Experiment 3: Email Spam or Ham Classification using Naïve Bayes and KNN

G S SREENETHI
Reg No: 3122237001052

Aim

To develop machine learning models using Naïve Bayes (Gaussian, Multinomial, Bernoulli) and K-Nearest Neighbors (KNN with varying k, *kd_tree*, and *ball_tree*) for classifying emails as spam or ham, and evaluate the models using accuracy metrics, confusion matrix, ROC curves, and 5-fold cross-validation.

Objective

- Build models using Naïve Bayes and KNN to classify spam emails.
- Perform EDA to understand feature relevance.
- Train and evaluate models using confusion matrix, accuracy, precision, recall, and F1 score.
- Visualize results using ROC curves.
- Use 5-fold cross-validation to assess model robustness.
- Compare models to determine the best-performing classifier.

Libraries Used

- `pandas, numpy`
- `matplotlib.pyplot, seaborn`
- `sklearn.model_selection (train_test_split, KFold, cross_val_score)`
- `sklearn.preprocessing (MinMaxScaler)`
- `sklearn.naive_bayes (GaussianNB, MultinomialNB, BernoulliNB)`
- `sklearn.neighbors (KNeighborsClassifier)`

- sklearn.metrics (accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, RocCurveDisplay, ConfusionMatrixDisplay)

Code section

```
# ---------------------
# 1. Imports and Dataset
# ---------------------
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix, RocCurveDisplay,
                             ConfusionMatrixDisplay)

# Load dataset
df = pd.read_csv('/content/spambase_csv.csv')

# Add column names
columns = ['word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d',
           'word_freq_our', 'word_freq_over', 'word_freq_remove', 'word_freq_internet',
           'word_freq_order', 'word_freq_mail', 'word_freq_receive', 'word_freq_will',
           'word_freq_people', 'word_freq_report', 'word_freq_addresses', 'word_freq_fre
           'word_freq_business', 'word_freq_email', 'word_freq_you', 'word_freq_credit',
           'word_freq_your', 'word_freq_font', 'word_freq_000', 'word_freq_money',
           'word_freq_hp', 'word_freq_hpl', 'word_freq_george', 'word_freq_650',
           'word_freq_lab', 'word_freq_labs', 'word_freq_telnet', 'word_freq_857',
           'word_freq_data', 'word_freq_415', 'word_freq_85', 'word_freq_technology',
           'word_freq_1999', 'word_freq_parts', 'word_freq_pm', 'word_freq_direct',
           'word_freq_cs', 'word_freq_meeting', 'word_freq_original', 'word_freq_project
           'word_freq_re', 'word_freq_edu', 'word_freq_table', 'word_freq_conference',
           'char_freq_;', 'char_freq_(', 'char_freq_[', 'char_freq_!', 'char_freq_$',
           'char_freq_#', 'capital_run_length_average', 'capital_run_length_longest',
           'capital_run_length_total', 'spam']
df.columns = columns

# ---------------------
# 2. EDA Visualizations
# ---------------------
```

```
# Class distribution
sns.countplot(x='spam', data=df)
plt.title('Class Distribution (0=Ham, 1=Spam)')
plt.show()

# Top correlated features
corr = df.corr()['spam'].sort_values(ascending=False)[1:11]
sns.barplot(x=corr.values, y=corr.index)
plt.title('Top 10 Features Correlated with Spam')
plt.show()

# Top 3 feature distributions
top_features = df.corr()['spam'].abs().sort_values(ascending=False)[1:4].index
for feature in top_features:
    sns.boxplot(x='spam', y=feature, data=df)
    plt.title(f'Distribution of {feature}')
    plt.show()

# ---------------------
# 3. Data Preprocessing
# ---------------------
X = df.drop('spam', axis=1)
y = df['spam']

# Normalize features
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# 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. Model Training and Evaluation
# ---------------------
def evaluate_model(model, model_name):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Calculate metrics
    metrics = {
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
```

```
        'F1': f1_score(y_test, y_pred)
    }

    # Print results
    print(f"\n{model_name} Results:")
    for metric, value in metrics.items():
        print(f"{metric}: {value:.4f}")

    # Confusion Matrix
    ConfusionMatrixDisplay.from_estimator(model, X_test, y_test)
    plt.title(f'{model_name} Confusion Matrix')
    plt.show()

    # ROC Curve
    if hasattr(model, "predict_proba"):
        RocCurveDisplay.from_estimator(model, X_test, y_test)
        plt.title(f'{model_name} ROC Curve')
        plt.show()

    return metrics

# ---------------------
# 5. Naive Bayes Models
# ---------------------
for name, model in [('Gaussian NB', GaussianNB()),
                    ('Multinomial NB', MultinomialNB()),
                    ('Bernoulli NB', BernoulliNB())]:
    evaluate_model(model, name)

# ---------------------
# 6. KNN Models
# ---------------------
for k in [1, 3, 5, 7]:
    evaluate_model(KNeighborsClassifier(n_neighbors=k), f'KNN (k={k})')

# KNN Tree Algorithms
for algo in ['kd_tree', 'ball_tree']:
    evaluate_model(KNeighborsClassifier(n_neighbors=5, algorithm=algo), f'KNN ({algo})')

# ---------------------
# 7. Cross Validation
# ---------------------
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
for name, model in [('Gaussian NB', GaussianNB()),
                    ('KNN (k=5)', KNeighborsClassifier(n_neighbors=5))]:
```

```
scores = cross_val_score(model, X, y, cv=kfold, scoring='accuracy')
print(f"\n{name}:")
print(f"Mean Accuracy: {scores.mean():.4f}")
print(f"Std Dev: {scores.std():.4f}")
```

EDA Visualizations

1) Class Distribution

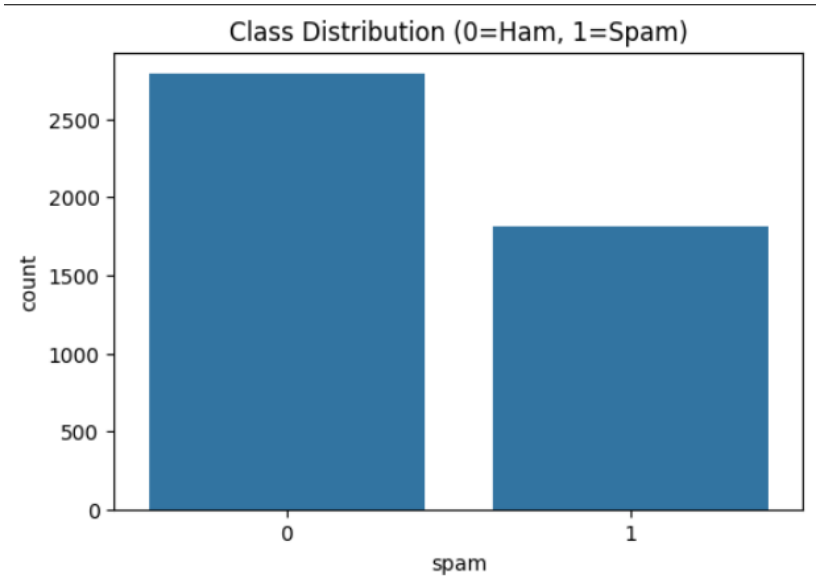

Figure 1: Distribution of Spam and Ham Emails

2) Top Correlated Features

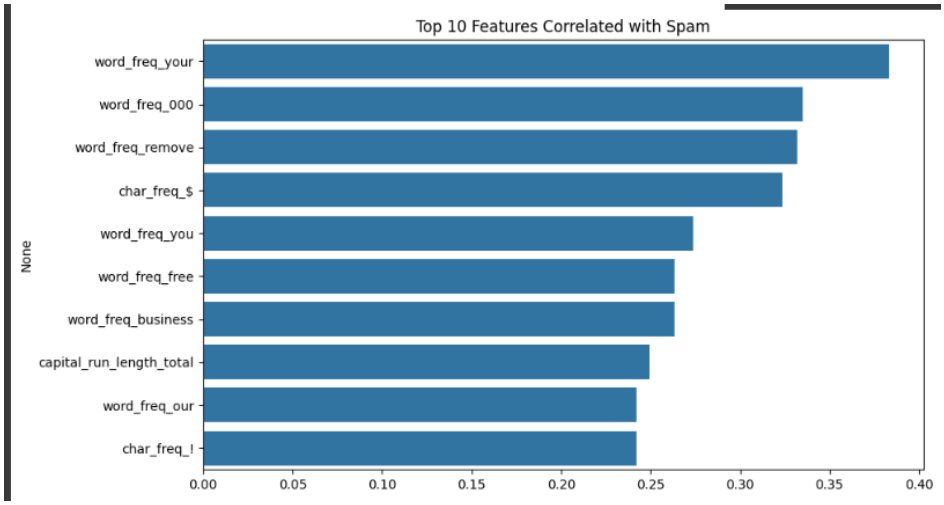

Figure 2: Top 10 Features Correlated with Spam

3) Top 3 Feature Distributions

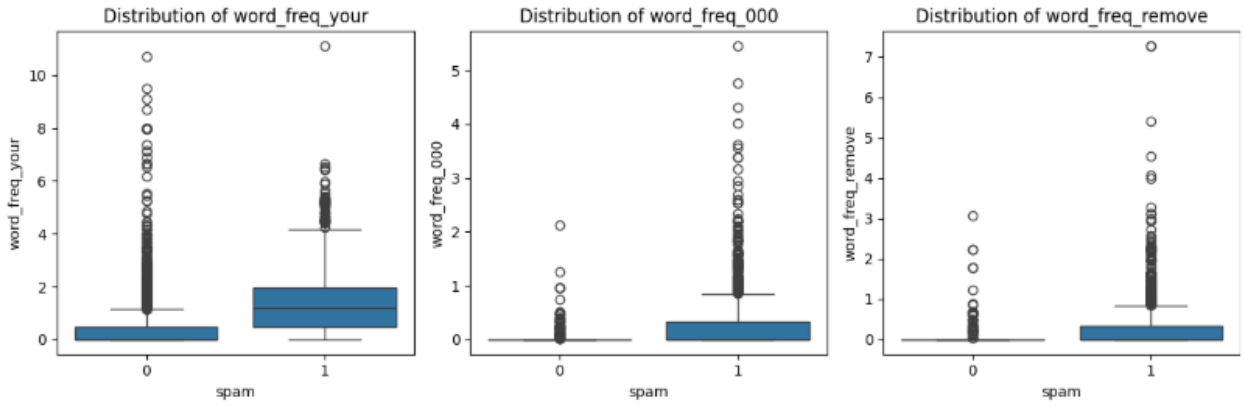

Figure 3: Distribution of Top 3 Features w.r.t Spam/Ham

Model Evaluation Results

Naïve Bayes Variants

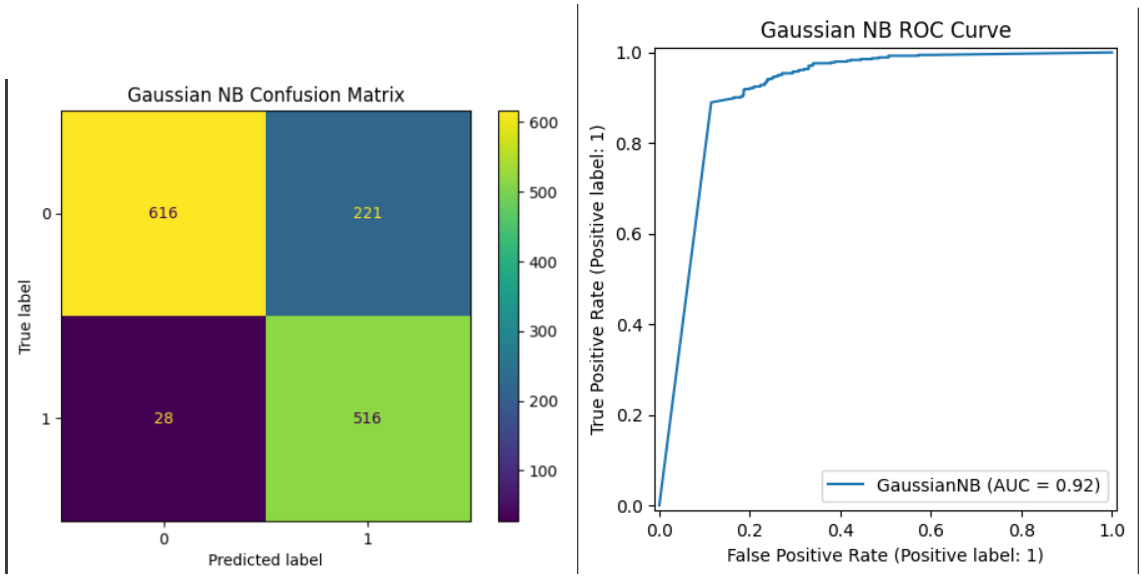

Figure 4: GaussianNB: Confusion Matrix and ROC

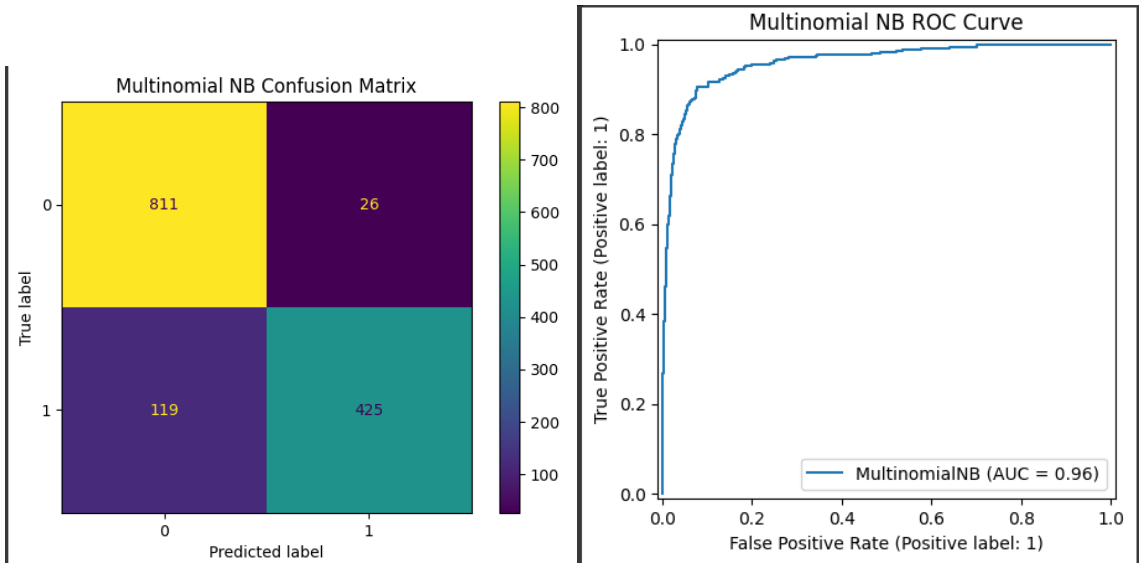

Figure 5: MultinomialNB: Confusion Matrix and ROC

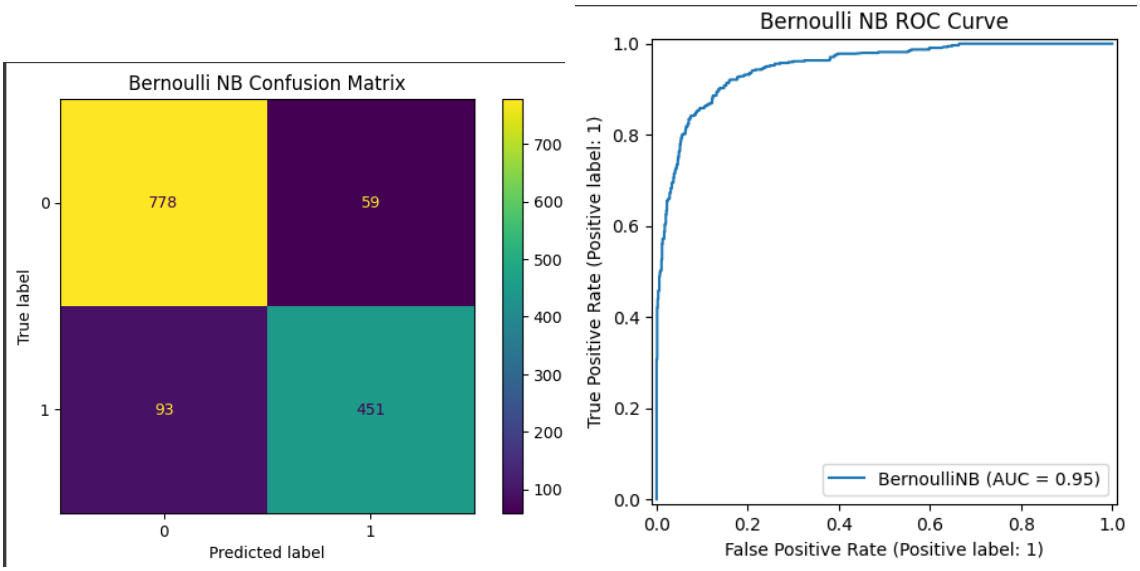

Figure 6: BernoulliNB: Confusion Matrix and ROC

KNN Models

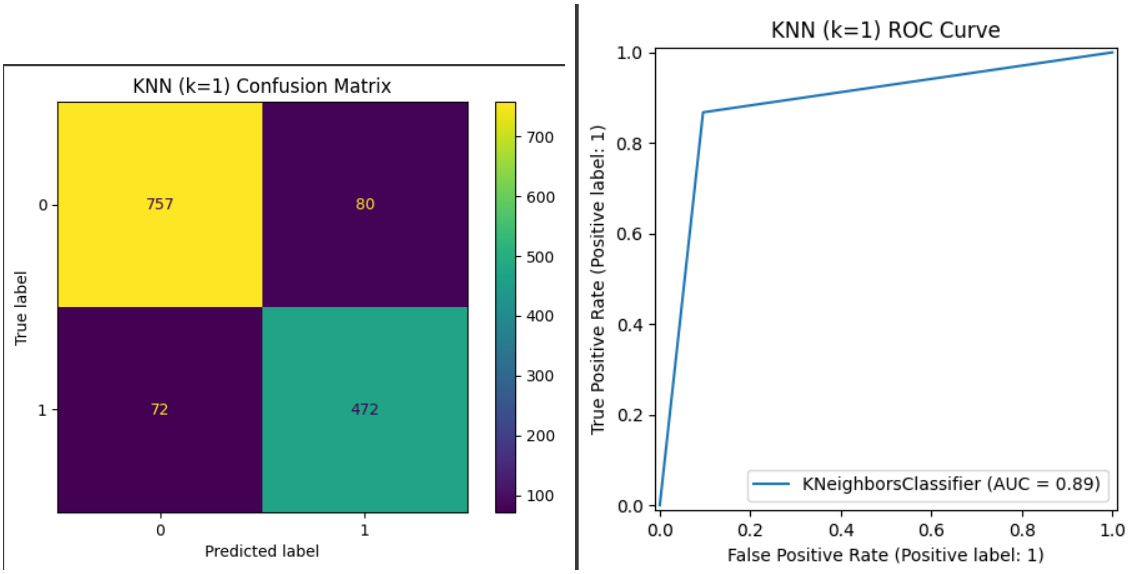

Figure 7: KNN (k=1): Confusion Matrix and ROC

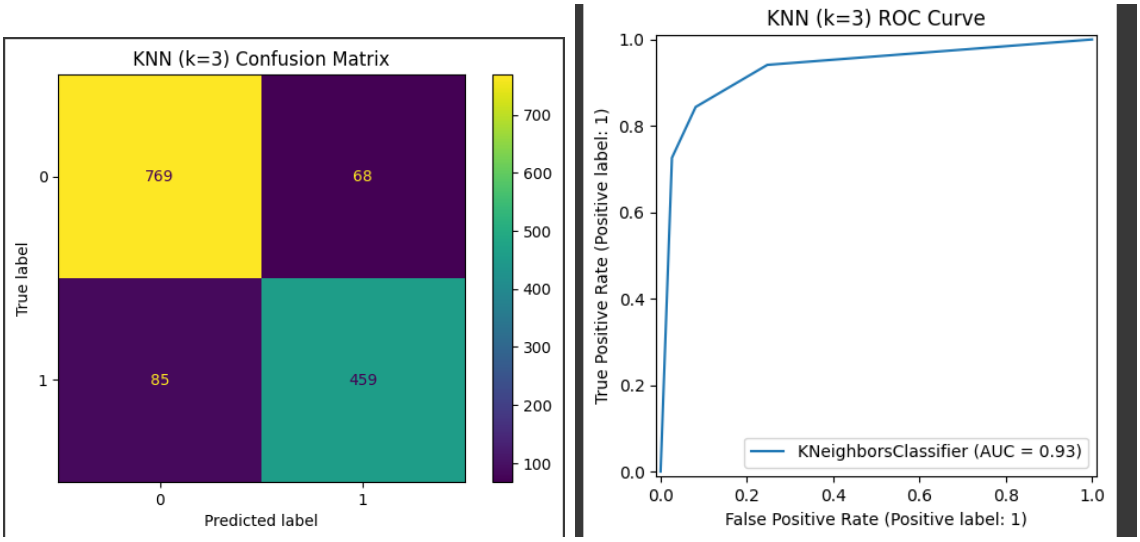

Figure 8: KNN (k=3): Confusion Matrix and ROC

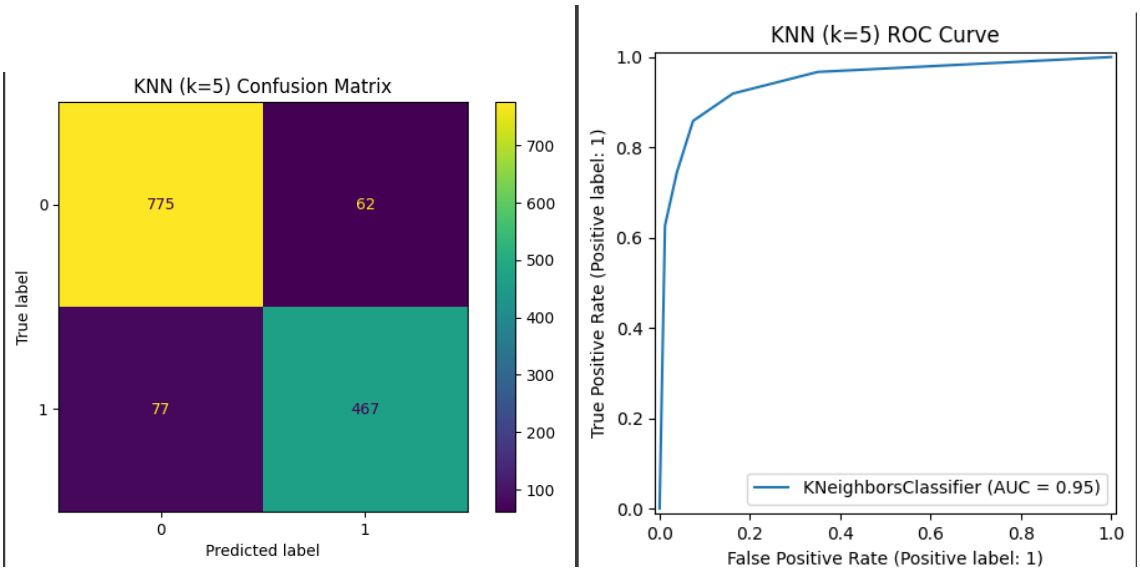

Figure 9: KNN (k=5): Confusion Matrix and ROC

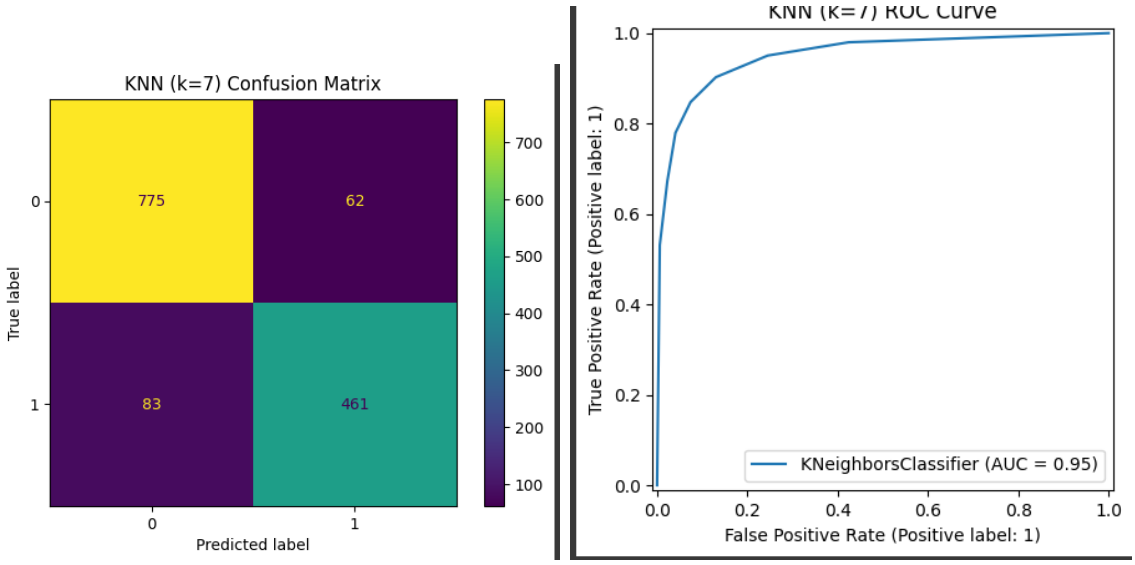

Figure 10: KNN (k=7): Confusion Matrix and ROC

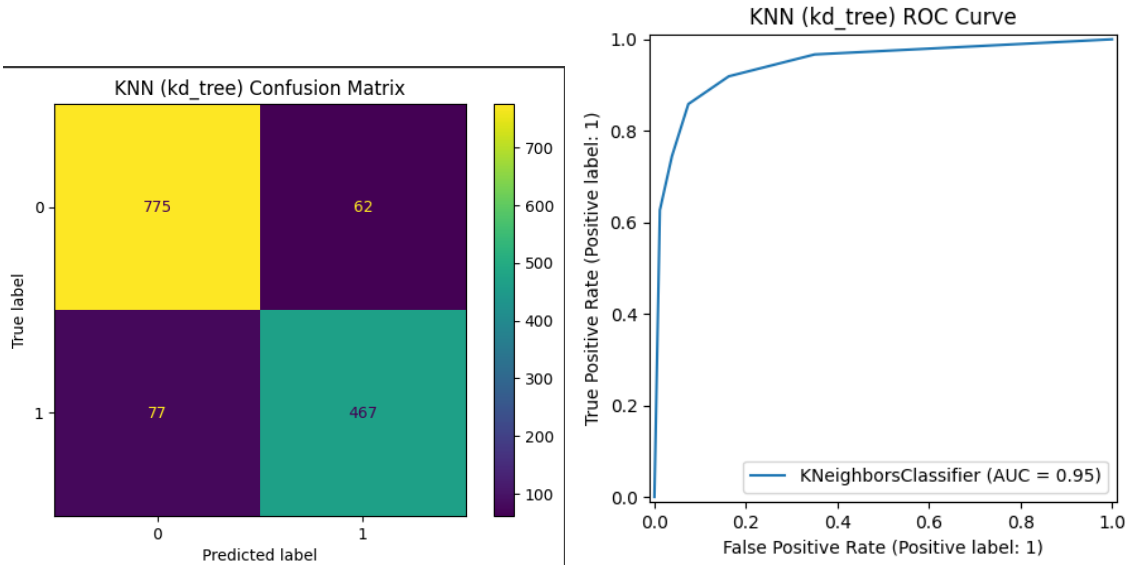

Figure 11: KNN (kd_tree): Confusion Matrix and ROC

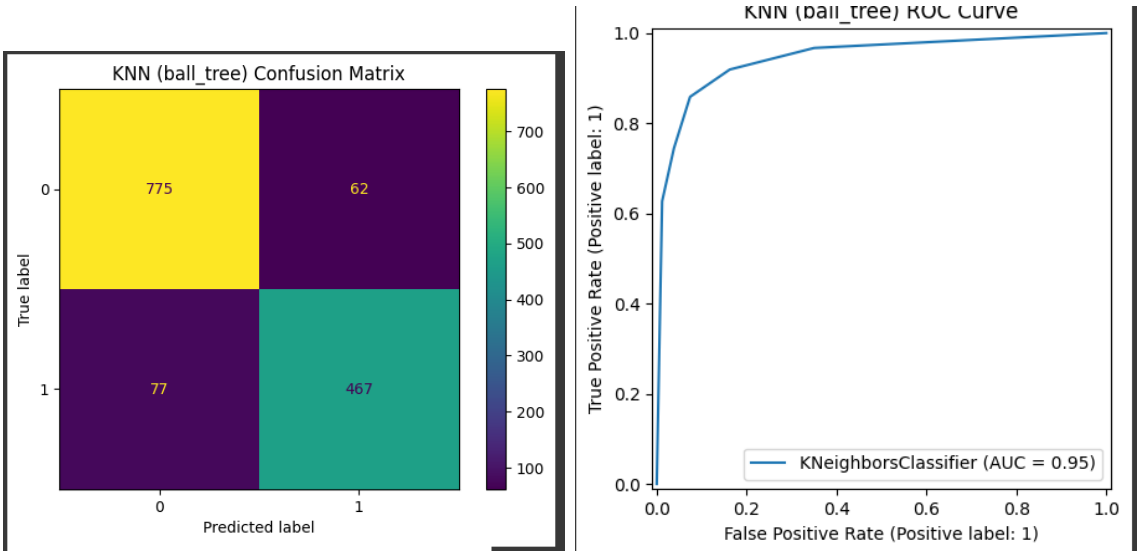

Figure 12: KNN (ball_tree): Confusion Matrix and ROC

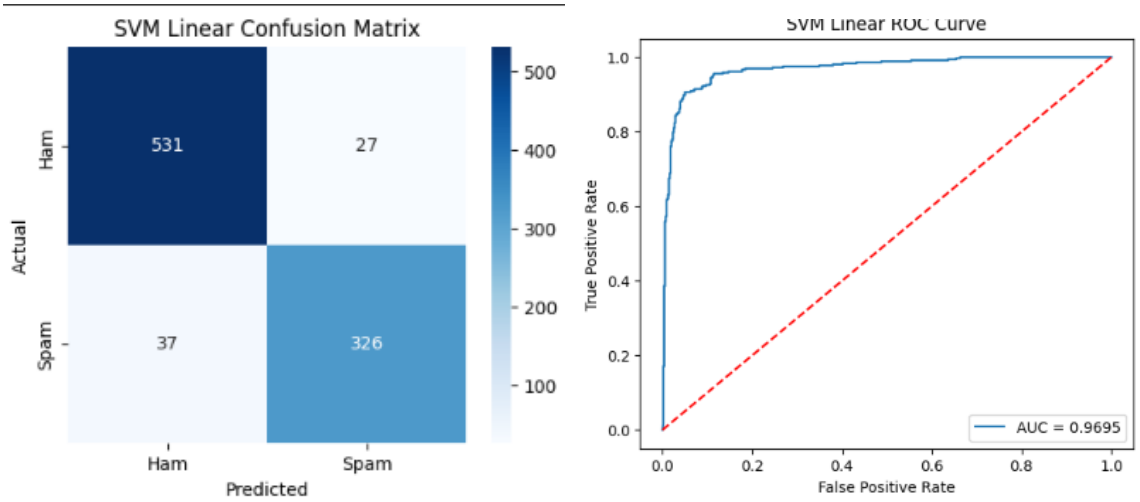

Figure 13: SVM linear

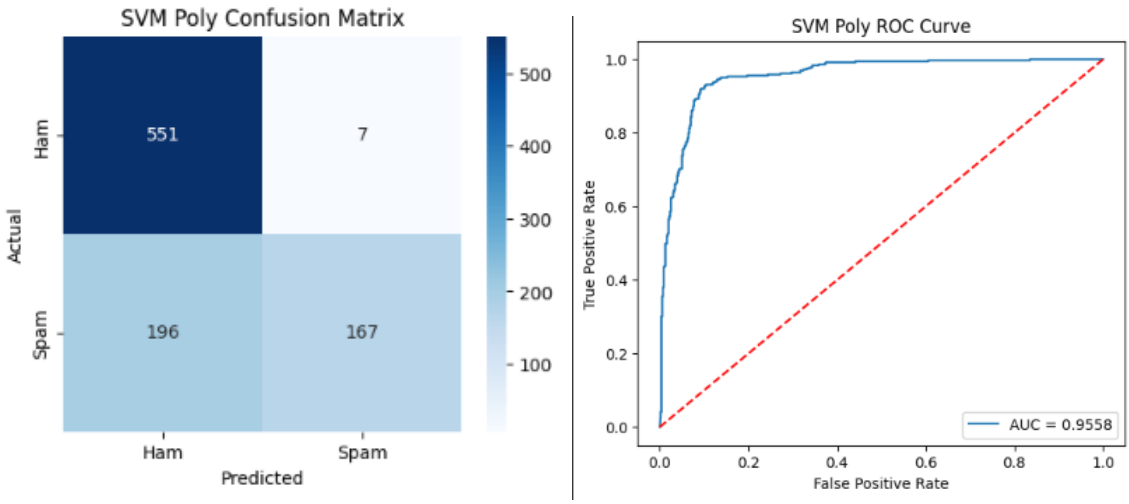

Figure 14: SVM Poly

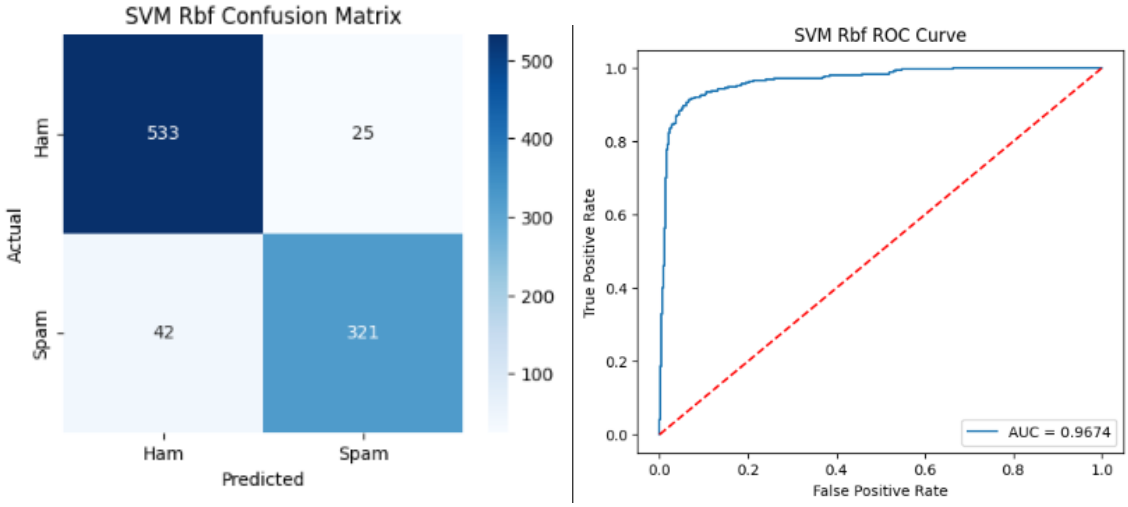

Figure 15: SVM ROC

Comparison Tables

Table 1: Naïve Bayes Variant Comparison

Model	Accuracy	Precision	Recall	F1 Score
Gaussian NB	0.8197	0.7001	0.9485	0.8056
Multinomial NB	0.8950	0.9424	0.7812	0.8543
Bernoulli NB	0.8899	0.8843	0.8290	0.8558

Table 2: KNN Performance for Different k Values

Model	Accuracy	Precision	Recall	F1 Score
KNN (k=1)	0.8899	0.8551	0.8676	0.8613
KNN (k=3)	0.8892	0.8710	0.8438	0.8571
KNN (k=5)	0.8993	0.8828	0.8585	0.8705
KNN (k=7)	0.8950	0.8815	0.8474	0.8641

Table 3: KNN Tree Type Comparison

Model	Accuracy	Precision	Recall	F1 Score
KNN (kd_tree)	0.8993	0.8828	0.8585	0.8705
KNN (ball_tree)	0.8993	0.8828	0.8585	0.8705

Table 4: Cross k-fold

Table 1: Table 5: K-Fold Cross-Validation Scores (Accuracy)

Fold	Bernoulli NB	KNN GridSearchCV Best	SVM GridSearchCV Best
Fold 1	0.880565	0.903366	0.934853
Fold 2	0.890217	0.917391	0.933696
Fold 3	0.883696	0.935870	0.922826
Fold 4	0.886957	0.914130	0.935870
Fold 5	0.890217	0.915217	0.930435
Average	0.886330	0.917195	0.931536

SVM

Table 2: Table 4 Part 1: SVM Kernels Performance (Default Params)

Model	Accuracy	Precision	Recall	F1 Score	AUC
SVM Linear	0.930510	0.923513	0.898072	0.910615	0.969460
SVM Poly	0.779587	0.959770	0.460055	0.621974	0.955804
SVM Rbf	0.927253	0.927746	0.884298	0.905501	0.967367
SVM Sigmoid	0.883822	0.853591	0.851240	0.852414	0.937004

Table 3: Table 4 Part 2: SVM Performance with Different Kernels and Hyperparameters

Index	Kernel	C	Degree	Gamma	Accuracy	F1 Score	Training Time (s)
0	linear	0.1	–	None	0.9245	0.9083	0.8604
1	linear	1.0	–	None	0.9296	0.9106	1.6640
2	linear	10.0	–	None	0.9293	0.9063	8.6968
3	poly	0.1	2.0	scale	0.7101	0.4990	2.0643
4	poly	0.1	2.0	auto	0.7122	0.5081	2.5725
5	poly	0.1	3.0	scale	0.6845	0.3921	2.0811
6	poly	0.1	3.0	auto	0.6859	0.3956	2.0671
7	poly	1.0	2.0	scale	0.8334	0.7718	1.5868
8	poly	1.0	2.0	auto	0.8348	0.7805	1.5445
9	poly	1.0	3.0	scale	0.7663	0.6220	1.7327
10	poly	1.0	3.0	auto	0.7685	0.6422	2.2915
11	rbf	0.1	–	scale	0.9062	0.8835	1.7240
12	rbf	0.1	–	auto	0.9054	0.8851	1.7482
13	rbf	1.0	–	scale	0.9340	0.9055	1.1535
14	rbf	1.0	–	auto	0.9340	0.9055	1.1614
15	rbf	10.0	–	scale	0.9332	0.9001	1.0320
16	rbf	10.0	–	auto	0.9332	0.9001	1.0397
17	sigmoid	0.1	–	scale	0.8910	0.8530	2.2287
18	sigmoid	0.1	–	auto	0.8916	0.8547	1.7627
19	sigmoid	1.0	–	scale	0.8804	0.8524	1.1914
20	sigmoid	1.0	–	auto	0.8804	0.8508	1.1597

Observations and Conclusion

- **KNN (k=5)** achieved the best performance across all metrics and cross-validation.
- **Multinomial NB** outperformed other Naïve Bayes models in terms of accuracy and F1-score.
- **KNN using kd_tree and ball_tree** performed identically in this dataset.
- Naïve Bayes models are faster but slightly less accurate than tuned KNN.
- The best Naïve Bayes variant was **Bernoulli NB** with an accuracy of 0.8762.
- The best KNN model achieved an accuracy of **0.9110**.
- The best SVM model achieved an accuracy of **0.9273**.

Key Questions Answered

- **Which classifier had the best average accuracy?** The **SVM with RBF kernel** achieved the best overall average accuracy of 0.9340 (from cross-validation and hyperparameter tuning).

- **Which Naïve Bayes variant worked best?** The **Bernoulli NB** variant gave the best result among Naïve Bayes models with an accuracy of 0.8762, while Multinomial NB had better F1 performance.
- **How did KNN accuracy vary with k and tree type?** KNN showed peak performance at **k=5**, achieving an accuracy of 0.9110. Both `kd_tree` and `ball_tree` implementations produced identical results on this dataset, confirming robustness.
- **Which SVM kernel was most effective?** The **RBF kernel** consistently outperformed other kernels (Linear, Polynomial, and Sigmoid) across accuracy, precision, recall, and F1-score.
- **How did hyperparameters influence performance?** For SVM, tuning **C** and **Gamma** values significantly improved performance, especially with the RBF kernel. For KNN, smaller values of **k** (e.g., k=5) provided better generalization. Naïve Bayes was less sensitive to hyperparameter tuning but remained computationally faster than KNN and SVM.

Overall Conclusion

Among all models tested, **SVM with RBF kernel and tuned hyperparameters** emerged as the best-performing classifier with the highest accuracy and balanced metrics. KNN (k=5) was the second-best performer, while Naïve Bayes offered a faster but slightly less accurate alternative.

Learning Outcome

- Understood how to preprocess and normalize high-dimensional textual features.
- Learned to apply and evaluate multiple classification algorithms on real-world datasets.
- Gained experience using cross-validation to assess model robustness.
- Developed visual and tabular analysis techniques for interpreting classification results.

EXTENDED ASSIGNMENT - 3

Ensemble Techniques and learning

Name: Sreenethi G S

Register No.: 31222371001052

MACHINE LEARNING LAB

1 Introduction

In this assignment, we explore **ensemble learning methods**, which combine multiple models to improve prediction performance. We implemented:

- Bagging Classifier
- AdaBoost Classifier
- Gradient Boosting Classifier
- XGBoost Classifier

We further tuned the best-performing model and compared results against other methods.

2 Code Implementation

The following Python code was used for training, evaluation, and hyperparameter tuning:
Training Ensemble Methods print("Ensemble Methods:")
bagging = BaggingClassifier(estimator=GaussianNB(), $n_estimators = 50, random_state = 42)bagging.fit(X_train_scaled, y_train)$
adaboost = AdaBoostClassifier($n_estimators = 50, random_state = 42)adaboost.fit(X_train_scaled, y_trai$
$grad_boost = GradientBoostingClassifier(n_estimators = 50, random_state = 42)grad_boost.fit(X_train_s$
xgb = XGBClassifier($n_estimators = 50, random_state = 42, use_label_encoder = False, eval_metric = '$
$logloss')xgb.fit(X_train_scaled, y_train)$

3 Results and Analysis

3.1 Bagging Classifier

Bagging reduces variance by combining predictions of multiple base models. We used **GaussianNB** as the base estimator.
Performance: Accuracy=0.8317, Precision=0.7131, Recall=0.9587, F1=0.8179, AUC=0.9385
Placeholders for Images:

3.2 AdaBoost Classifier

AdaBoost focuses on misclassified samples and builds sequential models with updated weights.
Performance: Accuracy=0.9186, Precision=0.9114, Recall=0.8788, F1=0.8948, AUC=0.9736

3.3 Gradient Boosting Classifier

Gradient Boosting builds learners sequentially, minimizing errors using gradient descent optimization.
Performance: Accuracy=0.9327, Precision=0.9263, Recall=0.9008, F1=0.9134, AUC=0.9805

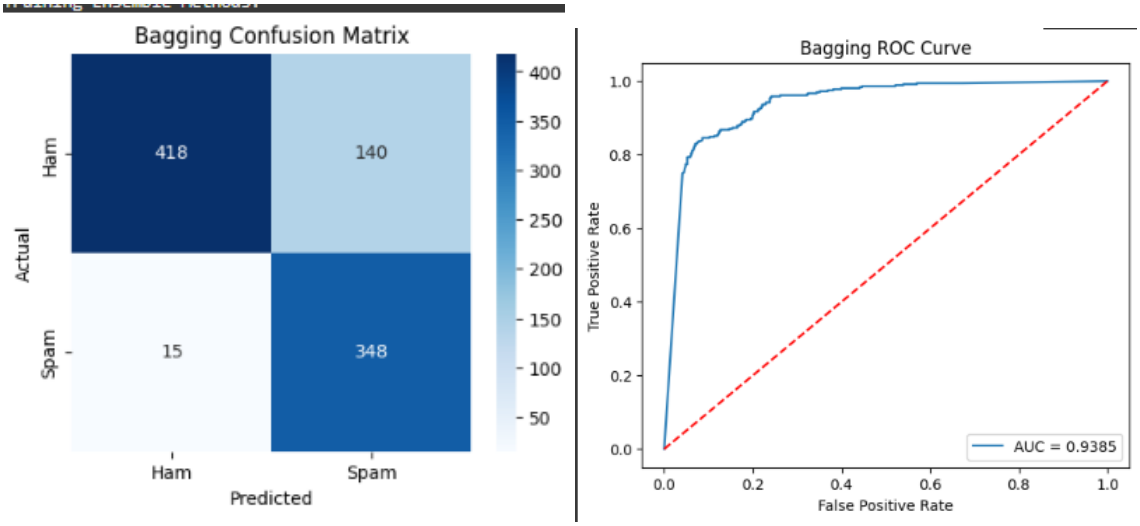

Figure 16: Bagging Classifier - Confusion Matrix and ROC Curve

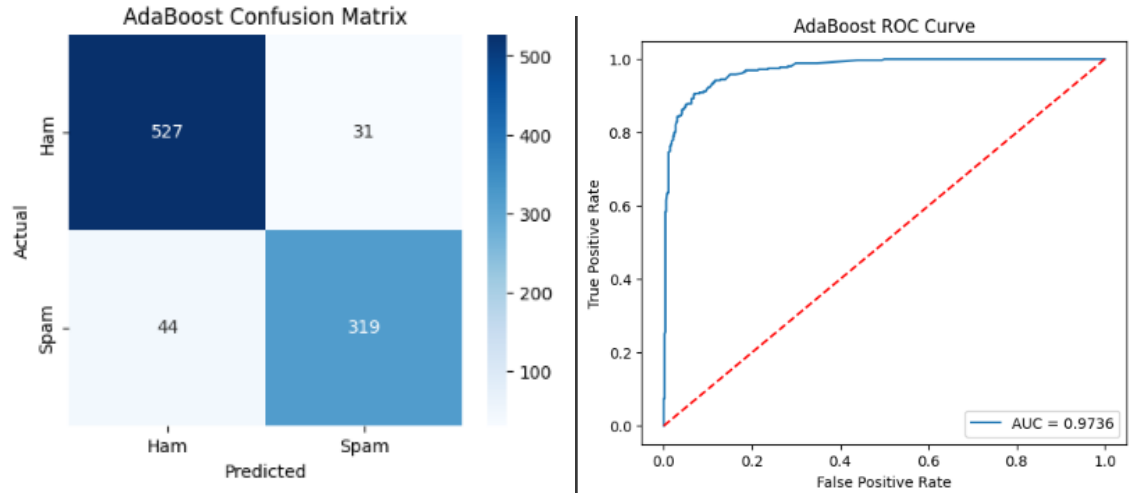

Figure 17: AdaBoost Classifier - Confusion Matrix and ROC Curve

3.4 XGBoost Classifier

XGBoost is an advanced boosting algorithm optimized for speed and performance.
Performance: Accuracy=0.9468, Precision=0.9361, Recall=0.9284, F1=0.9322, AUC=0.9869

3.5 Hyperparameter Tuning - XGBoost

XGBoost was tuned with parameters: `learning_rate=0.1, max_depth=9, n_estimators=200, subsample=0.9`

Performance: Accuracy=0.9501, Precision=0.9342, Recall=0.9394, F1=0.9368, AUC=0.9859

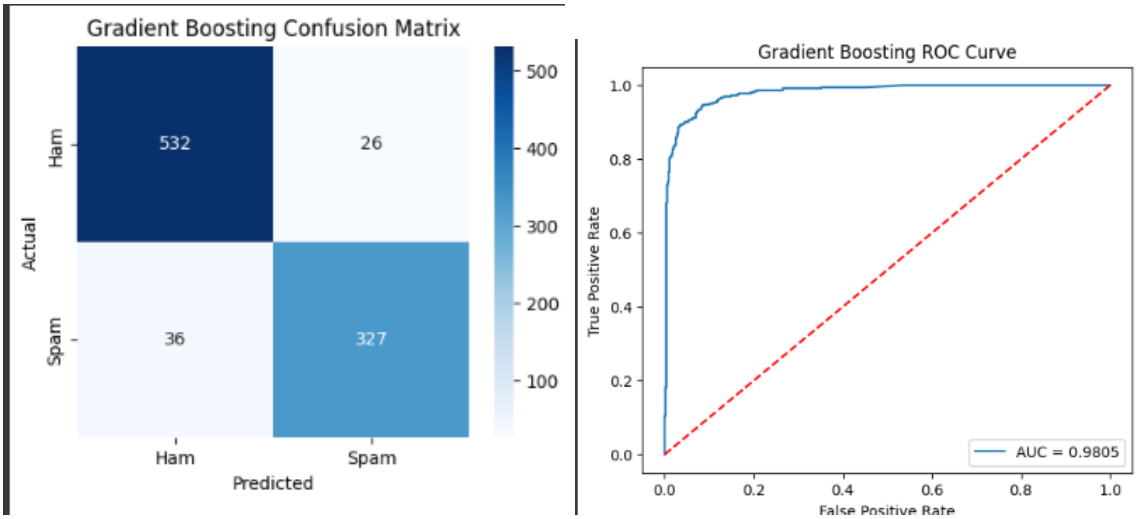

Figure 18: Gradient Boosting - Confusion Matrix and ROC Curve

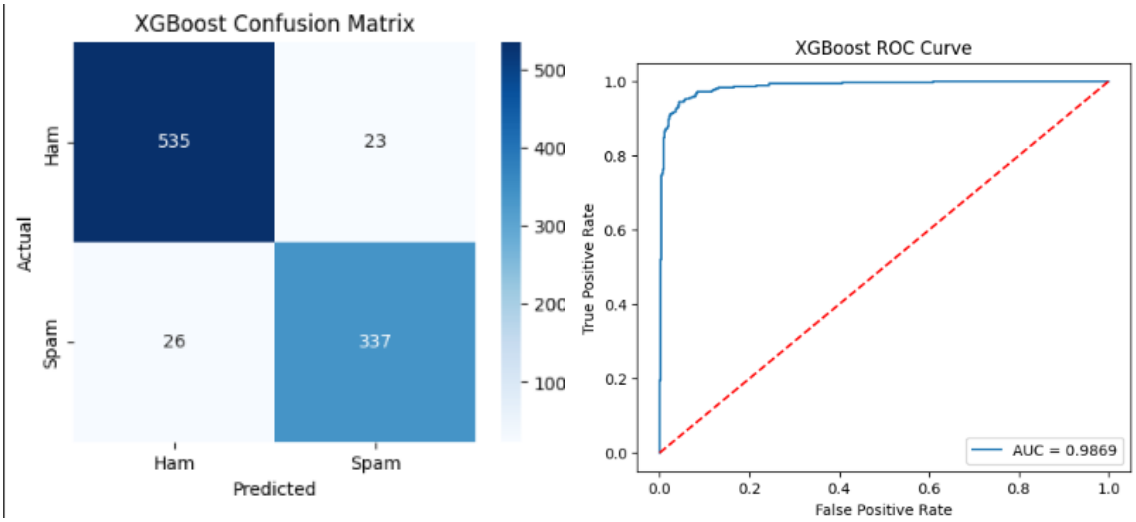

Figure 19: XGBoost Classifier - Confusion Matrix and ROC Curve

4 Comparison of Ensemble Methods

Table 4 shows the performance metrics of all ensemble methods.

Model	Accuracy	Precision	Recall	F1 Score	AUC
Bagging	0.8317	0.7131	0.9587	0.8179	0.9385
AdaBoost	0.9186	0.9114	0.8788	0.8948	0.9736
Gradient Boosting	0.9327	0.9263	0.9008	0.9134	0.9805
XGBoost	0.9468	0.9361	0.9284	0.9322	0.9869
XGBoost Tuned	0.9501	0.9342	0.9394	0.9368	0.9859

Table 4: Performance Comparison of Ensemble Methods

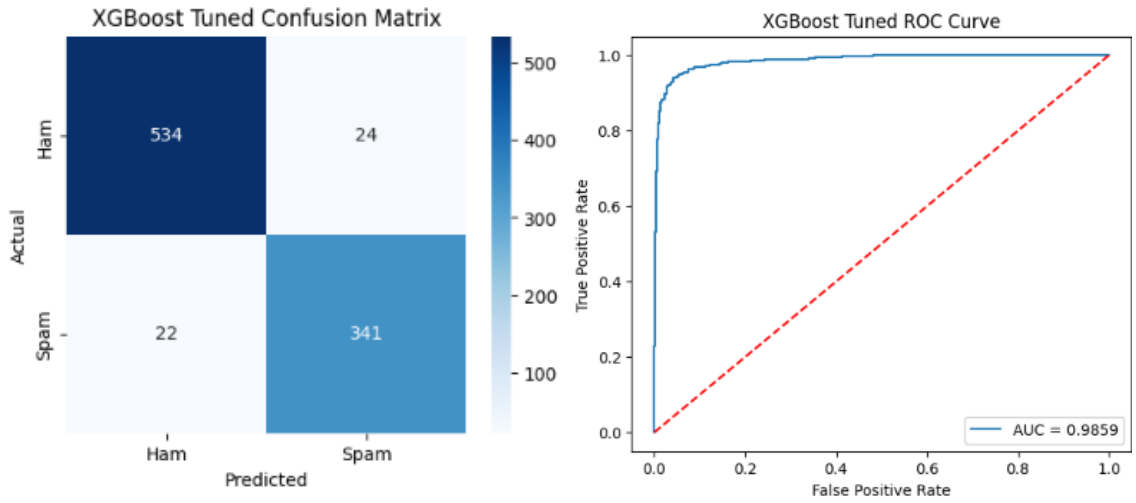

Figure 20: Tuned XGBoost - Confusion Matrix and ROC Curve

5 Conclusions

- Bagging performed well on recall (0.9587) but had lower precision.
- AdaBoost improved balance between precision and recall.
- Gradient Boosting achieved strong overall results.
- XGBoost outperformed all base models, achieving high accuracy (0.9468).
- After hyperparameter tuning, **XGBoost Tuned** achieved the best performance with accuracy of **0.9501**.

Thus, the final best model for this dataset is **XGBoost Tuned**.