.3f}, " + \
                 f"est={results[name]['params']['n_estimators']}, " + \
                 f"alpha={results[name]['params']['alpha']:.3f}"

    print(f"{name:<10} | {results[name]['accuracy']:<10.4f} | " +
          f"{results[name]['precision']:<10.4f} | {results[name]['recall']:<10.4f} | " +
          f"{results[name]['f1_score']:<10.4f} | {params_str}")
```

3.3. DTC

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold,
cross_val_score
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# 1. Load data
df = pd.read_csv('heart-disease.csv')

# 2. Prepare features and target (drop columns as in your reference)
X = df.drop(['target', 'restecg', 'sex', 'chol', 'fbs', 'trestbps', 'slope', 'exang'], axis=1)
y = df['target'].astype(int)

# 3. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# 4. DTC parameter search space
dtc_space = {
    'max_depth': (3, 10),
    'min_samples_split': (2, 10),
    'min_samples_leaf': (1, 5),
    'criterion': (0, 1),        # 0=gini, 1=entropy
    'splitter': (0, 1),         # 0=best, 1=random
    'max_features': (0.5, 1.0), # fraction
    'ccp_alpha': (0.0, 0.05),
    'class_weight': (0, 1),     # 0=None, 1=balanced
    'max_leaf_nodes': (10, 50)
```

```
}

# 5. Helper: convert vector to DTC params
def convert_params(params):
    return {
        'max_depth': int(round(params[0])),
        'min_samples_split': int(round(params[1])),
        'min_samples_leaf': int(round(params[2])),
        'criterion': ['gini', 'entropy'][int(round(params[3]))],
        'splitter': ['best', 'random'][int(round(params[4]))],
        'max_features': params[5],
        'ccp_alpha': params[6],
        'class_weight': [None, 'balanced'][int(round(params[7]))],
        'max_leaf_nodes': int(round(params[8]))
    }

# 6. Objective function for optimization
def objective(params):
    clf = DecisionTreeClassifier(**convert_params(params), random_state=42)
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    return cross_val_score(clf, X_train, y_train, cv=cv, scoring='accuracy').mean()

# 7. Metaheuristic optimizers

def SMA_optimization(search_space, max_iter=20):
    pop_size = 5
    population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                           for _ in range(pop_size)])
    for iteration in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])
        sorted_idx = np.argsort(-fitness)
        population = population[sorted_idx]
        a = np.arctanh(1 - (iteration+1)/max_iter)
        b = 1 - (iteration+1)/max_iter
        weights = 1 + np.random.rand() * np.log10((fitness[sorted_idx] - fitness.min())
/
                (fitness.max() - fitness.min() + 1e-10) + 1)
        new_population = []
        for i in range(pop_size):
            if i < pop_size//2:
                new_pos = population[i] + a * (np.random.rand() * (population[0] -
population[i]))
            else:
```

```
            new_pos = population[i] + b * (np.random.rand() * (population[i] -
population[i-1]))
            new_population.append(np.clip(new_pos, [v[0] for v in
search_space.values()],
                                          [v[1] for v in search_space.values()]))
        population = np.array(new_population)
    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

def FOA_optimization(search_space, max_iter=20):
    num_trees = 5
    trees = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(num_trees)])
    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in trees])
        best_idx = np.argmax(fitness)
        new_trees = []
        for tree in trees:
            if objective(tree) < np.median(fitness):
                new_trees.append(tree + np.random.normal(0, 0.1,
size=len(search_space)))
            else:
                new_trees.append(tree)
        new_trees.append(trees[best_idx] + np.random.normal(0, 0.05,
size=len(search_space)))
        trees = np.clip(new_trees, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])
    best_idx = np.argmax([objective(ind) for ind in trees])
    return trees[best_idx]

def PFA_optimization(search_space, max_iter=20):
    group_size = 5
    group = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                      for _ in range(group_size)])
    leader = group[np.argmax([objective(ind) for ind in group])]
    for _ in range(max_iter):
        new_group = []
        for member in group:
            if np.array_equal(member, leader):
                new_pos = member + np.random.rand() * (leader - member)
            else:
                new_pos = member + 2*np.random.rand()*(leader - member) + \
```

```
                    np.random.rand()*(member - leader)
            new_group.append(new_pos)
        group = np.clip(new_group, [v[0] for v in search_space.values()],
                        [v[1] for v in search_space.values()])
        current_leader = group[np.argmax([objective(ind) for ind in group])]
        if objective(current_leader) > objective(leader):
            leader = current_leader
    return leader

def GAO_optimization(search_space, max_iter=20):
    pop_size = 5
    population = np.array([[np.random.uniform(low, high) for (low, high) in
search_space.values()]
                           for _ in range(pop_size)])
    for _ in range(max_iter):
        fitness = np.array([objective(ind) for ind in population])
        for i in range(pop_size):
            if np.random.rand() < 0.5:
                population[i] += np.random.normal(0, 0.1, size=len(search_space))
        best_idx = np.argmax(fitness)
        for i in range(pop_size):
            if i != best_idx:
                population[i] += np.random.rand() * (population[best_idx] -
population[i])
        population = np.clip(population, [v[0] for v in search_space.values()],
                             [v[1] for v in search_space.values()])
    best_idx = np.argmax([objective(ind) for ind in population])
    return population[best_idx]

# 8. Run all optimizations and evaluate
optimizers = {
    'SMA': SMA_optimization,
    'FOA': FOA_optimization,
    'PFA': PFA_optimization,
    'GAO': GAO_optimization
}

results = {}
plt.figure(figsize=(8, 6))  # For ROC curves

for name, optimizer in optimizers.items():
    print(f"Optimizing DTC with {name}...")
    best_params = optimizer(dtc_space)
    dtc = DecisionTreeClassifier(**convert_params(best_params), random_state=42)
```

```
    dtc.fit(X_train, y_train)
    y_pred = dtc.predict(X_test)
    y_proba = dtc.predict_proba(X_test)[:, 1]  # <-- Get positive class probability

    # Calculate metrics
    report = classification_report(y_test, y_pred, output_dict=True)
    accuracy = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba)       # <-- Compute AUC
    fpr, tpr, _ = roc_curve(y_test, y_proba)   # <-- Compute ROC curve

    results[name] = {
        'params': convert_params(best_params),
        'accuracy': accuracy,
        'precision': report['1']['precision'],
        'recall': report['1']['recall'],
        'f1_score': report['1']['f1-score'],
        'auc': auc
    }

    print(f"  Accuracy: {accuracy:.4f}")
    print(f"  Precision: {report['1']['precision']:.4f}")
    print(f"  Recall: {report['1']['recall']:.4f}")
    print(f"  F1-score: {report['1']['f1-score']:.4f}")
    print(f"  AUC: {auc:.4f}")               # <-- Print AUC
    print(f"  Parameters: {results[name]['params']}\n")

    # --- Plot ROC curve for this optimizer ---
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC={auc:.3f})')

# --- Finalize ROC plot ---
plt.plot([0, 1], [0, 1], 'k--', label='Random guess')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Decision Tree (All Optimizers)')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# 9. Plot results
metrics = ['accuracy', 'precision', 'recall', 'f1_score']
optimizer_names = list(results.keys())
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
axs = axs.flatten()
for i, metric in enumerate(metrics):
```

```
    values = [results[name][metric] for name in optimizer_names]
    bars = axs[i].bar(optimizer_names, values, color=['#1f77b4', '#ff7f0e', '#2ca02c',
'#d62728'])
    axs[i].set_title(f'DTC: {metric.capitalize()}')
    axs[i].set_ylim(0.5, 1.0)
    for bar in bars:
        height = bar.get_height()
        axs[i].text(bar.get_x() + bar.get_width()/2., height + 0.01,
                f'{height:.4f}', ha='center', va='bottom')
plt.tight_layout()
plt.suptitle('DTC Optimization Comparison', fontsize=16)
plt.subplots_adjust(top=0.9)
plt.show()

# 10. Print summary table
print("Optimization Results Summary:")
print("-" * 80)
print(f"{'Optimizer':<10} | {'Accuracy':<10} | {'Precision':<10} | {'Recall':<10} |
{'F1-score':<10} | Parameters")
print("-" * 80)
for name in optimizer_names:
    params_str = str(results[name]['params'])
    print(f"{name:<10} | {results[name]['accuracy']:<10.4f} | "
          f"{results[name]['precision']:<10.4f} | {results[name]['recall']:<10.4f} | "
          f"{results[name]['f1_score']:<10.4f} | {params_str}")
```

CHAPTER 4

SNAPSHOTS

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.313531	0.544554
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.612277	0.498835
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.000000	0.000000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.000000	1.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.000000	1.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.000000	1.000000

Fig. 1. Attributes and their statistical make-up

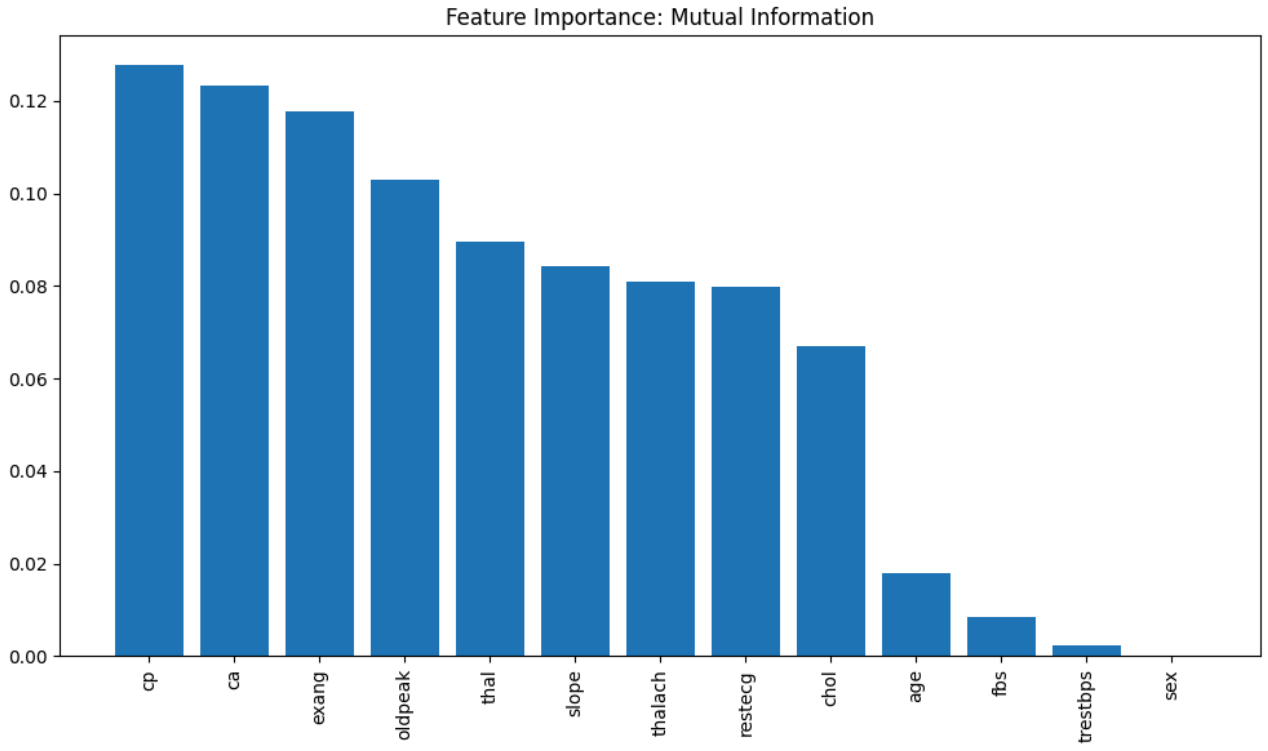

Fig. 2.1. Bar Chart: Feature Importance by Mutual Information

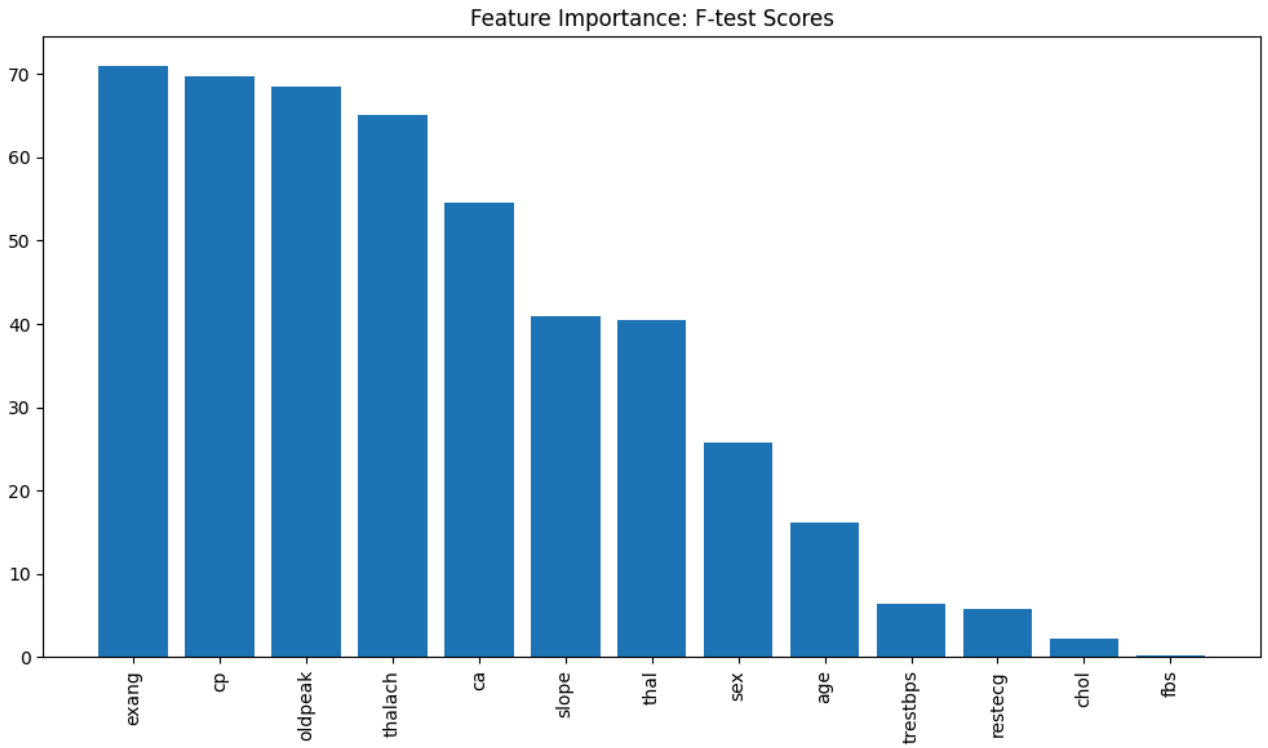

Fig. 2.2 Bar Chart: Feature Importance by F-test Scores

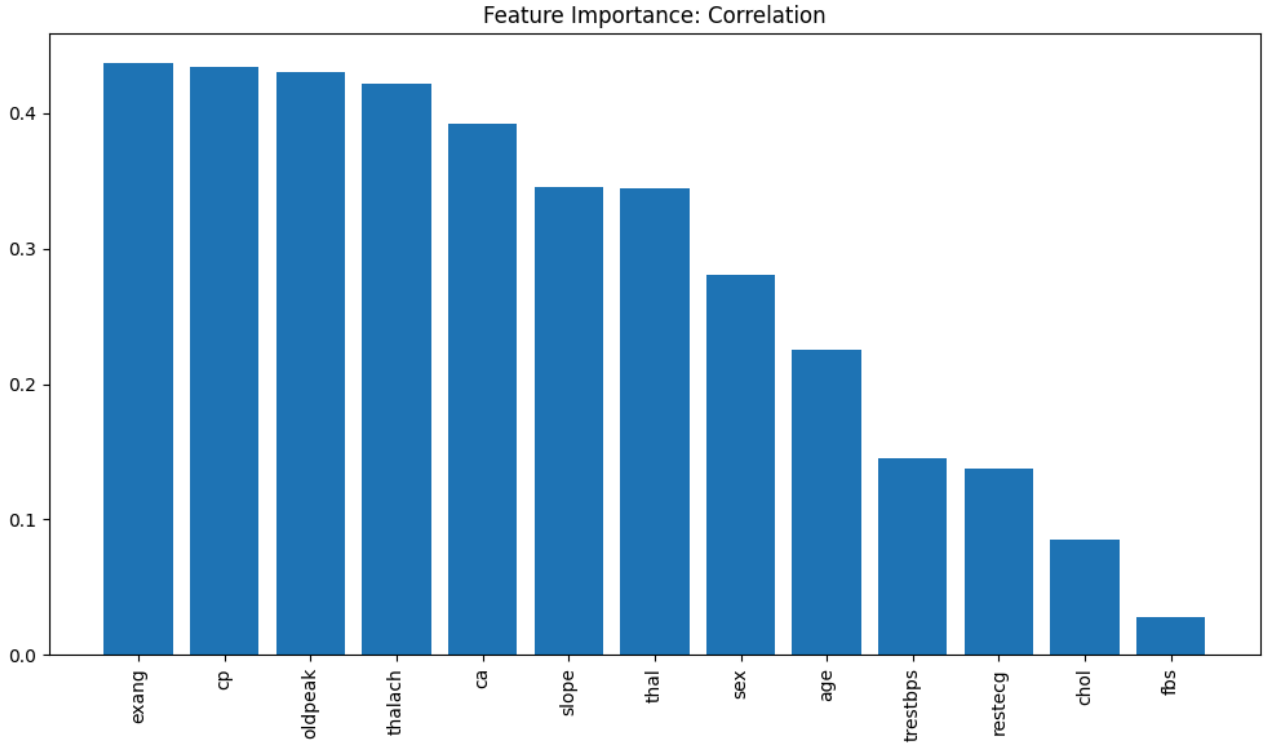

Fig. 2.3. Bar Chart: Feature Importance by Correlation

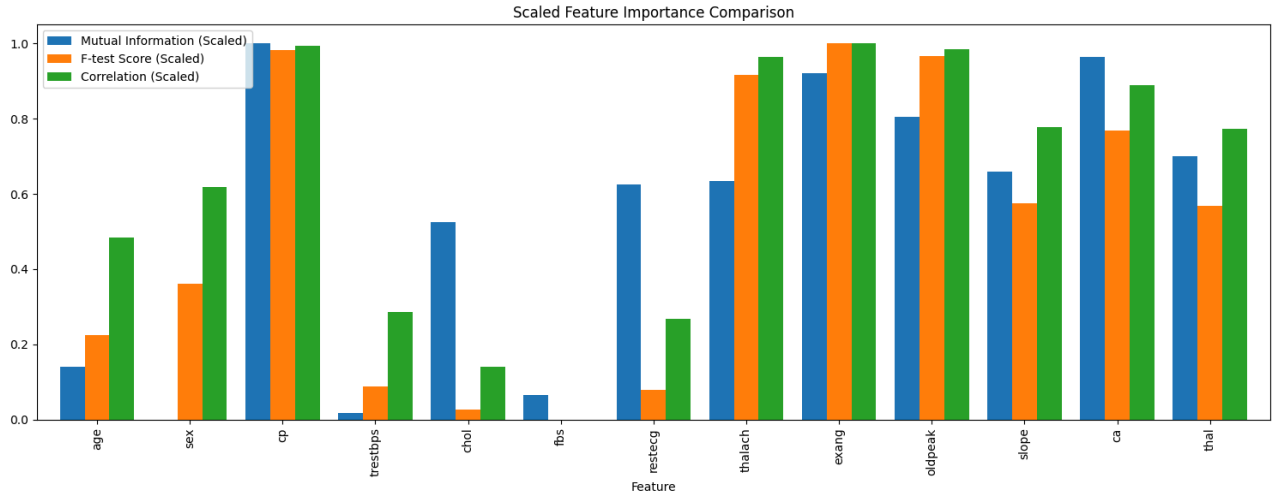

Fig. 2.4. Bar Chart: Scaled Feature Importance Comparison

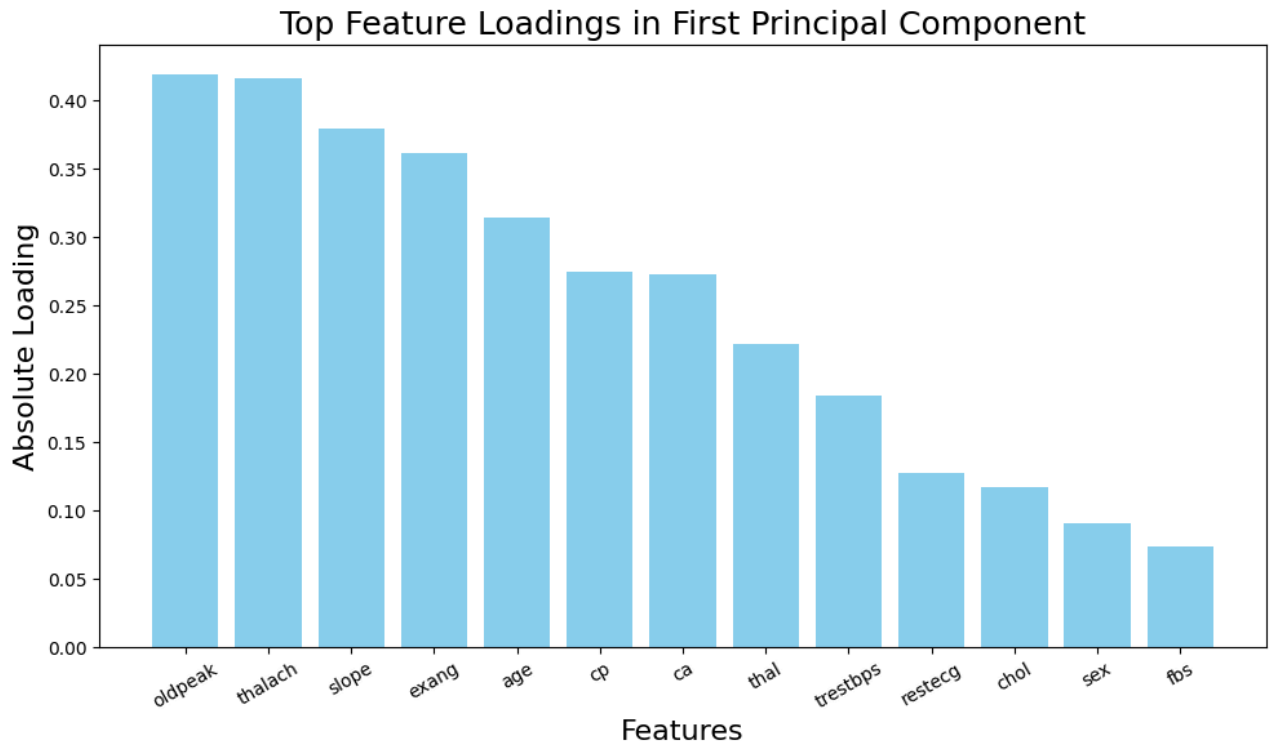

Fig. 3. Bar Chart: Top Feature Loadings in First Principal Component

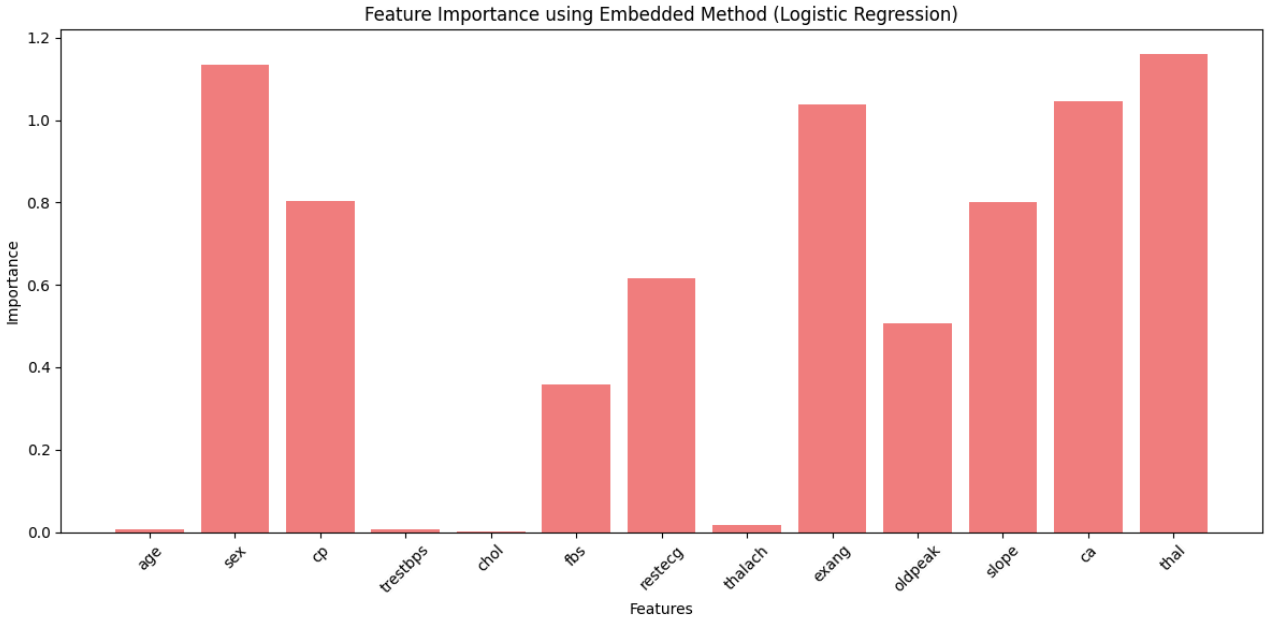

Fig. 4. Bar Chart: Feature Importance using Embedded Methods

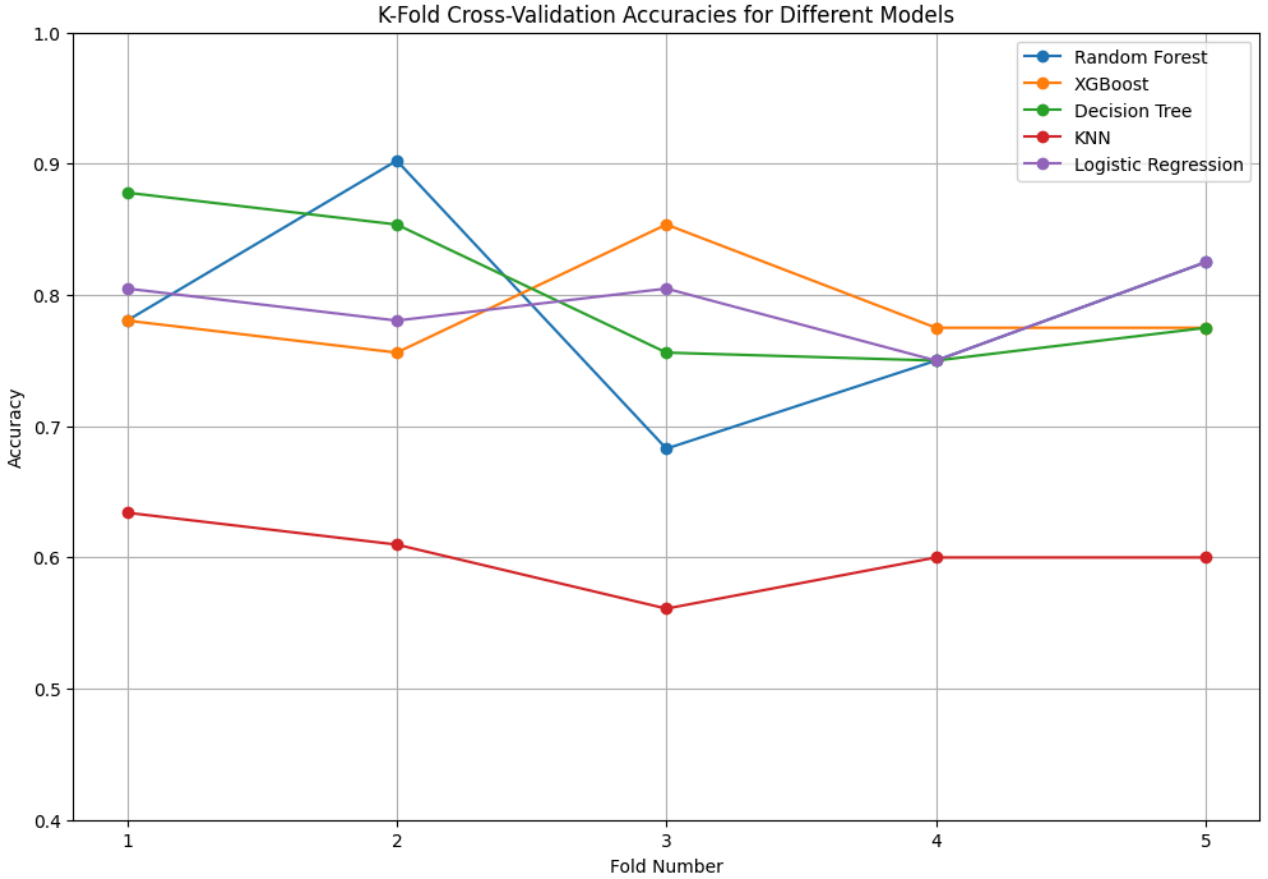

Fig. 5. Line Chart: Comparison of Model K-fold Accuracies

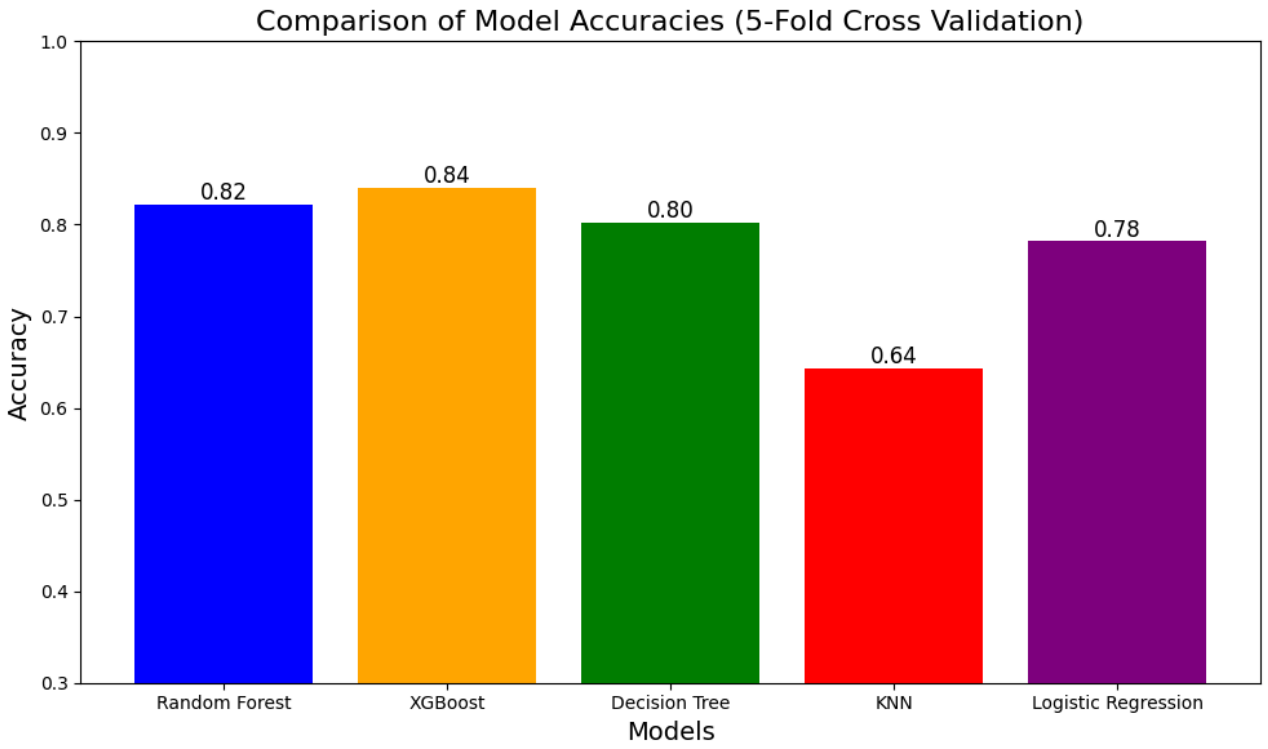

Fig. 6. Bar Chart: Comparison of Models Average Accuracies

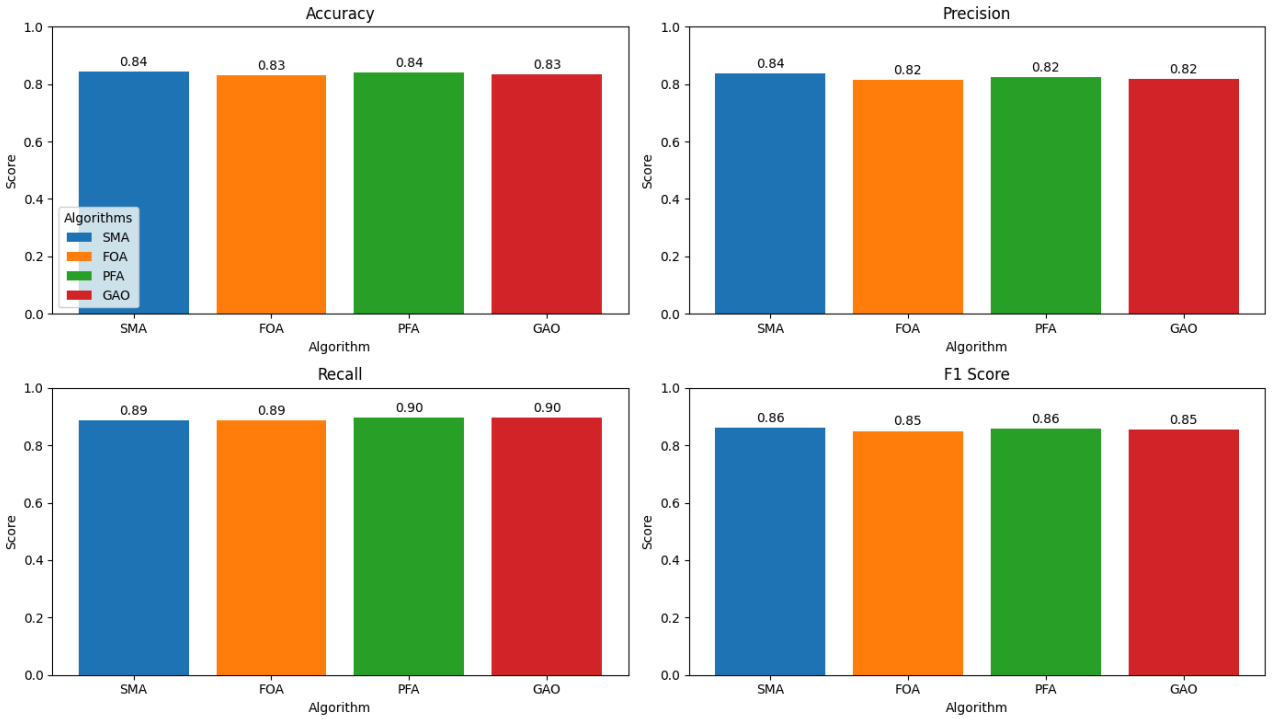

Fig. 7.1. Bar Chart: Comparison of XGBC with different optimization techniques

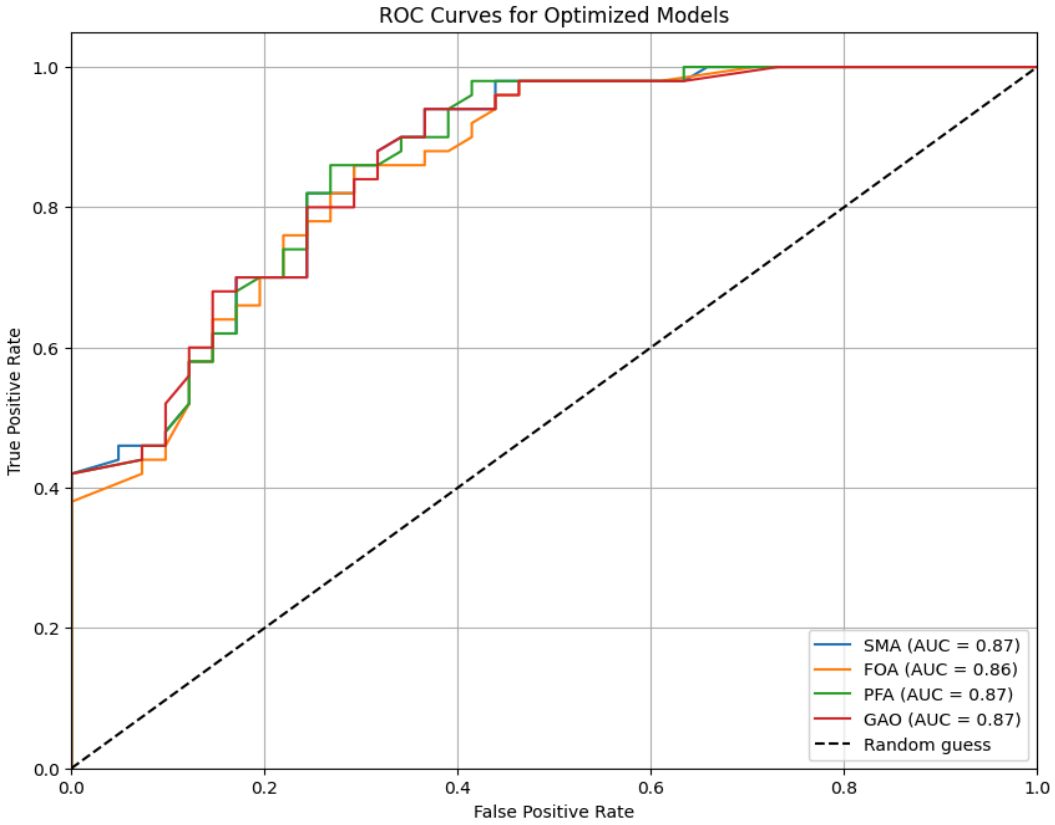

Fig. 7.2. ROC curve: XGBC with all optimizers

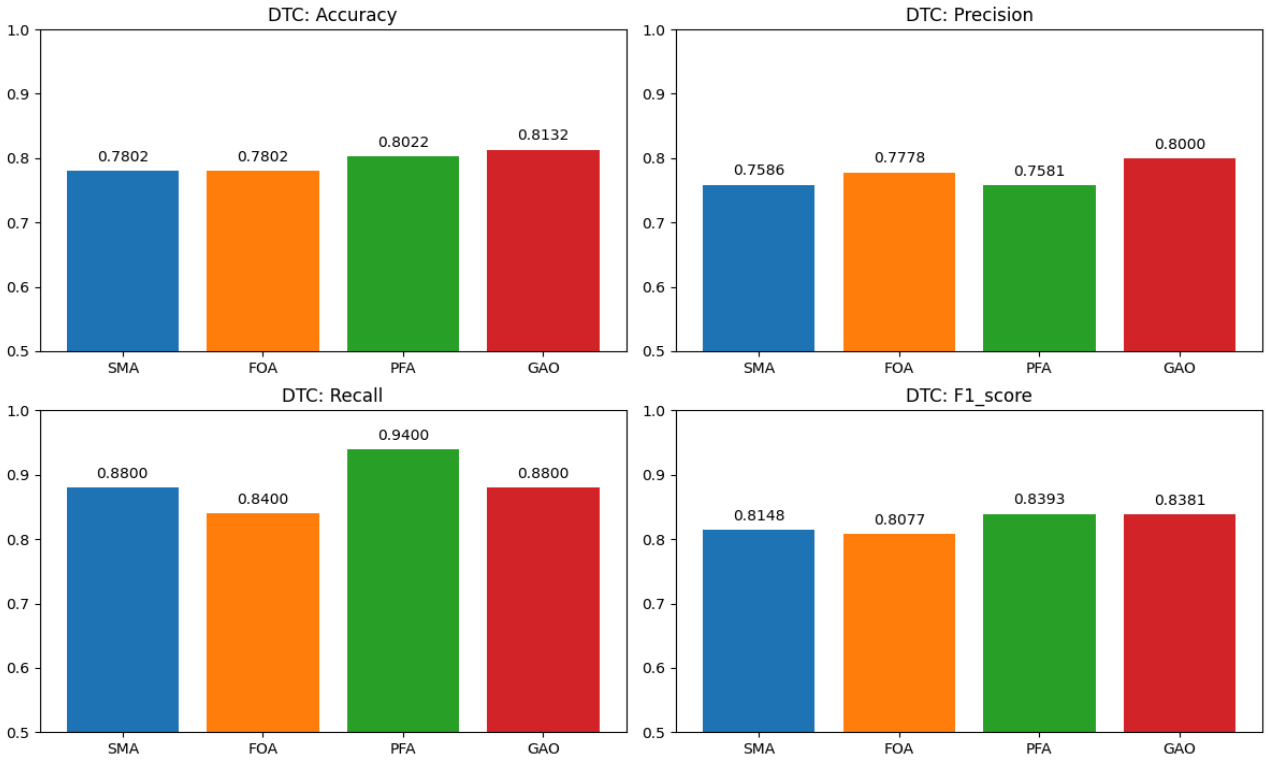

Fig. 8.1. Bar Chart: Comparison of DTC with different optimization techniques

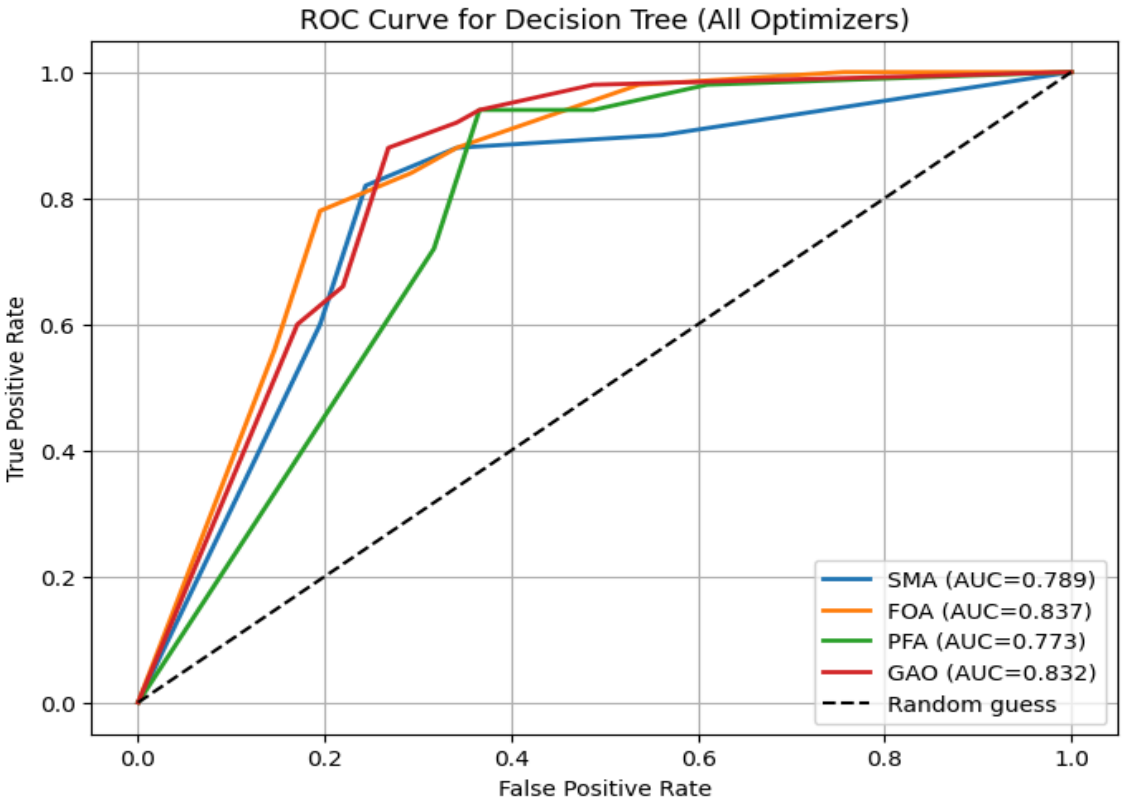

Fig. 8.2. ROC curve: DTC with all optimizers

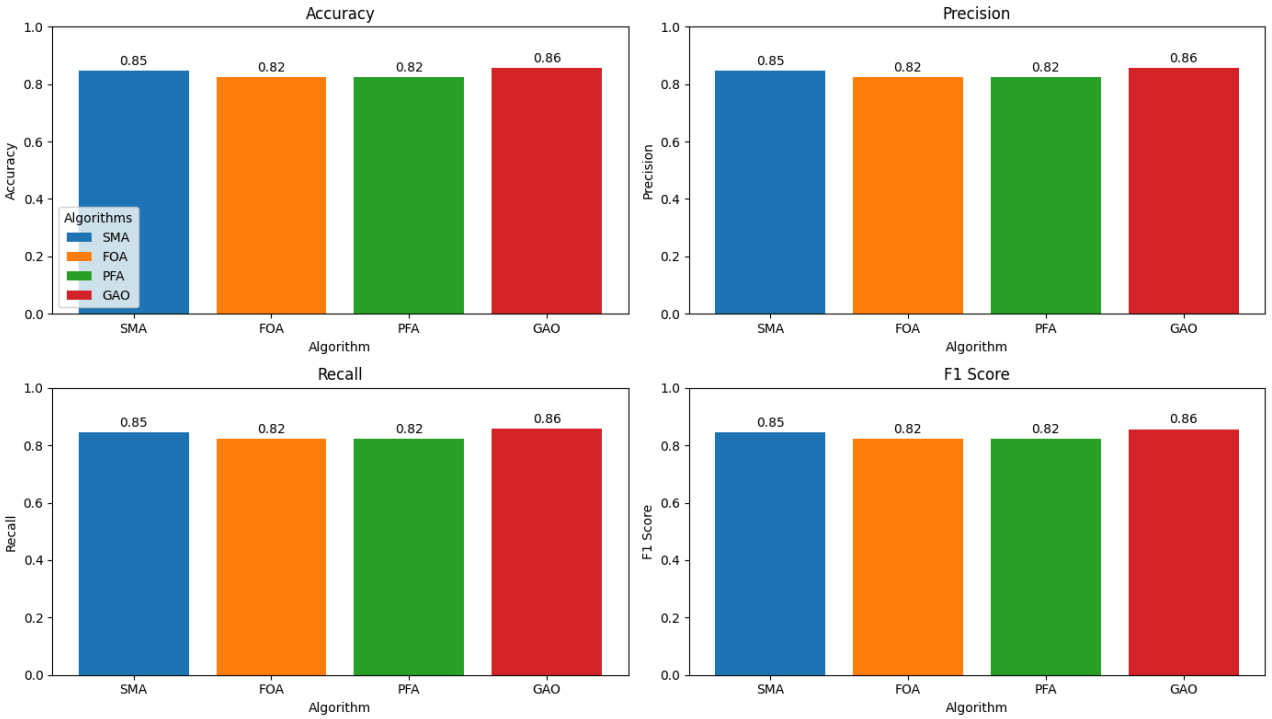

Fig. 9.1. Bar Chart: Comparison of RFC with different optimization techniques

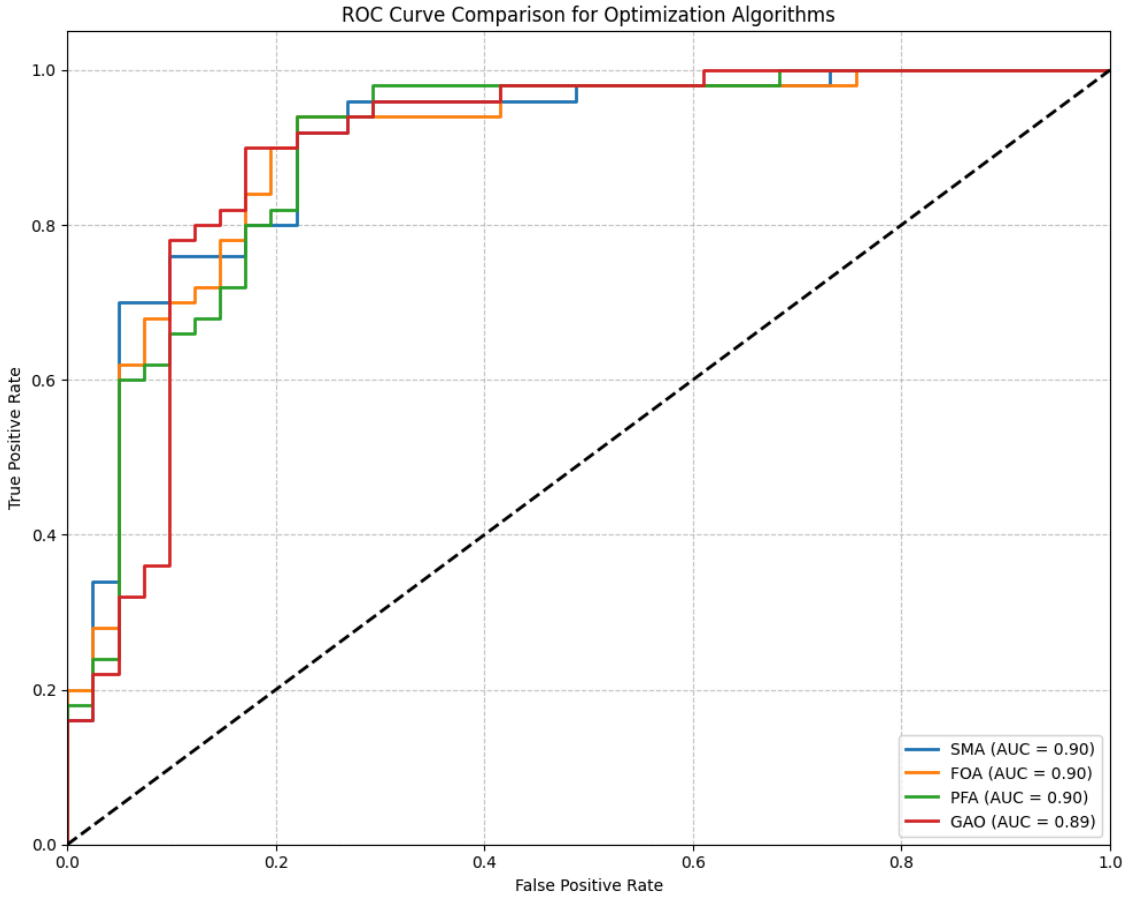

Fig. 9.2. ROC curve: RFC with all optimizers

CHAPTER 5

CONCLUSION

This project successfully demonstrates that combining machine learning models with advanced optimization algorithms can significantly improve the accuracy of heart disease prediction. Heart disease remains a leading cause of death, and early detection is essential for effective treatment. However, traditional methods often fail to capture complex patterns in patient data, which can lead to missed diagnoses.

In this study, five widely-used machine learning models-XGBoost, Random Forest, Decision Tree, K-Nearest Neighbors, and Logistic Regression-were applied. The most important features, such as age, cholesterol, and blood pressure, were carefully selected to train these models. To further enhance performance, the top models were optimized using four nature-inspired algorithms, including the Giant Armadillo Optimization technique.

The results showed that the hybrid model combining RFC with Giant Armadillo Optimization achieved the highest accuracy of 85.65%. This indicates a much more reliable prediction of heart disease compared to standard models. Overall, this approach can help doctors identify heart disease earlier and more accurately, leading to better treatment and improved patient outcomes. The method is practical and has strong potential for use in real-world clinical settings.

CHAPTER 6

REFERENCES

1. R.J.P. Princy, S. Parthasarathy, P.S.H. Jose, A.R. Lakshminarayanan, S. Jeganathan, Prediction of cardiac disease using supervised machine learning algorithms, in: 2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS), 2020, pp. 570–575, https://doi.org/10.1109/ ICICCS48265.2020.9121169.
2. V. Shorewala, Early detection of coronary heart disease using ensemble techniques, Inform. Med. Unlocked. 26 (2021) 100655, https://doi.org/10.1016/j. imu.2021.100655.
3. M.C. Das, et al., A comparative study of machine learning approaches for heart stroke prediction, in: 2023 International Conference on Smart Applications, Communications and Networking (SmartNets), 2023, pp. 1–6, https://doi.org/10.1109/SmartNets58706.2023.10216049.
4. M. Kumar, A. Rai, Surbhit, N. Kumar, Autonomic edge cloud assisted framework for heart disease prediction using RF-LRG algorithm, Multimed. Tools Appl. 83 (2) (Jan. 2024) 5929–5953, https://doi.org/10.1007/s11042-023-15736-9.
5. N. Chandrasekhar, S. Peddakrishna, Enhancing heart disease prediction accuracy through machine learning techniques and optimization, Processes 11 (4) (Apr. 2023) 1210, https://doi.org/10.3390/pr11041210.
6. G. Saranya, A. Pravin, A novel feature selection approach with integrated feature sensitivity and feature correlation for improved prediction of heart disease, J. Ambient. Intell. Humaniz. Comput. 14 (9) (Sep. 2023) 12005–12019, https://doi.org/10.1007/s12652-022-03750-y.
7. G.A. Ansari, S.S. Bhat, M.D. Ansari, S. Ahmad, J. Nazeer, A.E.M. Eljialy, Performance evaluation of machine learning techniques (MLT) for heart disease prediction, Comput. Math. Methods Med. 2023 (May 2023) 1–10, https://doi.org/ 10.1155/2023/8191261.

CHAPTER 7

APPENDIX - BASE PAPER

TITLE :

Refining heart disease prediction accuracy using hybrid machine learning techniques with novel metaheuristic algorithms

CITATION:

Haoqian Pan, Yoshiyasu Takefuji,Enhancing heart disease feature analysis with spearman's correlation with p-values,International Journal of Cardiology,Volume 430, 1 July 2025, 133207 (https://www.sciencedirect.com/science/article/pii/S0167527325002505)