Pattern Recognition Using the Iris Dataset

To explore the Iris dataset, extract statistical features, and implement pattern recognition techniques to classify the data into its respective species categories.

Part 1: Data Preparation

1. **Importing the Dataset:**
    - Load the Iris dataset using `sklearn.datasets` or a CSV file.
    - Display the first 5 rows of the dataset.
2. **Exploratory Data Analysis:**
    - Identify the features and labels in the dataset.
    - Plot the distribution of each feature using histograms.
    - Visualize the dataset using a scatterplot matrix.

Part 2: Feature Extraction

1. **Statistical Features:**
    - Compute the following statistical features for each numerical column (sepal length, sepal width, petal length, petal width):
        - Mean
        - Median
        - Variance
        - Standard Deviation
        - Minimum and Maximum
    - Normalize the features (e.g., using Min-Max scaling or Z-score normalization).
2. **Feature Selection:**
    - Discuss which features seem most relevant for classification based on their statistical properties.

Part 3: Pattern Recognition

1. **Data Splitting:**
    - Split the dataset into training and testing sets (e.g., 70% training, 30% testing).
2. **Model Implementation:**
    - Implement the following pattern recognition models using `scikit-learn`:
        - K-Nearest Neighbors (KNN)
        - Support Vector Machine (SVM)

    - Use cross-validation to fine-tune hyperparameters (e.g., k for KNN, kernel for SVM).
3. **Evaluation:**
    - Evaluate each model using metrics such as:
        - Accuracy
        - Precision
        - Recall
        - F1-score
    - Plot confusion matrices for each model.

Part 4: Comparison and Conclusion

1. **Model Comparison:**
    - Compare the performance of the models using a bar chart.
    - Discuss which model performed best and why.
2. **Future Improvements:**
    - Suggest potential improvements to the models or additional features that could be used.

Deliverables:

1. A Jupyter Notebook or Python script with the complete implementation.
2. A short report summarizing:
    - The feature extraction process
    - Model evaluation results

Assignment 10

Pattern Recognition

Part 1: Data Preparation

1. Importing the Dataset:

○ Load the Iris dataset using sklearn.datasets or a CSV file.

○ Display the first 5 rows of the dataset.

```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
iris = load_iris()
# Convert to DataFrame for better readability
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target
df.head()
```

Output:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

2. Exploratory Data Analysis:

○ Identify the features and labels in the dataset.

```
# Features and labels
features = iris.feature_names
labels = iris.target_names
print(f"Features: {features}")
print(f"Labels: {labels}")
```

Output:

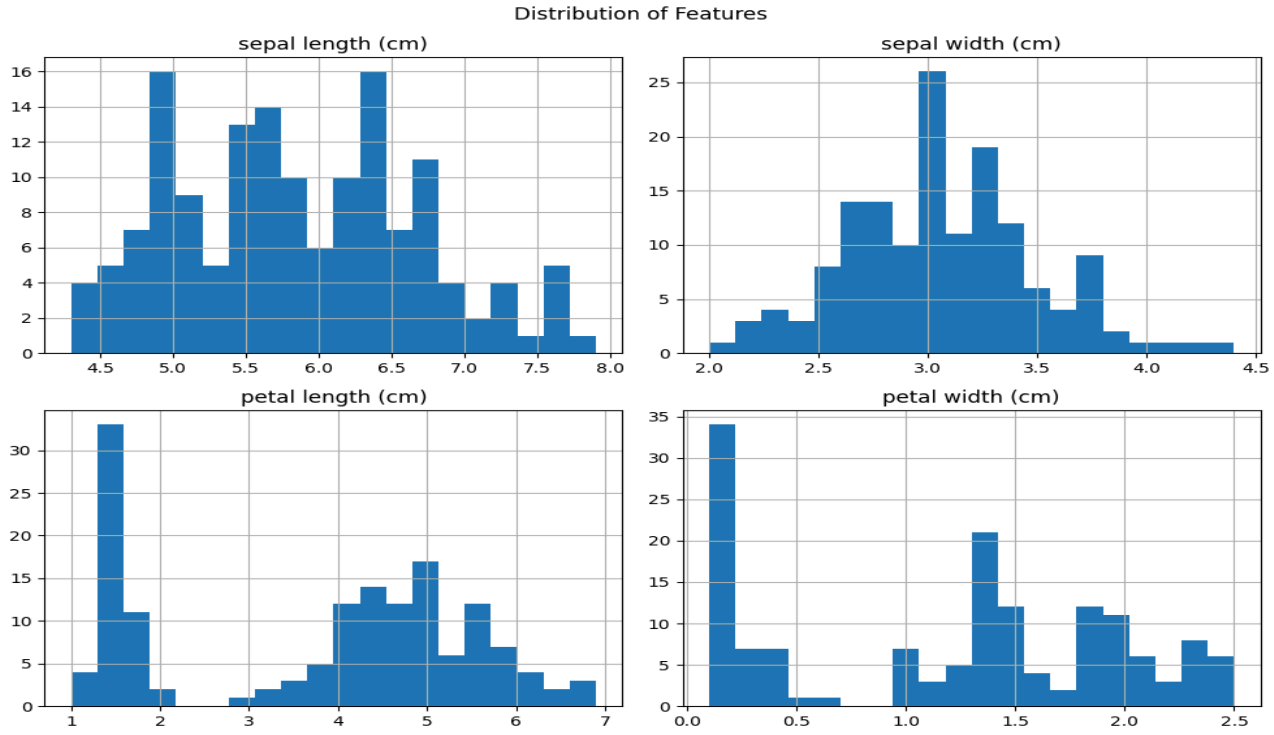
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Labels: ['setosa' 'versicolor' 'virginica']

○ Plot the distribution of each feature using histograms.

```
import matplotlib.pyplot as plt
import seaborn as sns
# Set up the plot for histograms
df[features].hist(figsize=(10, 8), bins=20)
```

```
plt.suptitle('Distribution of Features')
plt.tight_layout()
plt.show()
```

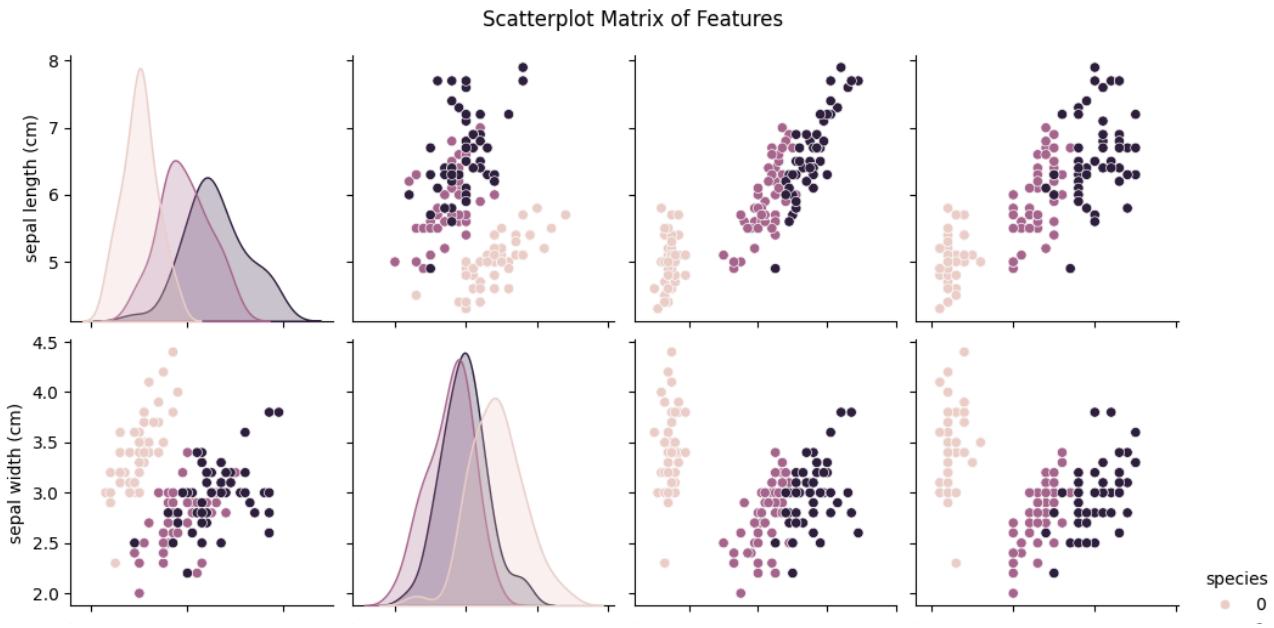
Output:

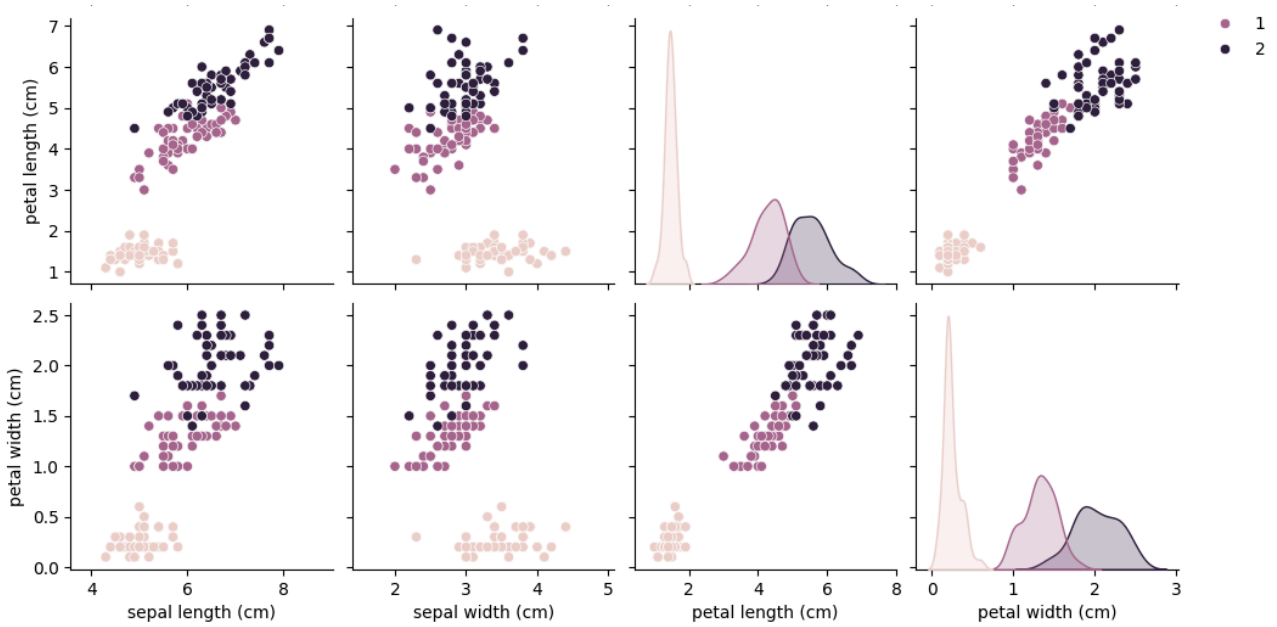


○ Visualize the dataset using a scatterplot matrix.

```
# Scatterplot matrix using seaborn
sns.pairplot(df, hue="species", vars=features)
plt.suptitle('Scatterplot Matrix of Features', y=1.02)
plt.show()
```

Output:





Part 2: Feature Extraction

1. Statistical Features:

○ Compute the following statistical features for each numerical column (sepal length, sepal width, petal length, petal width):

■ Mean
■ Median
■ Variance
■ Standard Deviation
■ Minimum and Maximum

```
# Compute statistical features
statistics = df[features].agg(['mean', 'median', 'var', 'std', 'min',
'max'])
statistics
```

Output:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
mean	5.843333	3.057333	3.758000	1.199333
median	5.800000	3.000000	4.350000	1.300000
var	0.685694	0.189979	3.116278	0.581006
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
max	7.900000	4.400000	6.900000	2.500000

○ Normalize the features (e.g., using Min-Max scaling or Z-score normalization).

```
from sklearn.preprocessing import MinMaxScaler
```

```
# Normalize the features using Min-Max scaling
scaler = MinMaxScaler()
normalized_df = df[features].copy()
normalized_df[features] = scaler.fit_transform(df[features])
# Display the normalized dataframe
normalized_df.head()
```

Output:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	0.222222	0.625000	0.067797	0.041667
1	0.166667	0.416667	0.067797	0.041667
2	0.111111	0.500000	0.050847	0.041667
3	0.083333	0.458333	0.084746	0.041667
4	0.194444	0.666667	0.067797	0.041667

2. Feature Selection:
○ Discuss which features seem most relevant for classification based on their statistical properties.

→ Based on the statistical properties (e.g., mean, variance), petal-related features (petal length and petal width) seem more relevant for classification since they show higher variance and are more distinct across species. Sepal-related features are useful but less discriminative.

Part 3: Pattern Recognition
1. Data Splitting:
○ Split the dataset into training and testing sets (e.g., 70% training, 30% testing).

```
from sklearn.model_selection import train_test_split
# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
# Output dataset sizes
print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")
```

Output:

```
Training set size: 105 samples
Testing set size: 45 samples
```

2. Model Implementation:
○ Implement the following pattern recognition models using scikit-learn:
○ Use cross-validation to fine-tune hyperparameters (e.g., k for KNN, kernel for SVM).

■ K-Nearest Neighbors (KNN)

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
# Hyperparameter tuning for KNN
param_grid_knn = {'n_neighbors': range(1, 31)}
knn = GridSearchCV(KNeighborsClassifier(), param_grid_knn, cv=5)
knn.fit(X_train, y_train)
# Best KNN model
best_knn = knn.best_estimator_
print(f"Best KNN parameters: {knn.best_params_}")
```

Output:

```
Best KNN parameters: {'n_neighbors': 6}
```

■ Support Vector Machine (SVM)

```
from sklearn.svm import SVC
# Hyperparameter tuning for SVM
param_grid_svm = {'kernel': ['linear', 'rbf', 'poly'], 'C': [0.1, 1, 10]}
svm = GridSearchCV(SVC(), param_grid_svm, cv=5)
svm.fit(X_train, y_train)
# Best SVM model
best_svm = svm.best_estimator_
print(f"Best SVM parameters: {svm.best_params_}")
```

Output:

```
Best SVM parameters: {'C': 0.1, 'kernel': 'poly'}
```

3. Evaluation:

○ Evaluate each model using metrics such as:

■ Accuracy

■ Precision

■ Recall

■ F1-score

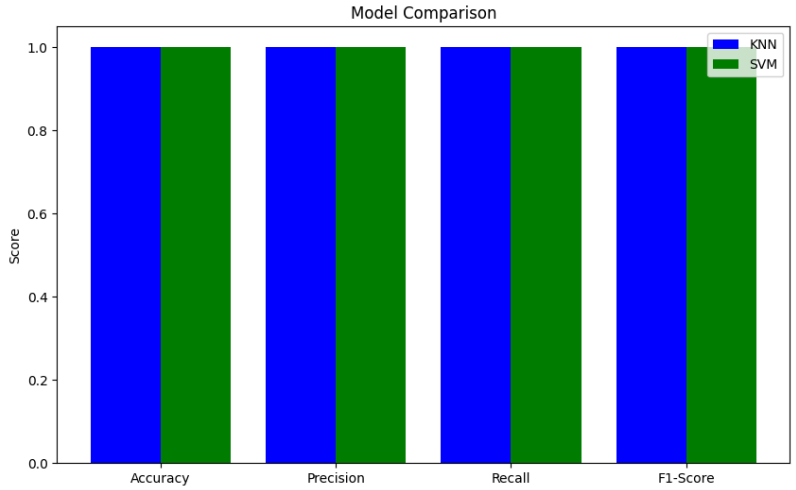
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# Metrics for KNN
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn, average='weighted')
recall_knn = recall_score(y_test, y_pred_knn, average='weighted')
f1_knn = f1_score(y_test, y_pred_knn, average='weighted')
# Metrics for SVM
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm, average='weighted')
```

```
recall_svm = recall_score(y_test, y_pred_svm, average='weighted')
f1_svm = f1_score(y_test, y_pred_svm, average='weighted')
# Bar Chart for Comparison
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
knn_scores = [accuracy_knn, precision_knn, recall_knn, f1_knn]
svm_scores = [accuracy_svm, precision_svm, recall_svm, f1_svm]

x = range(len(metrics))
plt.figure(figsize=(10, 6))
plt.bar(x, knn_scores, width=0.4, label='KNN', align='center',
color='blue')
plt.bar([i + 0.4 for i in x], svm_scores, width=0.4, label='SVM',
align='center', color='green')
plt.xticks([i + 0.2 for i in x], metrics)
plt.ylabel('Score')
plt.title('Model Comparison')
plt.legend()
plt.show()
```

Output:



○ Plot confusion matrices for each model.

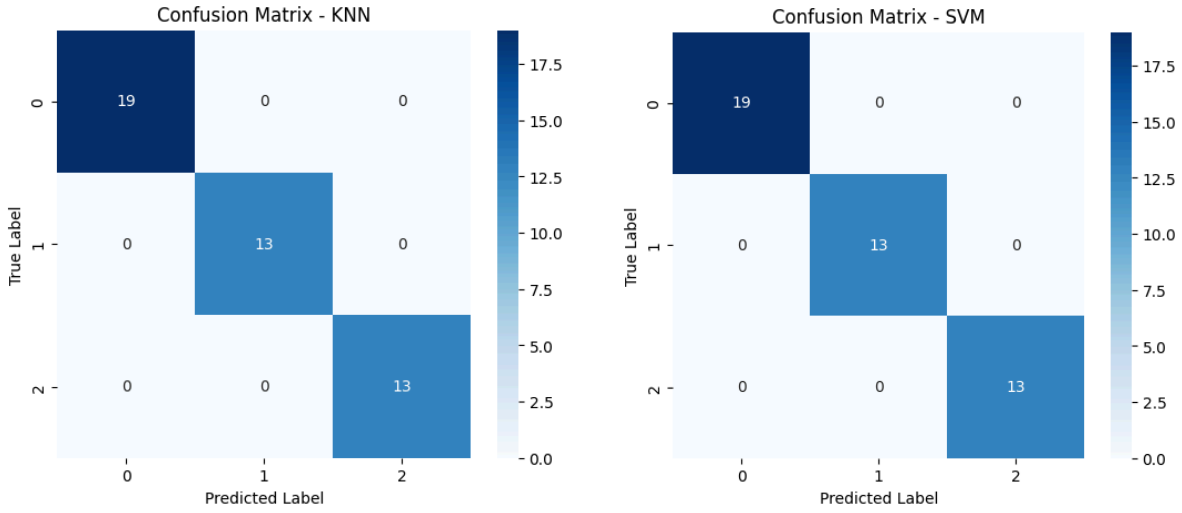
```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predictions
y_pred_knn = best_knn.predict(X_test)
# Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(confusion_matrix(y_test, y_pred_knn), annot=True, fmt='d',
cmap='Blues')
```

```
plt.title('Confusion Matrix - KNN')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```

```
# Predictions
y_pred_svm = best_svm.predict(X_test)
# Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(confusion_matrix(y_test, y_pred_svm), annot=True, fmt='d',
cmap='Blues')
plt.title('Confusion Matrix - SVM')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```

Output:



Part 4: Comparison and Conclusion

1. Model Comparison:

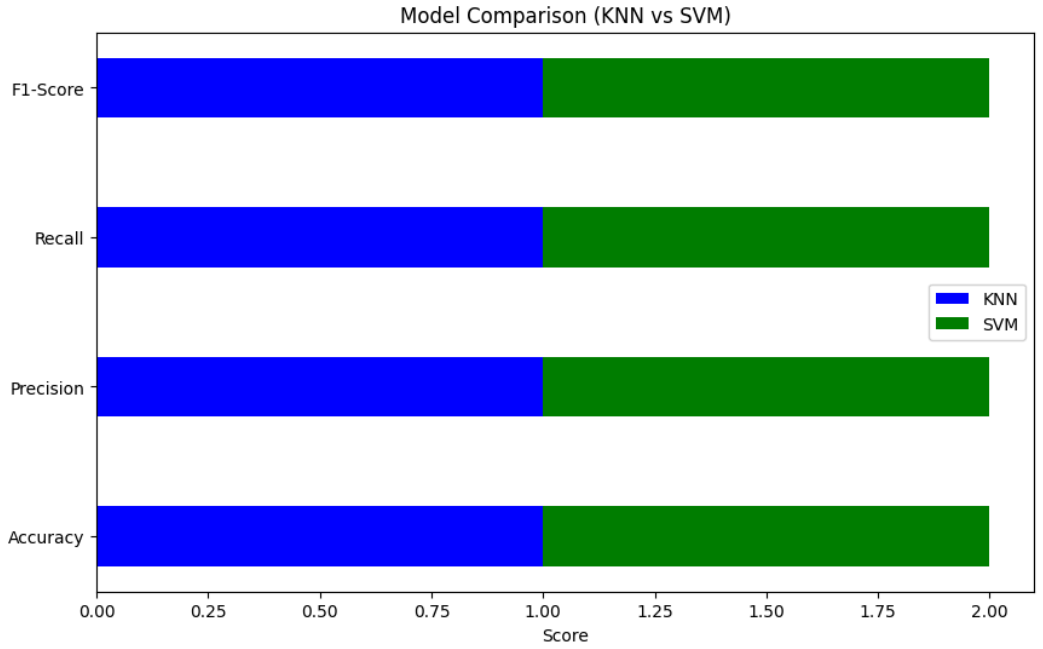
○ Compare the performance of the models using a bar chart.

○ Discuss which model performed best and why.

```
import matplotlib.pyplot as plt
# Metrics for KNN and SVM (already computed)
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
knn_scores = [accuracy_knn, precision_knn, recall_knn, f1_knn]
svm_scores = [accuracy_svm, precision_svm, recall_svm, f1_svm]
# Horizontal Bar Chart for Comparison
plt.figure(figsize=(10, 6))
# Plot KNN and SVM scores
plt.barh(metrics, knn_scores, height=0.4, label='KNN', color='blue')
```

```
plt.barh(metrics, svm_scores, height=0.4, left=knn_scores, label='SVM',
color='green')
# Adding titles and labels
plt.xlabel('Score')
plt.title('Model Comparison (KNN vs SVM)')
plt.legend()
# Display the chart
plt.show()
```

Output:



2. Future Improvements:

○ Suggest potential improvements to the models or additional features that could be used.

→ Future improvements for the models could include experimenting with different kernel functions for SVM (e.g., radial basis function or polynomial kernel) to improve performance. Additionally, hyperparameter tuning using grid search or randomized search can optimize model parameters like the number of neighbors for KNN. Feature engineering, such as adding new features or using dimensionality reduction techniques like PCA, could also enhance model accuracy. Moreover, exploring ensemble methods like Random Forest or Gradient Boosting could provide better classification results.