23026170

Comparing SVM and Random Forests for Diabetes Prediction

Imports necessary libraries

[3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, roc_auc_score, confusion_matrix, roc_curve
```

To effectively manipulate data, you'll need to import essential libraries such as NumPy and Pandas. For data visualization, Matplotlib and Seaborn are key tools. Additionally, for building and evaluating models, as well as preprocessing data, you should incorporate modules from Scikit-learn. These libraries together provide a comprehensive toolkit for handling data science tasks efficiently.

Load and explore data

[6]:
```
# Function to load and explore data
def load_and_explore_data(filepath):
    data = pd.read_csv(filepath)
    print("First few rows of the dataset:")
    print(data.head())
    print("\nDataset Information:")
    print(data.info())
    print("\nSummary statistics of the dataset:")
    print(data.describe())

    # Plotting distributions
    data.hist(figsize=(12, 10), bins=20)
    plt.suptitle("Feature Distributions")
    plt.show()
```

```
    # Plotting correlation heatmap
    plt.figure(figsize=(10, 8))
    sns.heatmap(data.corr(), annot=True, cmap='coolwarm', fmt=".2f")
    plt.title("Correlation Heatmap")
    plt.show()

    return data
```

To analyze a dataset, we start by loading a CSV file into a Pandas DataFrame. Once the data is loaded, we print the first few rows to get a glimpse of its structure and content. Following this, we use the info() method to display details about the dataset, such as the number of entries, data types, and presence of null values. We then generate summary statistics using the describe() method to understand the distribution, central tendency, and variability of the data. To visualize the distribution of individual features, we create histograms, which help in identifying patterns such as skewness or outliers. Finally, we display a correlation heatmap to explore the relationships between different features, highlighting any potential multicollinearity or important correlations that could guide further analysis.

Split the dataset

[9]:
```
# Function to split data
def split_data(data, target_column):
    X = data.drop(columns=[target_column])
    y = data[target_column]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    return X_train, X_test, y_train, y_test
```

To prepare the dataset for modeling, first, separate it into features (X) and the target variable (y). After that, use the train_test_split function to divide the data into training and testing sets. Allocate 80% of the data to the training set and 20% to the testing set, ensuring reproducibility by setting a fixed random state during the split.

Preprocess data

[12]:
```
# Function to preprocess data
def preprocess_data(X_train, X_test):
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)
    return X_train_scaled, X_test_scaled
```

To preprocess the data, we standardize both the training and testing datasets by removing the mean and scaling to unit variance using the StandardScaler. This step is essential to ensure that the features are on the same scale, which is particularly important for models like Support Vector Machines (SVM) that are sensitive to feature scaling.

Grid Search for SVM

[15]:
```
# Function to perform grid search for SVM
def grid_search_svm(X_train, y_train):
    svm = SVC(probability=True)
    param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma':␣
 ↪['scale', 'auto']}
    grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='roc_auc',␣
 ↪verbose=1)
    grid_search.fit(X_train, y_train)
    return grid_search
```

To optimize an SVM model, GridSearchCV can be used to tune hyperparameters such as C, kernel, and gamma. This process involves setting up the SVM model and systematically searching through a predefined hyperparameter grid to find the best combination that maximizes performance. The model's effectiveness is then evaluated using 5-fold cross-validation, with the roc_auc score serving as the evaluation metric. This approach ensures a robust assessment of the model's ability to distinguish between classes.

Grid Search for Random Forest

[18]:
```
# Function to perform grid search for Random Forest
def grid_search_rf(X_train, y_train):
    rf = RandomForestClassifier(random_state=42)
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10]
    }
    grid_search = GridSearchCV(rf, param_grid, cv=5, scoring='roc_auc',␣
 ↪verbose=1)
    grid_search.fit(X_train, y_train)
    return grid_search
```

Grid search for a Random Forest classifier is a systematic way to tune hyperparameters to optimize model performance. Similar to how it's done for an SVM, grid search can be applied to Random Forest to find the best combination of parameters. Key hyperparameters include the number of trees (n_estimators), which determines how many trees are in the forest; the maximum depth of the trees (max_depth), which controls how deep each tree can grow; and the minimum number of samples required to split a node (min_samples_split), which influences how nodes are split during the training process. By evaluating different combinations of these parameters, grid search helps identify the optimal settings that improve the model's accuracy and generalization ability.

Evaluate the model

[21]:
```python
# Function to evaluate the model
def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    precision = precision_score(y_test, predictions)
    recall = recall_score(y_test, predictions)
    f1 = f1_score(y_test, predictions)
    roc_auc = roc_auc_score(y_test, model.predict_proba(X_test)[:, 1])

    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"ROC-AUC Score: {roc_auc:.4f}")

    # Plotting confusion matrix
    plot_confusion_matrix(y_test, predictions)

    # Plotting ROC Curve
    plot_roc_curve(y_test, model.predict_proba(X_test)[:, 1], roc_auc)

    return accuracy, precision, recall, f1, roc_auc
```

To evaluate the model, start by using the trained model to make predictions on the test set. Next, compute various evaluation metrics to assess its performance, including accuracy, precision, recall, F1 score, and ROC-AUC score. To visually analyze the model's performance, plot the confusion matrix and the ROC curve. Finally, return all calculated metrics to facilitate a comprehensive comparison and analysis of the model's effectiveness.

Plot Confusion Matrix

[24]:
```python
# Function to plot confusion matrix
def plot_confusion_matrix(y_test, predictions):
    conf_matrix = confusion_matrix(y_test, predictions)
    plt.figure(figsize=(6, 4))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix')
    plt.ylabel('Actual Class')
    plt.xlabel('Predicted Class')
    plt.show()
```

A confusion matrix is a valuable tool for evaluating the performance of a classification model. It visually represents the counts of four key types of predictions: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). True positives are instances where the model correctly predicted the positive class, while true negatives are instances where the model correctly

predicted the negative class. False positives occur when the model incorrectly predicts the positive class for a negative instance, and false negatives occur when the model incorrectly predicts the negative class for a positive instance. By plotting the confusion matrix, you can gain insight into how well the model is performing, including its accuracy, precision, recall, and the balance between false positives and false negatives. This visualization helps identify areas where the model may need improvement and provides a comprehensive view of its classification capabilities.

Plot ROC Curve

[27]:
```
# Function to plot ROC Curve
def plot_roc_curve(y_test, y_proba, roc_auc):
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.
 ↪2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
    plt.show()
```

The Receiver Operating Characteristic (ROC) curve is a graphical representation used to assess the performance of a classification model. It plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. This curve illustrates the trade-off between sensitivity and specificity, showing how the model's ability to correctly identify positive cases (true positives) changes with different decision thresholds. The area under the ROC curve (AUC) is a key metric derived from this plot, providing a single value that summarizes the model's overall ability to discriminate between the classes. A higher AUC value indicates a better performance, as it reflects a higher true positive rate and a lower false positive rate across different thresholds.

Main function to run the entire process

[30]:
```
# Main function to run the whole process
def main():
    # Load and explore the dataset
    data = load_and_explore_data("diabetes.csv")

    # Split the dataset
    X_train, X_test, y_train, y_test = split_data(data, "Outcome")

    # Preprocess the data
    X_train_scaled, X_test_scaled = preprocess_data(X_train, X_test)
```

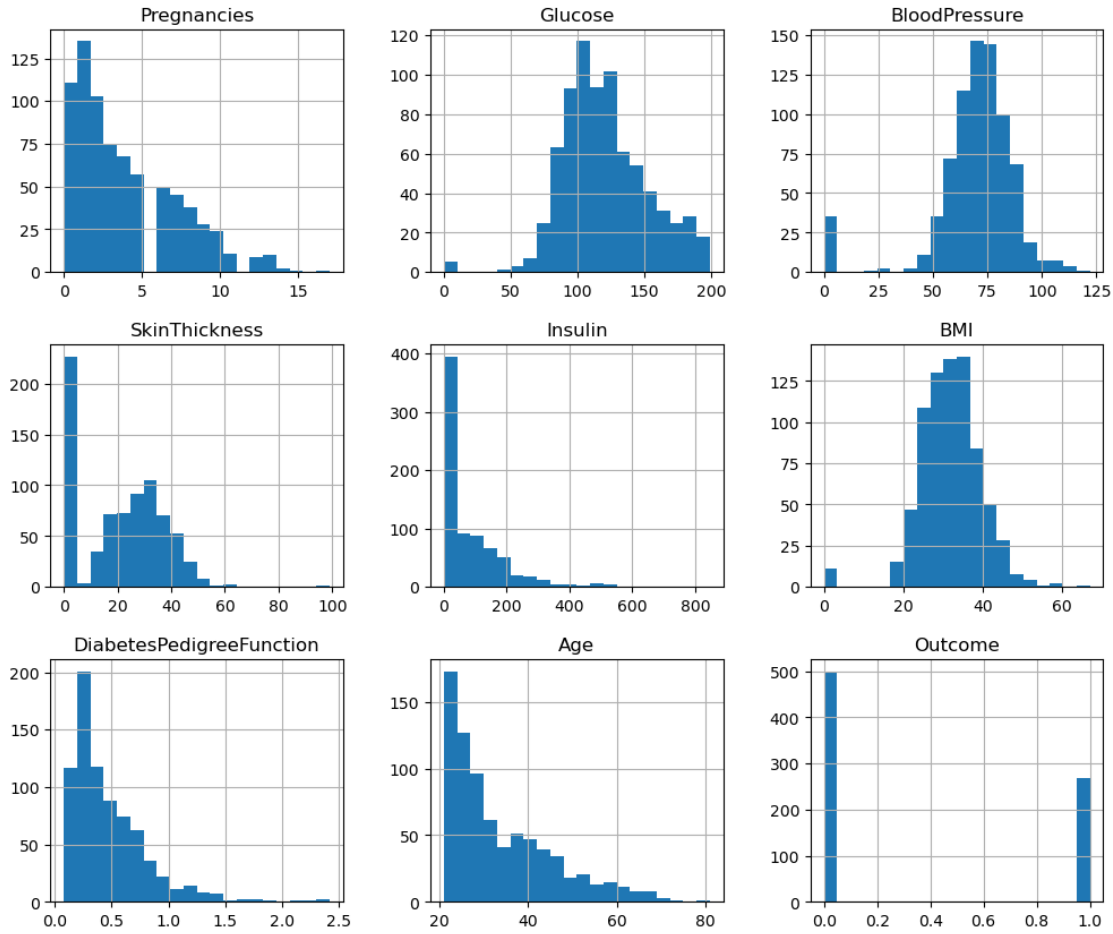
```
    # SVM Model with Grid Search
    print("\nTraining SVM model...")
    svm_grid = grid_search_svm(X_train_scaled, y_train)
    print("\nBest SVM Parameters:", svm_grid.best_params_)
    print("\nEvaluating SVM model...")
    svm_metrics = evaluate_model(svm_grid.best_estimator_, X_test_scaled,␣
↪y_test)

    # Random Forest Model with Grid Search
    print("\nTraining Random Forest model...")
    rf_grid = grid_search_rf(X_train, y_train)
    print("\nBest Random Forest Parameters:", rf_grid.best_params_)
    print("\nEvaluating Random Forest model...")
    rf_metrics = evaluate_model(rf_grid.best_estimator_, X_test, y_test)

    # Comparison of the two models
    print("\nComparison of SVM and Random Forest Models:")
    metrics = ["Accuracy", "Precision", "Recall", "F1 Score", "ROC-AUC"]
    print(f"{'Metric':<10} {'SVM':<10} {'Random Forest':<15}")
    for metric, svm_value, rf_value in zip(metrics, svm_metrics, rf_metrics):
        print(f"{metric:<10} {svm_value:<10.4f} {rf_value:<15.4f}")
```

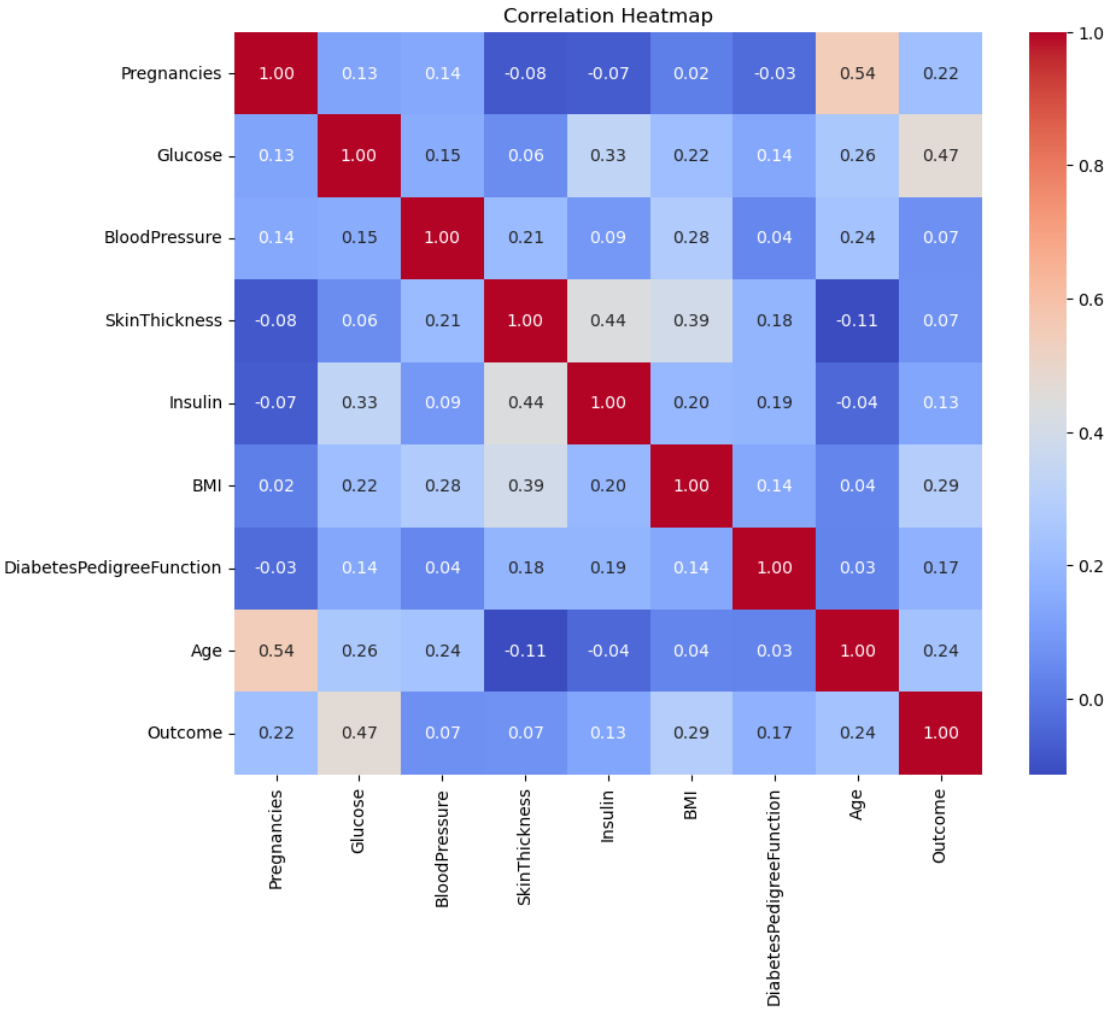
To run the entire process, the main function begins by loading and exploring the dataset to understand its structure and characteristics. It then splits the data into training and testing sets to ensure proper evaluation of the models. Next, the function preprocesses the data by scaling it to standardize feature values. Following preprocessing, the function trains and tunes a Support Vector Machine (SVM) model using grid search to find the best hyperparameters, and evaluates its performance. Similarly, it trains and tunes a Random Forest model using grid search and evaluates it. Finally, the function compares the performance of the SVM and Random Forest models using various evaluation metrics to determine which model performs better.

[32]:
```
if __name__ == "__main__":
    main()
```

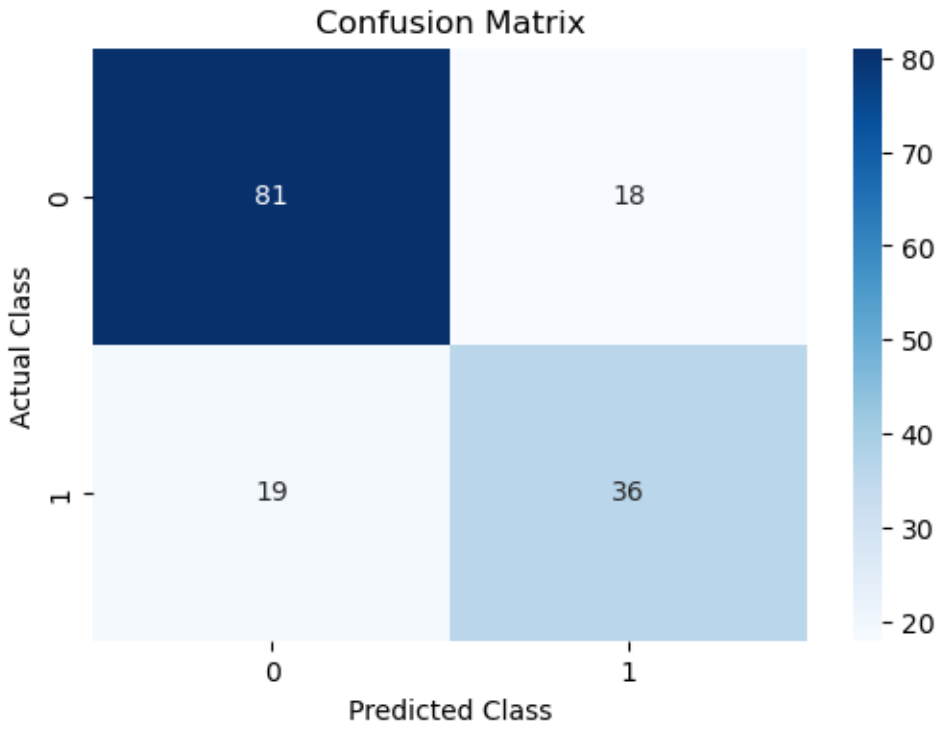
```
First few rows of the dataset:
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

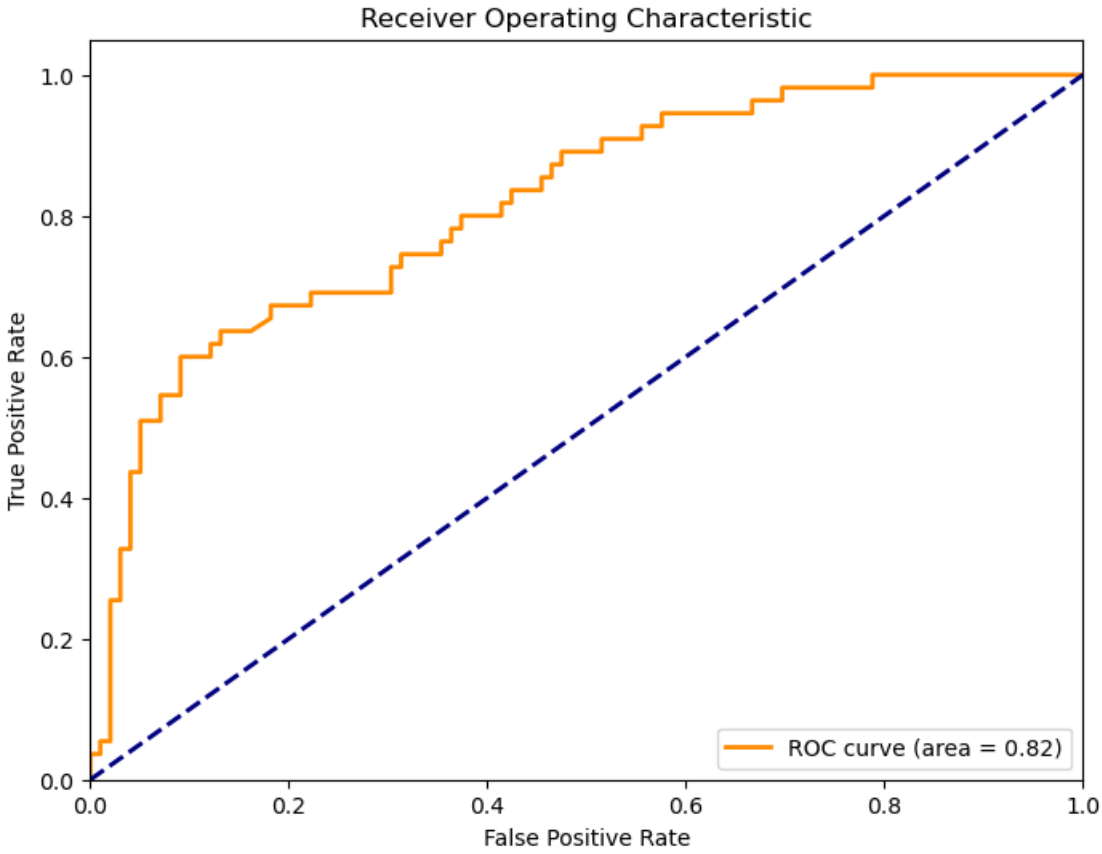
```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
None

Summary statistics of the dataset:
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  768.000000     768.000000     768.000000  768.000000
mean      3.845052  120.894531      69.105469      20.536458   79.799479
std       3.369578   31.972618      19.355807      15.952218  115.244002
min       0.000000    0.000000       0.000000       0.000000    0.000000
25%       1.000000   99.000000      62.000000       0.000000    0.000000
50%       3.000000  117.000000      72.000000      23.000000   30.500000
75%       6.000000  140.250000      80.000000      32.000000  127.250000
max      17.000000  199.000000     122.000000      99.000000  846.000000

              BMI  DiabetesPedigreeFunction         Age     Outcome
count  768.000000                768.000000  768.000000  768.000000
mean    31.992578                  0.471876   33.240885    0.348958
std      7.884160                  0.331329   11.760232    0.476951
min      0.000000                  0.078000   21.000000    0.000000
25%     27.300000                  0.243750   24.000000    0.000000
50%     32.000000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
```


Feature Distributions
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
Outcome

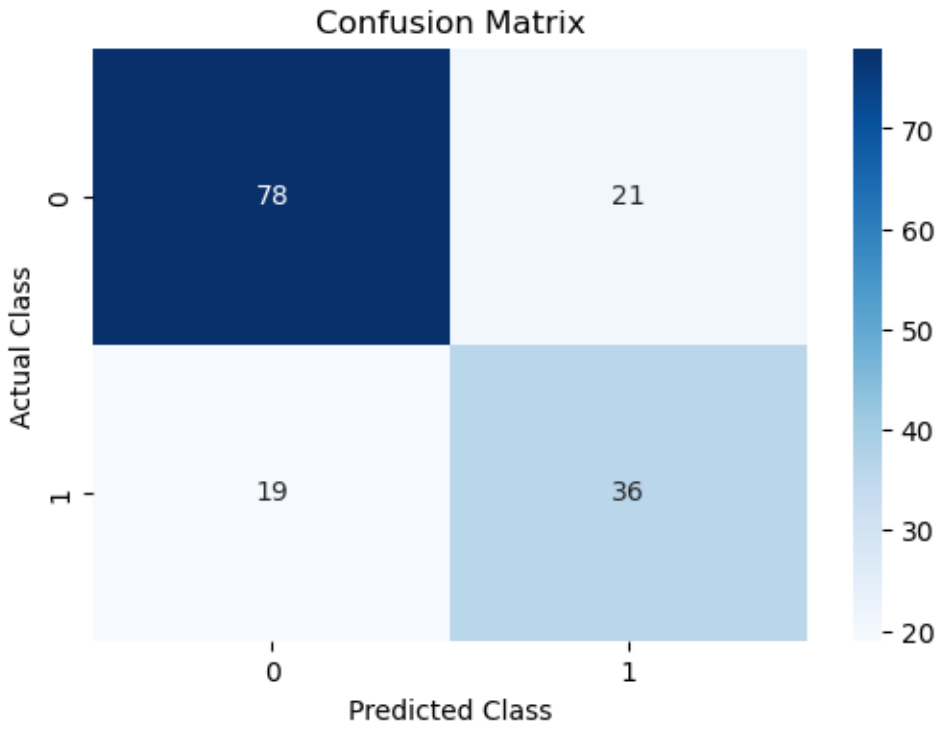


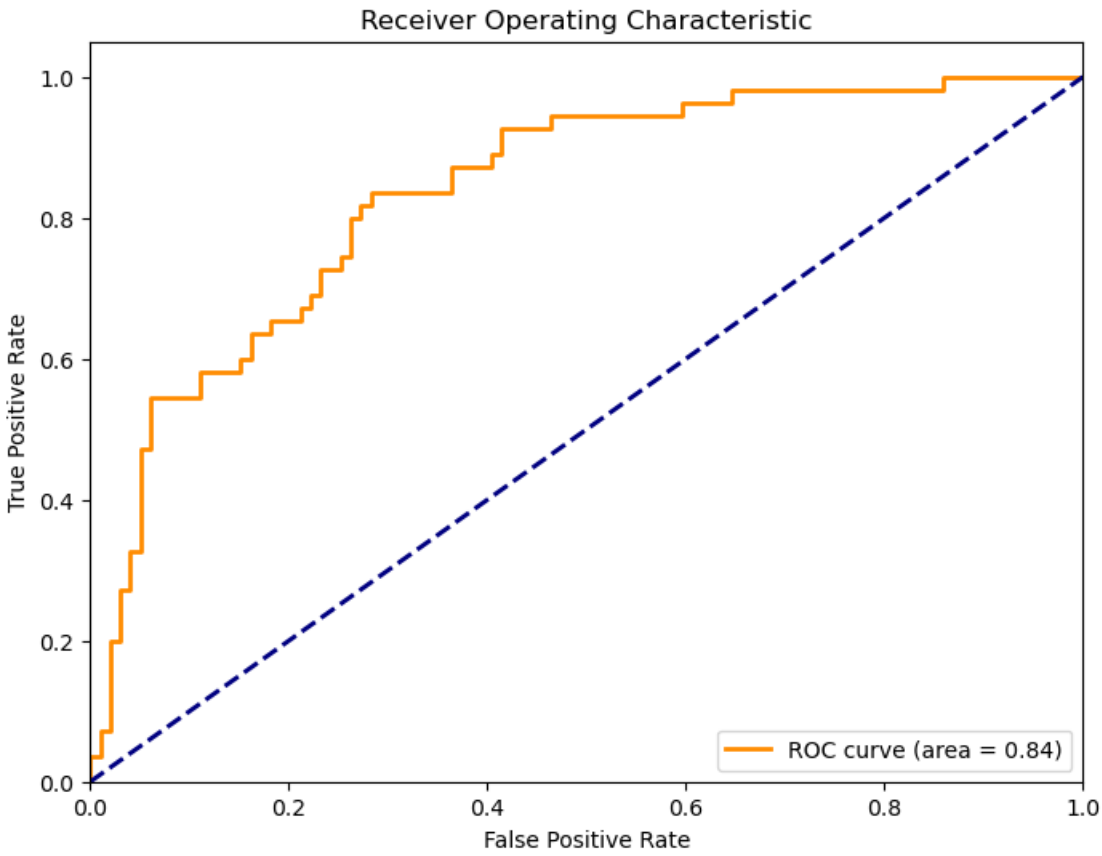

```
Training SVM model…
Fitting 5 folds for each of 12 candidates, totalling 60 fits

Best SVM Parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}

Evaluating SVM model…
Accuracy: 0.7597
Precision: 0.6667
Recall: 0.6545
F1 Score: 0.6606
ROC-AUC Score: 0.8167
```


Confusion Matrix
81
18
19
36
0
1
Actual Class
0
1
Predicted Class
80
70
60
50
40
30
20




```
Training Random Forest model…
Fitting 5 folds for each of 36 candidates, totalling 180 fits

Best Random Forest Parameters: {'max_depth': 10, 'min_samples_split': 10,
'n_estimators': 200}

Evaluating Random Forest model…
Accuracy: 0.7403
Precision: 0.6316
Recall: 0.6545
F1 Score: 0.6429
ROC-AUC Score: 0.8373
```


Confusion Matrix
78
21
19
36
Actual Class
0
1
0
1
Predicted Class
70
60
50
40
30
20




```
Comparison of SVM and Random Forest Models:
Metric     SVM        Random Forest
Accuracy   0.7597     0.7403
Precision  0.6667     0.6316
Recall     0.6545     0.6545
F1 Score   0.6606     0.6429
ROC-AUC    0.8167     0.8373
```

Advantages and Disadvantages of SVM and Random Forest

SVM(Support Vector Machine)

Advantages:

1. Effective in high-dimensional spaces: SVM is effective in cases where the number of dimensions is greater than the number of samples.
2. Memory Efficient: It uses a subset of training points in the decision function (called support vectors), making it memory efficient.

3. Robust to Overfitting: Especially in high-dimensional space, SVM is robust to overfitting.

Disadvantages:

1. Computationally Intensive: Training can be slow for large datasets.
2. Choice of Kernel: The performance heavily depends on the choice of the kernel and the regularization parameter.
3. Non-Probabilistic: SVM inherently does not provide probability estimates; these are calculated using cross-validation which can be computationally expensive.

Random Forest

Advantages:es.

1. Handles Missing Values: Random Forest can handle missing values internally.
2. Reduces Overfitting: By averaging multiple trees, Random Forest reduces the risk of overfitting.
3. Feature Importance: It can provide insights into feature importan trees.

Disadvantages:

1. Computationally Intensive: It can be computationally intensive due to the construction of a large number of trees.
2. Memory Usage: It requires more memory as it builds multiple trees.

Model Performance Comparison

SVM Performance:

1. Accuracy: 0.7597
2. Precision: 0.6667
3. Recall: 0.6545
4. F1 Score: 0.6606
5. ROC-AUC Score: 0.8167

Random Forest Performance:

1. Accuracy: 0.7403
2. Precision: 0.6316
3. Recall: 0.6545
4. F1 Score: 0.6429
5. ROC-AUC Score: 0.8373

Both models have shown competitive performance with slight differences. However, the Random Forest model has a slightly higher ROC-AUC compared to the SVM.

Best Model Choice

1. Random Forest has a slight edge due to its higher ROC-AUC score, which suggests it has a better overall performance in distinguishing between the positive and negative classes.
2. SVM performed slightly better in terms of accuracy and F1 score, which indicates it may be better at minimizing both false positives and false negatives.

Conclusion

While both models are viable options, the choice of the model can depend on the specific requirements and constraints of the task at hand. If interpretability and feature importance are crucial, Random Forest might be preferred. If the dataset is high-dimensional and computational resources are available, SVM could be a good choice. In this particular comparison, Random Forest appears to provide a marginally better performance.