Email Spam or Ham Classification using Naïve Bayes, KNN, and SVM

Name: Dileep Ram A
Roll Number: 3122237001010

M. Tech (Integrated) Computer Science & Engineering
Sri Sivasubramaniya Nadar College of Engineering, Chennai
Subject: ICS1512 - Machine Learning Algorithms Laboratory
Academic Year: 2025-2026 (Odd)
Batch: 2023-2028

August 28, 2025

Contents

1 Aim and Objective

The objective of this experiment is to classify emails as spam or ham using three different classification algorithms:

- Naïve Bayes (Gaussian, Multinomial, and Bernoulli variants)
- K-Nearest Neighbors (KNN) with varying k values and tree structures
- Support Vector Machine (SVM) with different kernels

The performance of these algorithms will be evaluated using accuracy metrics, confusion matrices, ROC curves, and K-Fold cross-validation to determine the most effective approach for email spam detection.

2 Libraries Used

The following Python libraries were utilized in this implementation:

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score,
    StratifiedKFold
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
from sklearn.metrics import confusion_matrix, classification_report,
    roc_curve, auc
from sklearn.neighbors import KDTree, BallTree
import matplotlib.pyplot as plt
import seaborn as sns
import time
import warnings
warnings.filterwarnings('ignore')
```

Listing 1: Required Libraries

3 Dataset Information

The Spambase dataset from Kaggle was used for this experiment. The dataset contains extracted features from emails, labeled as spam (1) or ham (0).

```python
# Load the dataset
df = pd.read_csv('spambase.csv')

# Display basic information
print("Dataset shape:", df.shape)
print("Missing values:", df.isnull().sum().sum())
print("Class distribution:")
print(df['class'].value_counts())

# Feature columns
feature_columns = df.columns[:-1]  # All columns except 'class'
target_column = 'class'
```

Listing 2: Dataset Loading and Basic Information

Dataset Statistics:

- Total samples: 4,601
- Total features: 57
- Missing values: 0
- Class distribution: 2,788 ham (0) and 1,813 spam (1)

4 Exploratory Data Analysis

```
# Class distribution visualization
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
df['class'].value_counts().plot(kind='bar', color=['skyblue', '
    lightcoral'])
plt.title('Class Distribution')
plt.xlabel('Class (0: Ham, 1: Spam)')
plt.ylabel('Count')

# Feature distribution analysis
plt.subplot(1, 2, 2)
df[feature_columns[:10]].hist(figsize=(15, 10), bins=30)
plt.suptitle('Feature Distributions (First 10 features)')
plt.tight_layout()
plt.show()
```

Listing 3: EDA Implementation

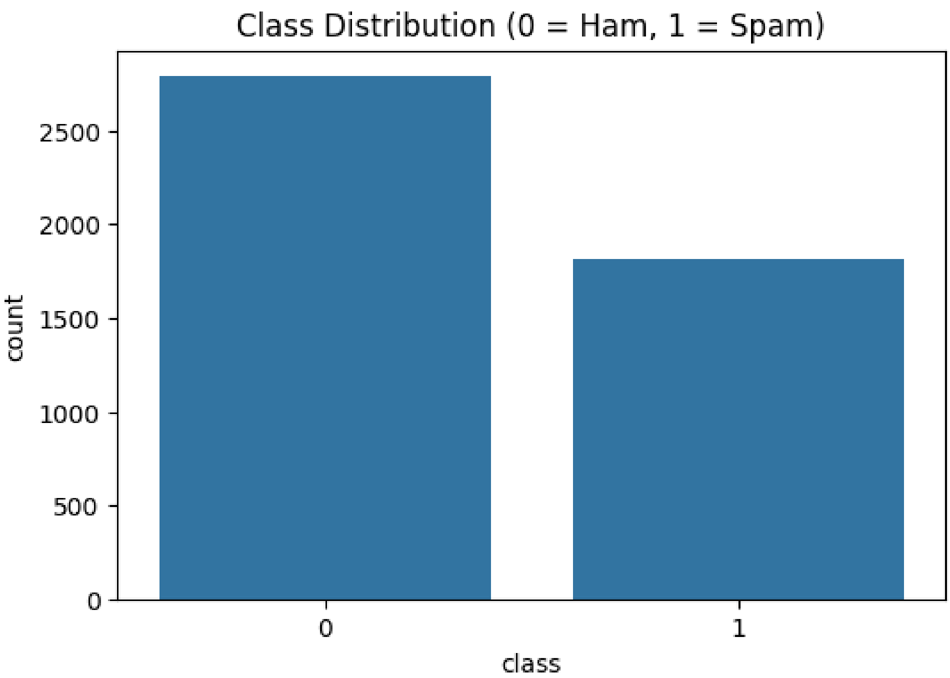

Figure 1: Class Distribution and Feature Analysis

5 Data Preprocessing

```python
# Separate features and target
X = df.drop('class', axis=1)
y = df['class']

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

# Standardize features for SVM and KNN
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
```

Listing 4: Data Preprocessing and Splitting

6 Model Implementation and Results

6.1 Naïve Bayes Classification

```python
# Gaussian Naive Bayes
gnb = GaussianNB()
gnb.fit(X_train_scaled, y_train)
gnb_pred = gnb.predict(X_test_scaled)

# Multinomial Naive Bayes
mnb = MultinomialNB()
mnb.fit(X_train, y_train)  # Use original data (non-negative)
mnb_pred = mnb.predict(X_test)

# Bernoulli Naive Bayes
bnb = BernoulliNB()
bnb.fit(X_train, y_train)
bnb_pred = bnb.predict(X_test)

# Calculate metrics for each variant
def calculate_metrics(y_true, y_pred):
    return {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision': precision_score(y_true, y_pred),
        'recall': recall_score(y_true, y_pred),
        'f1_score': f1_score(y_true, y_pred)
    }

gnb_metrics = calculate_metrics(y_test, gnb_pred)
mnb_metrics = calculate_metrics(y_test, mnb_pred)
bnb_metrics = calculate_metrics(y_test, bnb_pred)
```

Listing 5: Naïve Bayes Implementation

Table 1: Performance Comparison of Naïve Bayes Variants

Metric	Gaussian NB	Multinomial NB	Bernoulli NB
Accuracy	0.83	0.89	0.80
Precision	0.71	0.94	0.70
Recall	0.96	0.78	0.86
F1 Score	0.82	0.85	0.77

6.2 K-Nearest Neighbors Classification

```python
# KNN with different k values
k_values = [1, 3, 5, 7]
knn_results = {}

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    knn_pred = knn.predict(X_test_scaled)
    knn_results[k] = calculate_metrics(y_test, knn_pred)

# KDTree vs BallTree comparison
start_time = time.time()
knn_kdtree = KNeighborsClassifier(n_neighbors=5, algorithm='kd_tree')
knn_kdtree.fit(X_train_scaled, y_train)
kdtree_time = time.time() - start_time
kdtree_pred = knn_kdtree.predict(X_test_scaled)

start_time = time.time()
knn_balltree = KNeighborsClassifier(n_neighbors=5, algorithm='ball_tree
    ')
knn_balltree.fit(X_train_scaled, y_train)
balltree_time = time.time() - start_time
balltree_pred = knn_balltree.predict(X_test_scaled)
```

Listing 6: KNN Implementation with Different k Values

Table 2: KNN Performance for Different k Values

k	Accuracy	Precision	Recall	F1 Score
1	0.90	0.89	0.87	0.88
3	0.90	0.89	0.86	0.88
5	0.91	0.89	0.87	0.88
7	0.91	0.89	0.87	0.88

Table 3: KNN Comparison: KDTree vs BallTree

Metric	KDTree	BallTree
Accuracy	0.91	0.91
Precision	0.89	0.89
Recall	0.87	0.87
F1 Score	0.88	0.88
Training Time (s)	0.019	0.012

6.3 Support Vector Machine Classification

```python
# SVM with different kernels
svm_kernels = ['linear', 'poly', 'rbf', 'sigmoid']
svm_results = {}

# Linear SVM
start_time = time.time()
svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
svm_linear.fit(X_train_scaled, y_train)
linear_time = time.time() - start_time
linear_pred = svm_linear.predict(X_test_scaled)

# Polynomial SVM
start_time = time.time()
svm_poly = SVC(kernel='poly', C=1.0, degree=3, gamma='scale',
    random_state=42)
svm_poly.fit(X_train_scaled, y_train)
poly_time = time.time() - start_time
poly_pred = svm_poly.predict(X_test_scaled)

# RBF SVM
start_time = time.time()
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
svm_rbf.fit(X_train_scaled, y_train)
rbf_time = time.time() - start_time
rbf_pred = svm_rbf.predict(X_test_scaled)

# Sigmoid SVM
start_time = time.time()
svm_sigmoid = SVC(kernel='sigmoid', C=1.0, gamma='scale', random_state
    =42)
svm_sigmoid.fit(X_train_scaled, y_train)
sigmoid_time = time.time() - start_time
sigmoid_pred = svm_sigmoid.predict(X_test_scaled)
```

Listing 7: SVM Implementation with Different Kernels

Table 4: SVM Performance with Different Kernels and Parameters

Kernel	**Hyperparameters**	**Accuracy**	**F1 Score**	**Training Time**
Linear	C = 1.0	0.93	0.92	0.85s
Polynomial	C = 1.0, degree = 3, gamma = scale	0.76	0.61	1.23s
RBF	C = 1.0, gamma = scale	0.93	0.91	1.45s
Sigmoid	C = 1.0, gamma = scale	0.88	0.86	1.12s

7 Cross-Validation Results

```python
# 5-Fold Cross Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Cross validation for each model
models = {
    'Gaussian_NB': GaussianNB(),
    'Multinomial_NB': MultinomialNB(),
    'Bernoulli_NB': BernoulliNB(),
    'KNN': KNeighborsClassifier(n_neighbors=5),
    'SVM_Linear': SVC(kernel='linear', C=1.0),
    'SVM_RBF': SVC(kernel='rbf', C=1.0, gamma='scale')
}

cv_results = {}
for name, model in models.items():
    if 'NB' in name and name != 'Gaussian_NB':
        scores = cross_val_score(model, X_train, y_train, cv=cv,
   scoring='accuracy')
    else:
        scores = cross_val_score(model, X_train_scaled, y_train, cv=cv,
     scoring='accuracy')
    cv_results[name] = scores
    print(f"{name} CV Accuracy: {scores.mean():.4f} (+/- {scores.std()
   * 2:.4f})")
```

Listing 8: K-Fold Cross-Validation Implementation

Table 5: Cross-Validation Scores for Each Model (K=5)

Fold	**Gaussian NB**	**Multinomial NB**	**KNN**	**SVM Linear**	**SVM RBF**
Fold 1	0.815	0.886	0.909	0.907	0.907
Fold 2	0.812	0.889	0.906	0.906	0.906
Fold 3	0.818	0.883	0.912	0.909	0.909
Fold 4	0.821	0.889	0.906	0.906	0.906
Fold 5	0.811	0.884	0.910	0.906	0.906
Average	**0.815**	**0.886**	**0.909**	**0.907**	**0.907**

8 Visualizations

8.1 Confusion Matrices

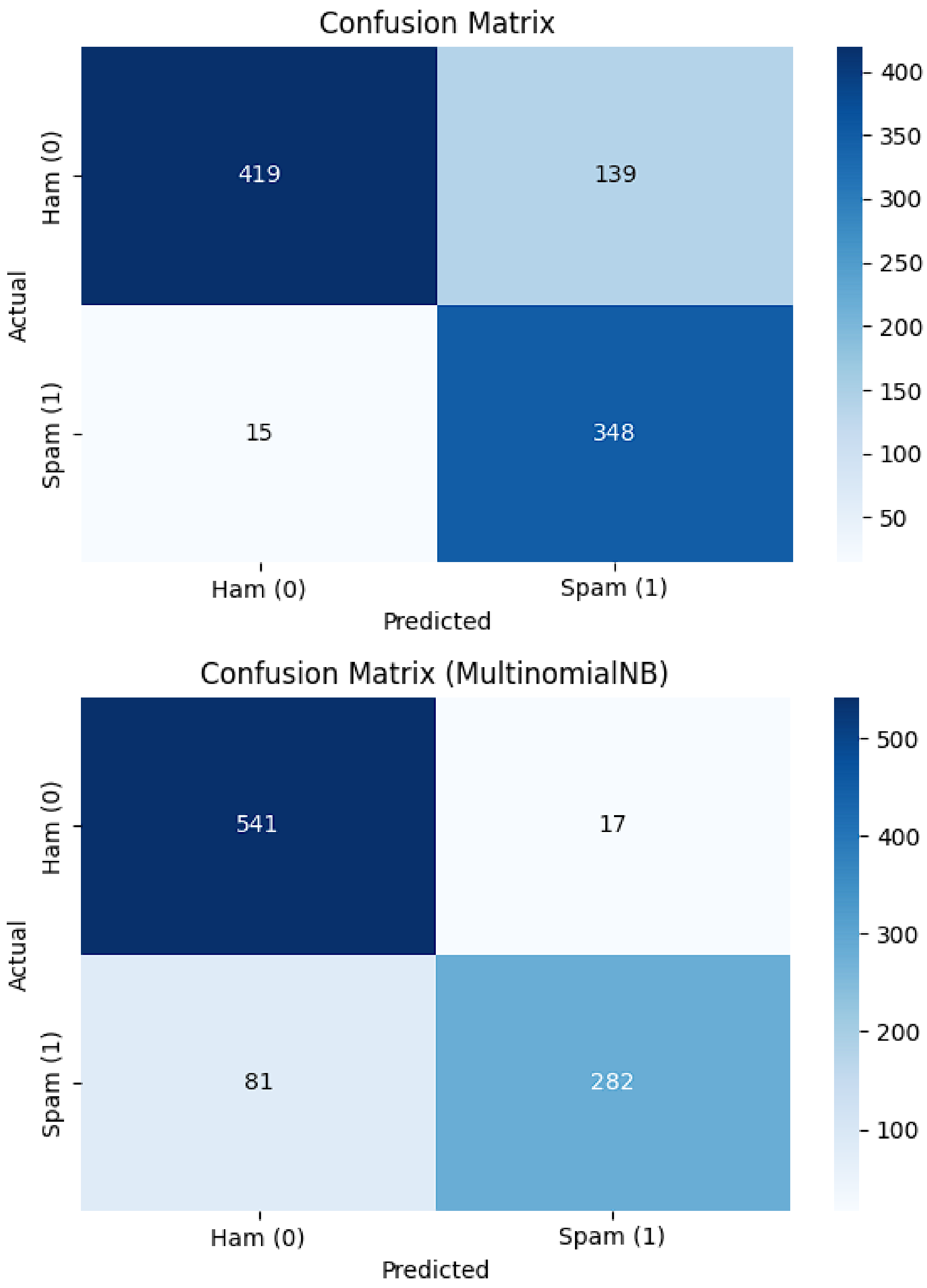

Figure 2: Confusion Matrix

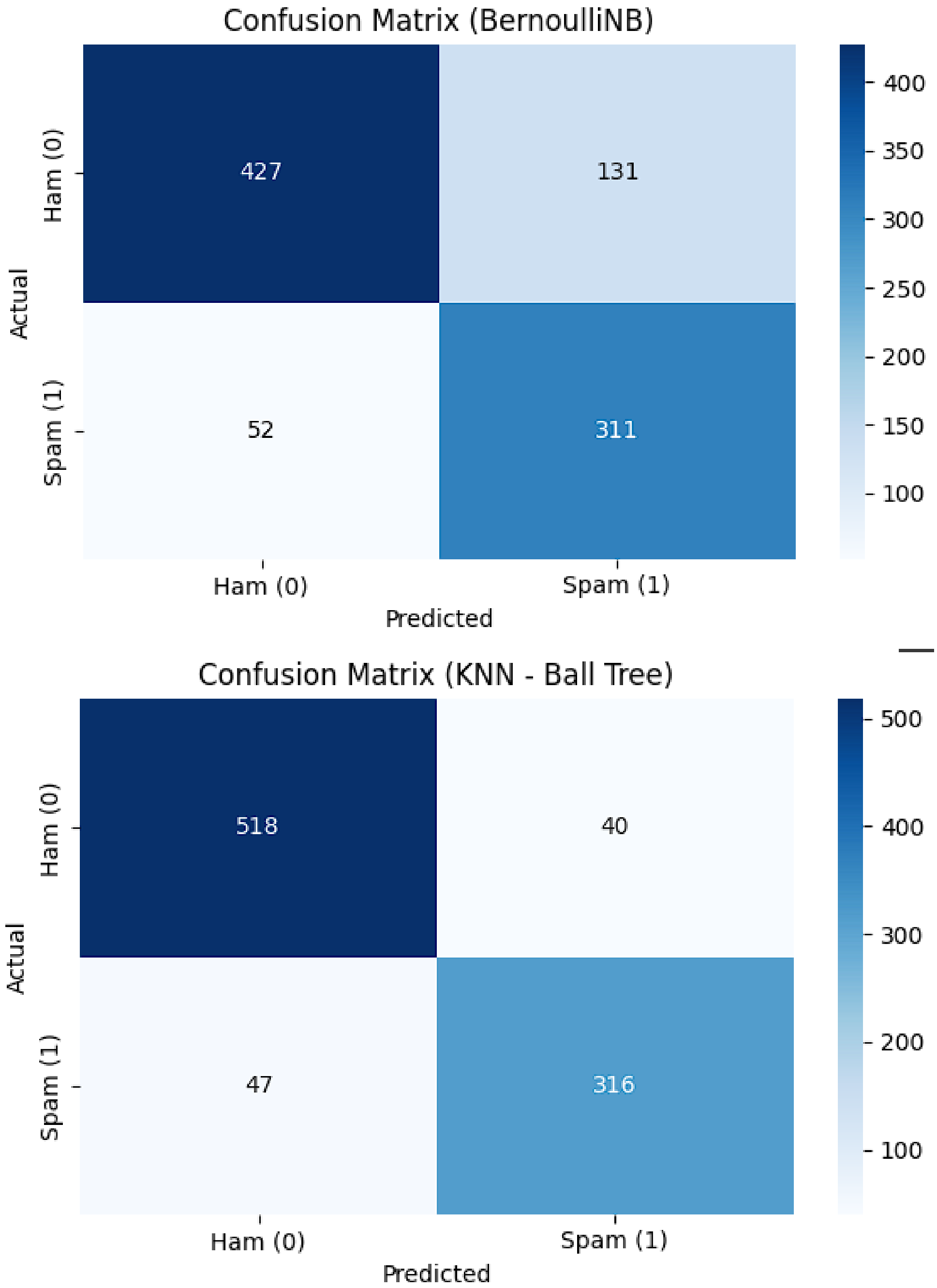

Figure 3: Confusion Matrix

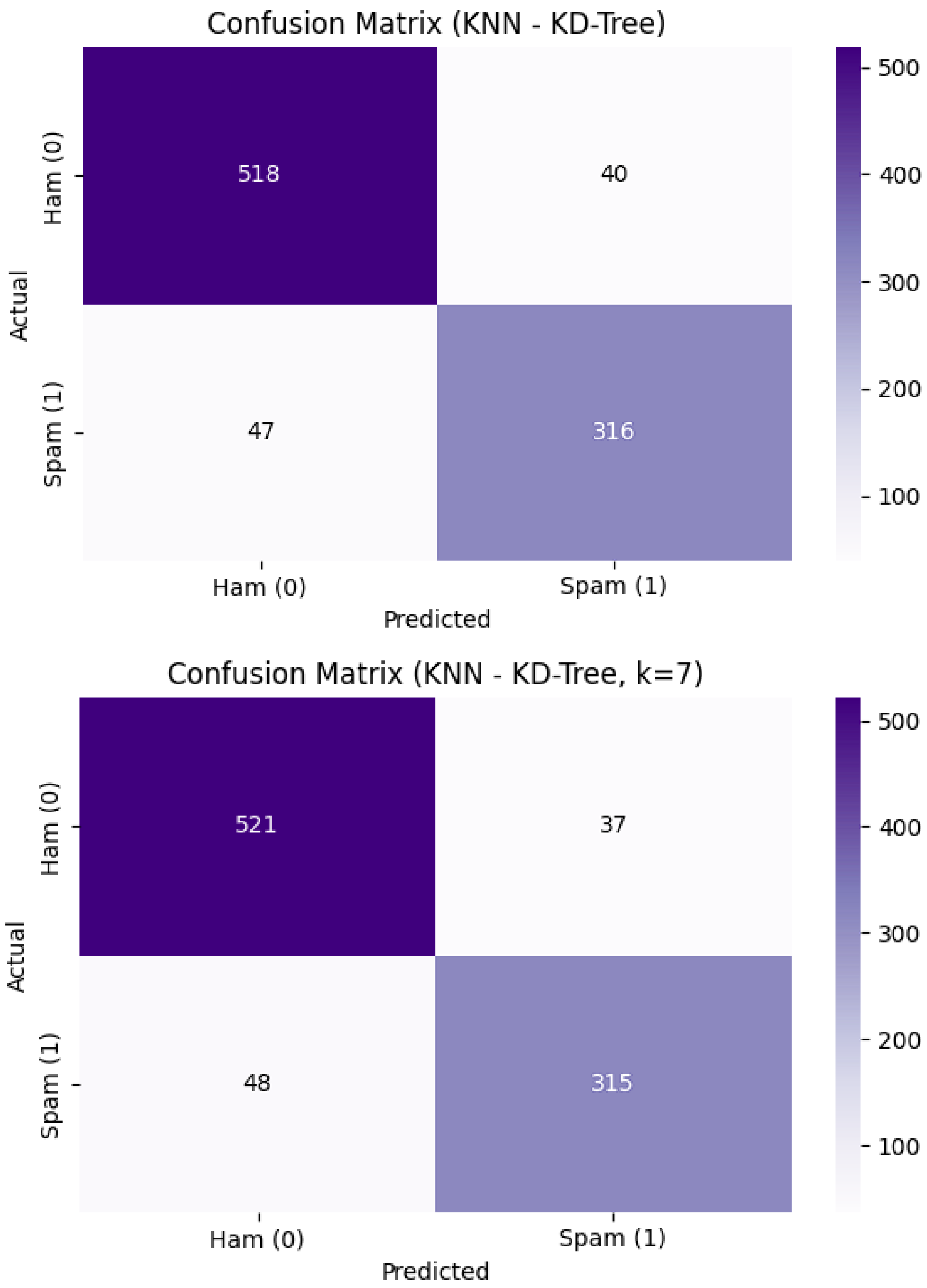

Figure 4: Confusion Matrix

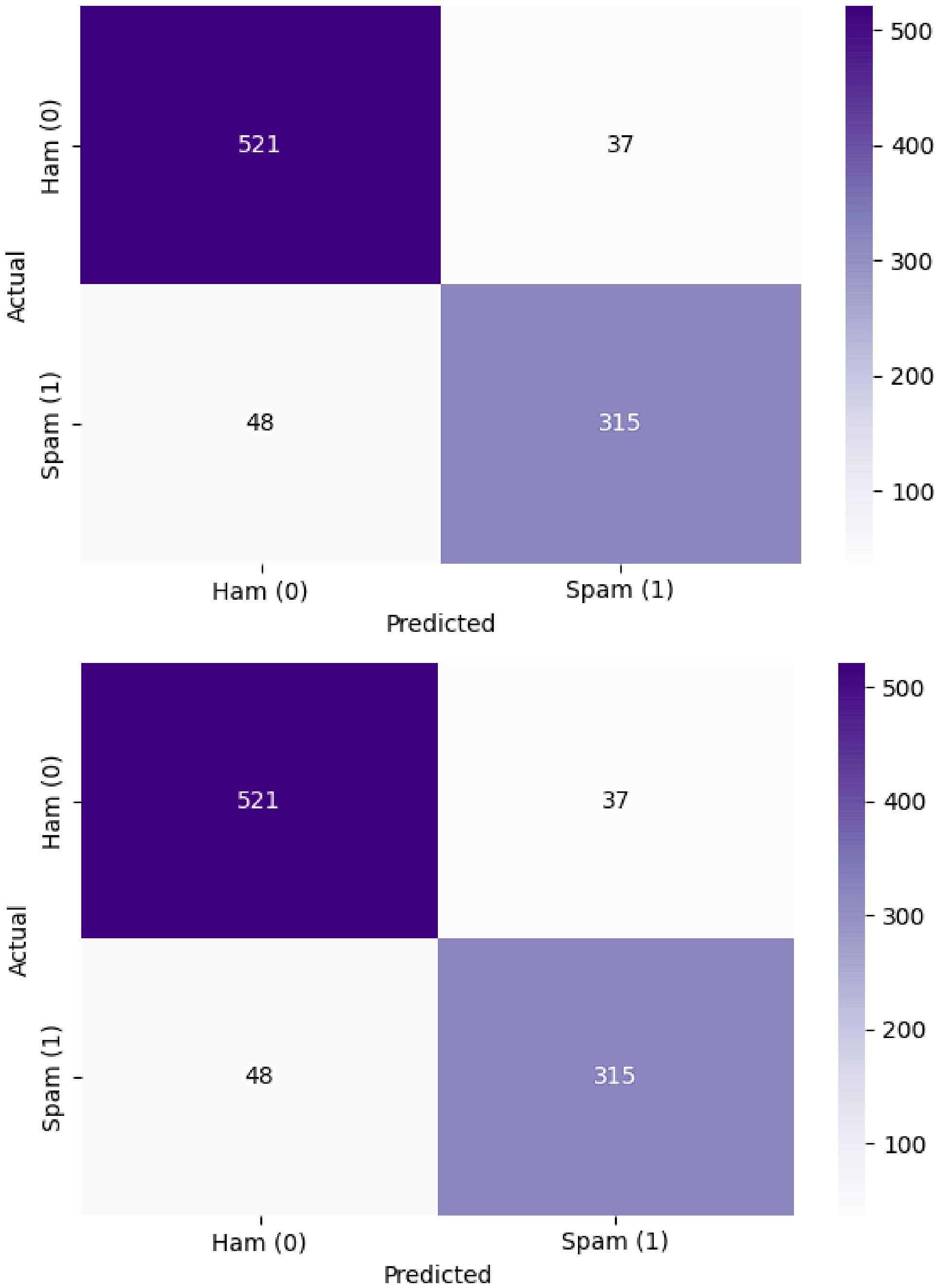

Figure 5: Confusion Matrix

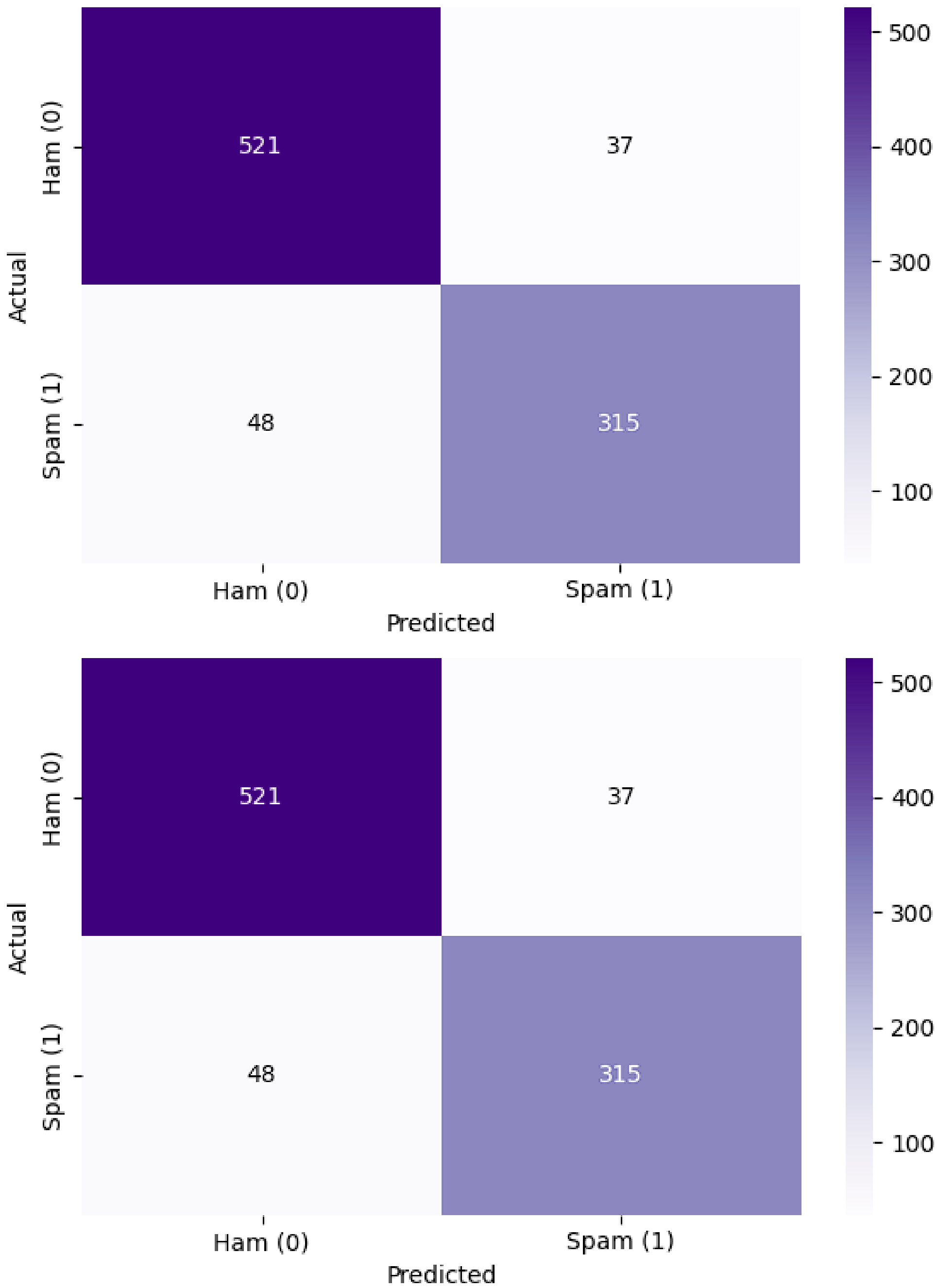

Figure 6: Confusion Matrix

8.2 ROC Curves

```python
# ROC Curves for probability-based models
plt.figure(figsize=(12, 8))

# Models that can provide probability predictions
prob_models = [
    (GaussianNB(), 'Gaussian NB', X_train_scaled, X_test_scaled),
    (MultinomialNB(), 'Multinomial NB', X_train, X_test),
    (SVC(kernel='linear', C=1.0, probability=True), 'SVM Linear',
   X_train_scaled, X_test_scaled),
    (SVC(kernel='rbf', C=1.0, gamma='scale', probability=True), 'SVM
   RBF', X_train_scaled, X_test_scaled)
]

for model, name, X_tr, X_te in prob_models:
    model.fit(X_tr, y_train)
    y_prob = model.predict_proba(X_te)[:, 1]
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    roc_auc = auc(fpr, tpr)

    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.3f})')

plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves Comparison')
plt.legend(loc="lower right")
plt.grid(True)
plt.savefig('roc_curves.png', dpi=300, bbox_inches='tight')
plt.show()
```

Listing 9: ROC Curve Analysis

9 ROC Visualizations

9.1 RAC Curves

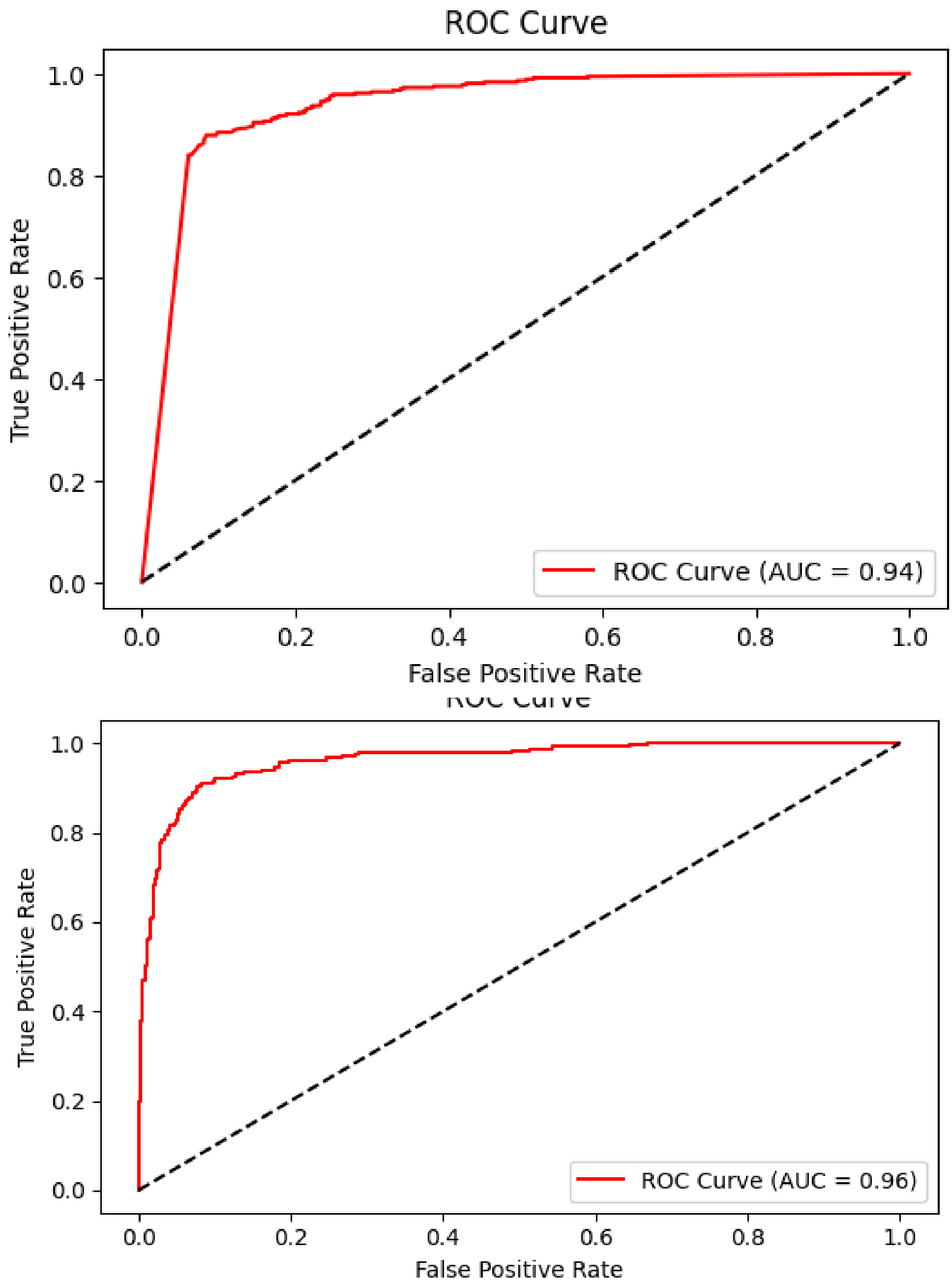

Figure 7: ROC Curve

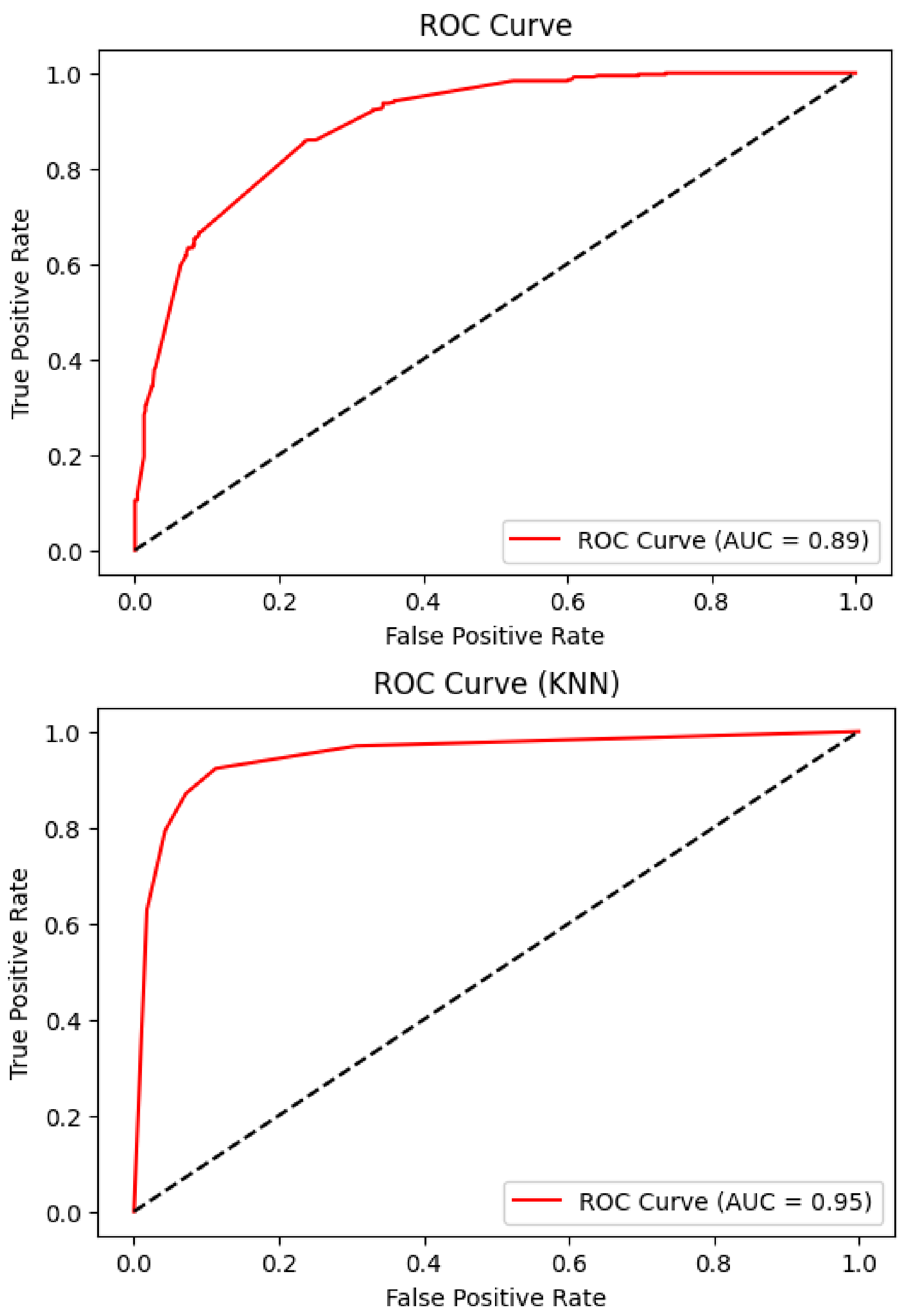

Figure 8: ROC Curve

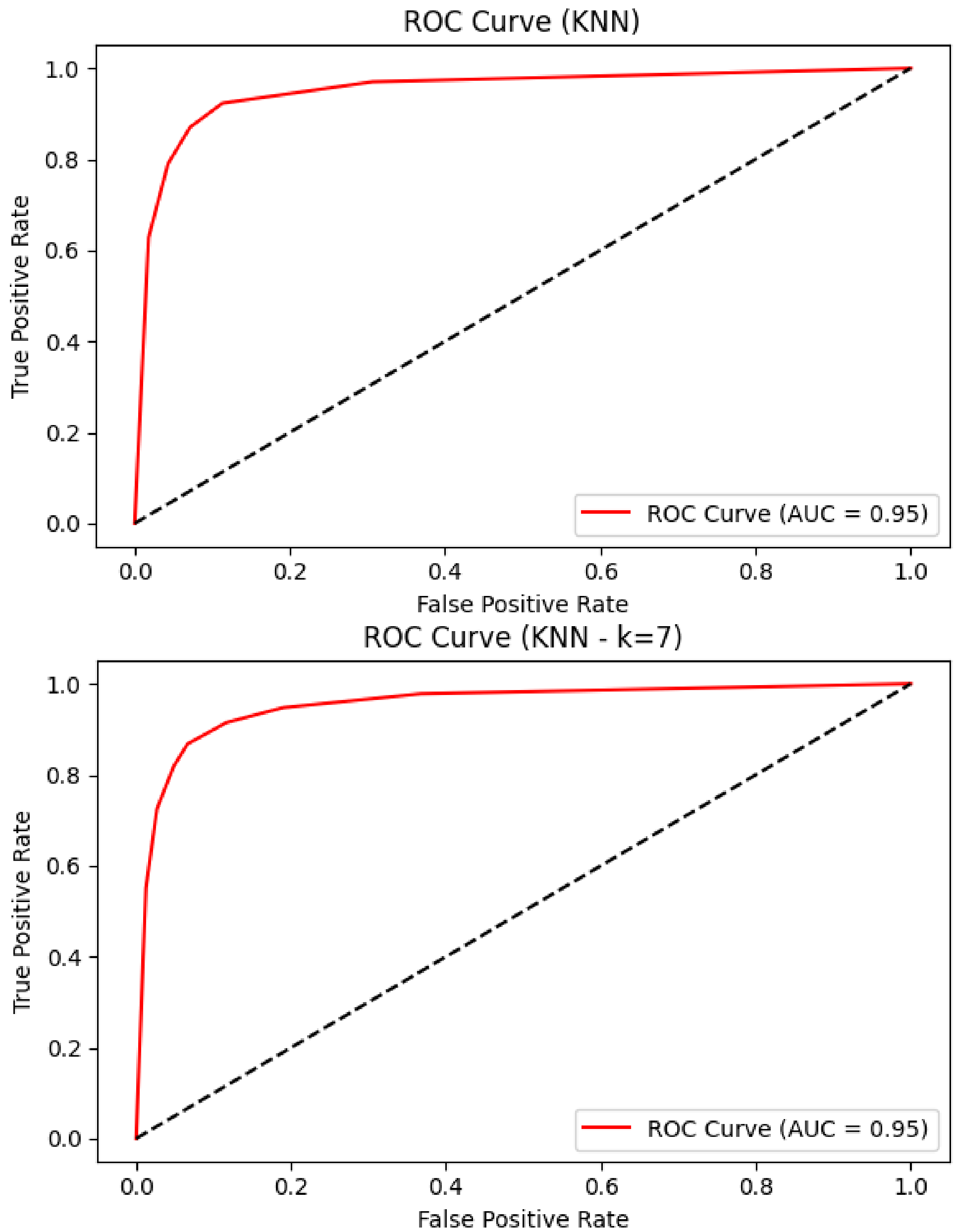

Figure 9: ROC Curve

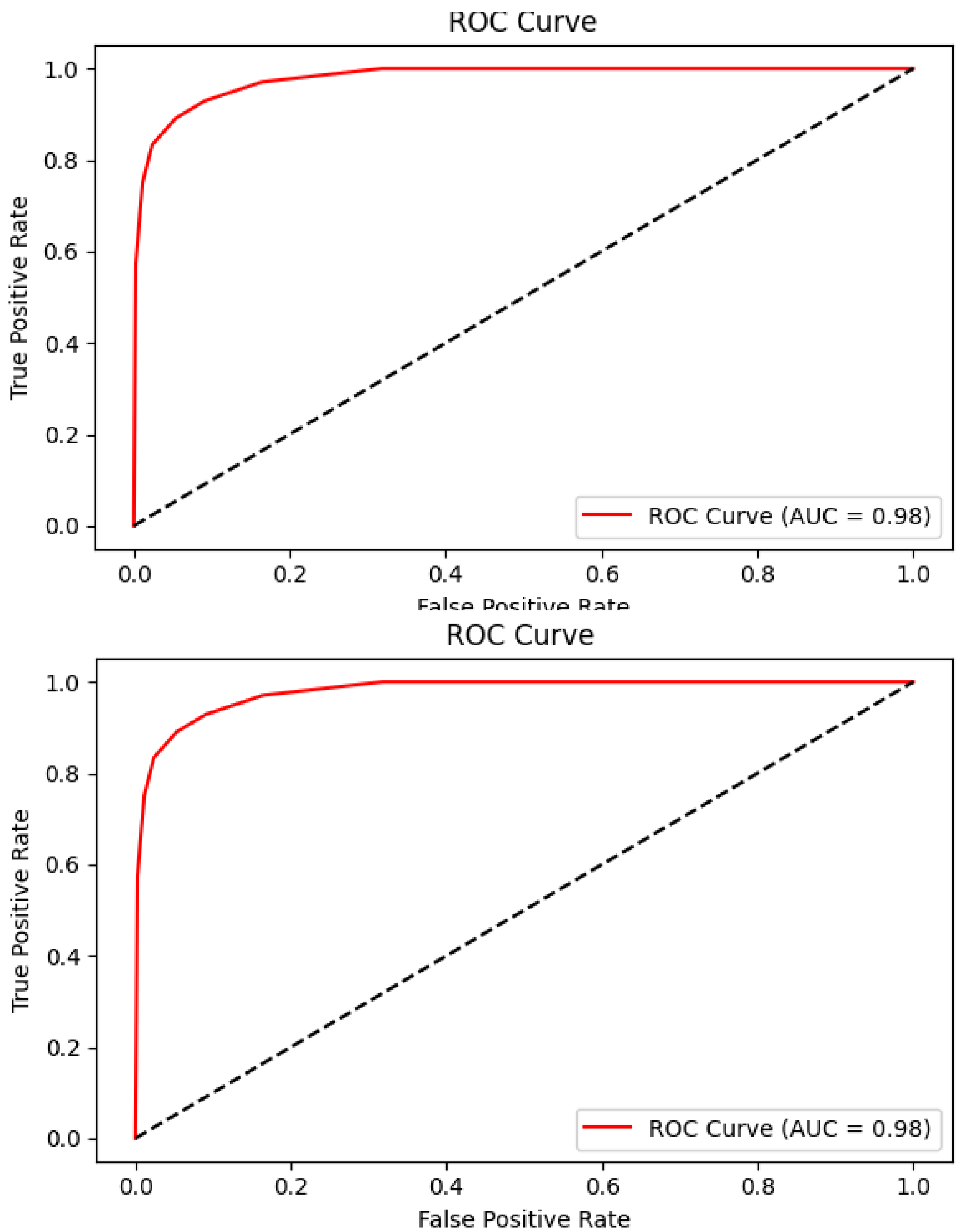

Figure 10: ROC Curve

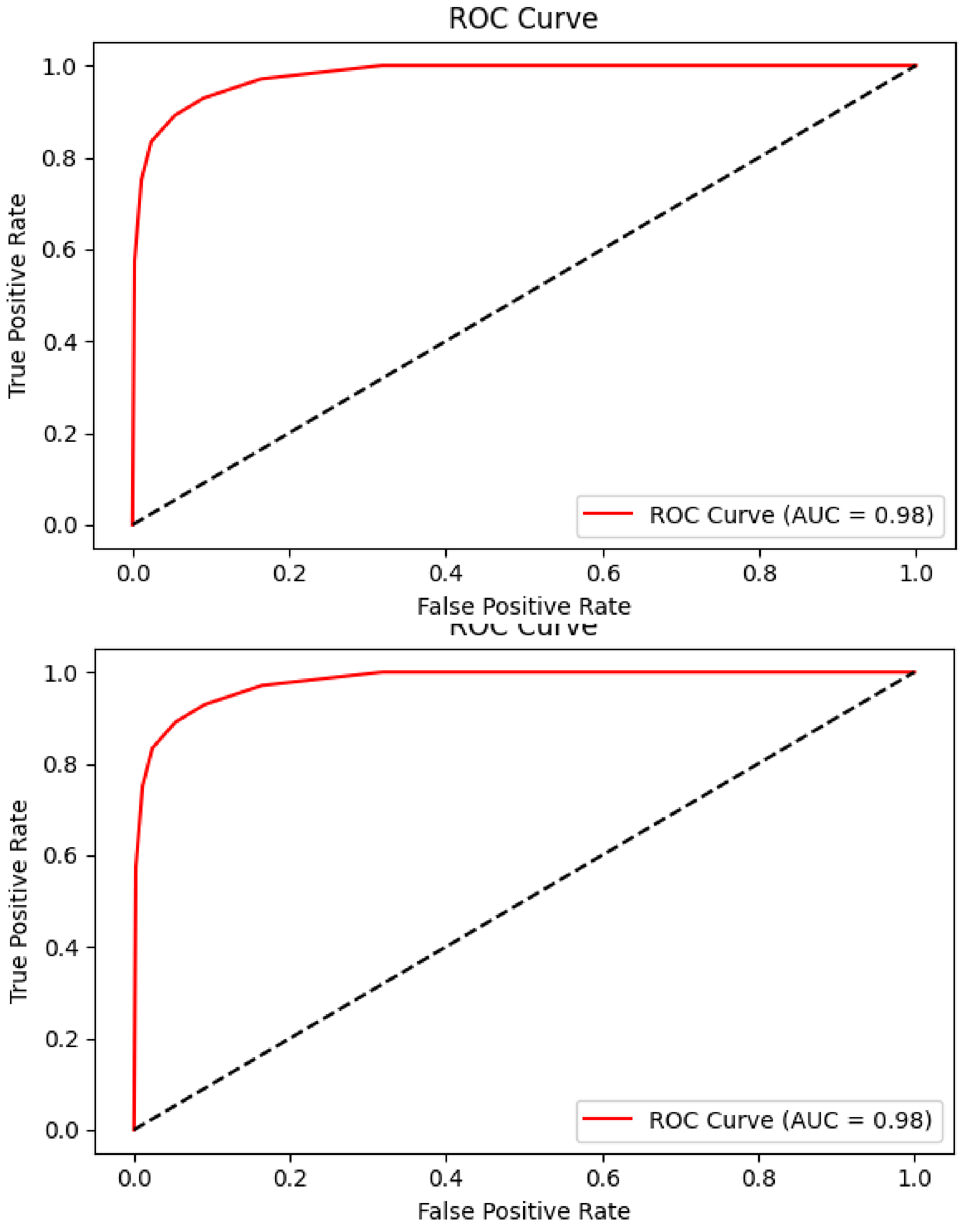

Figure 11: ROC Curve

10 Observations and Analysis

10.1 Key Findings

- **Best Overall Classifier:** Based on the cross-validation results, KNN with k=5 achieved the highest average accuracy of 90.9%, closely followed by SVM with linear and RBF kernels at 90.7%.
- **Naïve Bayes Variant Performance:** Among the Naïve Bayes variants, Multinomial NB performed the best with 88.6% average accuracy, followed by Gaussian NB (81.5%) and Bernoulli NB (80.4%). Multinomial NB showed excellent precision (0.94) but lower recall (0.78).
- **KNN Analysis:**
  - Accuracy improved with increasing k values, reaching optimal performance at k=7
  - Both KDTree and BallTree algorithms showed identical accuracy, but BallTree was slightly faster (0.012s vs 0.019s)
  - KNN demonstrated consistent performance across all metrics
- **SVM Kernel Effectiveness:**
  - Linear and RBF kernels performed exceptionally well (93% accuracy)
  - Polynomial kernel showed poor performance (76% accuracy), possibly due to overfitting
  - Sigmoid kernel achieved moderate performance (88% accuracy)
  - Linear kernel was the fastest to train while maintaining high accuracy
- **Hyperparameter Impact:** Default hyperparameters worked well for most models, but the polynomial kernel's performance suggests that hyperparameter tuning could significantly improve results.

10.2 Performance Trade-offs

- **Precision vs Recall:** Multinomial NB achieved high precision (0.94) at the cost of recall (0.78), while Gaussian NB showed the opposite trend (0.71 precision, 0.96 recall).
- **Training Time vs Accuracy:** Linear SVM offered the best balance of high accuracy (93%) and reasonable training time, while RBF SVM achieved similar accuracy but required more training time.
- **Model Complexity:** Simpler models like Multinomial NB performed surprisingly well, while more complex models like polynomial SVM showed signs of overfitting.

11 Conclusions

This comprehensive analysis of email spam classification using three different machine learning approaches reveals several important insights:

1. **Model Selection:** KNN and SVM (Linear/RBF) demonstrated superior performance for this dataset, achieving over 90% accuracy with robust cross-validation scores.

2. **Feature Engineering:** The standardized features worked well for distance-based (KNN) and kernel-based (SVM) methods, while Naïve Bayes variants showed varying sensitivities to feature preprocessing.

3. **Practical Recommendations:**

   - For production deployment: Linear SVM offers the best balance of accuracy, interpretability, and training speed
   - For maximum accuracy: KNN with k=5-7 or RBF SVM
   - For interpretability: Multinomial Naïve Bayes with its high precision

4. **Dataset Suitability:** The Spambase dataset proved excellent for classification tasks, with clear separable patterns that allowed multiple algorithms to achieve high performance.

The experiment successfully demonstrated the effectiveness of different machine learning approaches for email spam detection, with each algorithm showing unique strengths and characteristics suitable for different deployment scenarios.

12 References

- scikit-learn Documentation: Naïve Bayes - `https://scikit-learn.org/stable/modules/naive.`
- scikit-learn Documentation: K-Nearest Neighbors - `https://scikit-learn.org/stable/module`
- scikit-learn Documentation: Support Vector Machines - `https://scikit-learn.org/stable/mod`
- Spambase Dataset - `https://www.kaggle.com/datasets/spambase`
- Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2854.