Sri Sivasubramaniya Nadar College of Engineering, Chennai
(An autonomous Institution affiliated to Anna University)

Degree & Branch	B.E. Computer Science & Engineering	Semester	V
Subject Code & Name	ICS1512 & Machine Learning Algorithms Laboratory		
Academic year	2025-2026 (Odd)	Batch:2023-2028	**Due date:**

Experiment 3: Email Spam or Ham Classification using Naive Bayes, KNN, and SVM

1 Aim:

To design and implement classification models using Naive Bayes variants and K-Nearest Neighbors (KNN) algorithms to accurately classify emails as spam or ham. Additionally, to evaluate and compare their effectiveness using multiple performance metrics.

2 Libraries used:

- Numpy
- Pandas
- Matplotlib
- Scikit-learn
- Seaborn

3 Objective:

- To preprocess the email dataset by cleaning text data, vectorizing features, and splitting the data for training and testing.
- To implement Naive Bayes classifiers (Bernoulli, Multinomial, Gaussian) and KNN classifiers, tuning parameters such as k-value.
- To measure and compare model performance using accuracy, precision, recall, F1-score, confusion matrix, and ROC-AUC, enabling informed model selection.

4 Naive Bayes Code:

Importing Required Libraries

```
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
from sklearn.preprocessing import StandardScaler  # or MinMaxScaler, if used
from sklearn.preprocessing import LabelEncoder  # if categorical labels
from sklearn.naive_bayes import BernoulliNB, MultinomialNB, GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix, classification_re
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import cross_val_score, KFold
```

Loading dataset

```
df = pd.read_csv('spambase.csv')
```

Basic info

```
print("Dataset Info:\n", df.info())
print("\nFirst 5 rows:\n", df.head())
print("\nMissing Values:\n", df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 58 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
 16  word_freq_business          4601 non-null   float64
 17  word_freq_email             4601 non-null   float64
```

```
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
 20  word_freq_your              4601 non-null   float64
 21  word_freq_font              4601 non-null   float64
 22  word_freq_000               4601 non-null   float64
 23  word_freq_money             4601 non-null   float64
 24  word_freq_hp                4601 non-null   float64
 25  word_freq_hpl               4601 non-null   float64
 26  word_freq_george            4601 non-null   float64
 27  word_freq_650               4601 non-null   float64
 28  word_freq_lab               4601 non-null   float64
 29  word_freq_labs              4601 non-null   float64
 30  word_freq_telnet            4601 non-null   float64
 31  word_freq_857               4601 non-null   float64
 32  word_freq_data              4601 non-null   float64
 33  word_freq_415               4601 non-null   float64
 34  word_freq_85                4601 non-null   float64
 35  word_freq_technology        4601 non-null   float64
 36  word_freq_1999              4601 non-null   float64
 37  word_freq_parts             4601 non-null   float64
 38  word_freq_pm                4601 non-null   float64
 39  word_freq_direct            4601 non-null   float64
 40  word_freq_cs                4601 non-null   float64
 41  word_freq_meeting           4601 non-null   float64
 42  word_freq_original          4601 non-null   float64
 43  word_freq_project           4601 non-null   float64
 44  word_freq_re                4601 non-null   float64
 45  word_freq_edu               4601 non-null   float64
 46  word_freq_table             4601 non-null   float64
 47  word_freq_conference        4601 non-null   float64
 48  char_freq_%3B               4601 non-null   float64
 49  char_freq_%28               4601 non-null   float64
 50  char_freq_%5B               4601 non-null   float64
 51  char_freq_%21               4601 non-null   float64
 52  char_freq_%24               4601 non-null   float64
```

Handling missing values

```
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

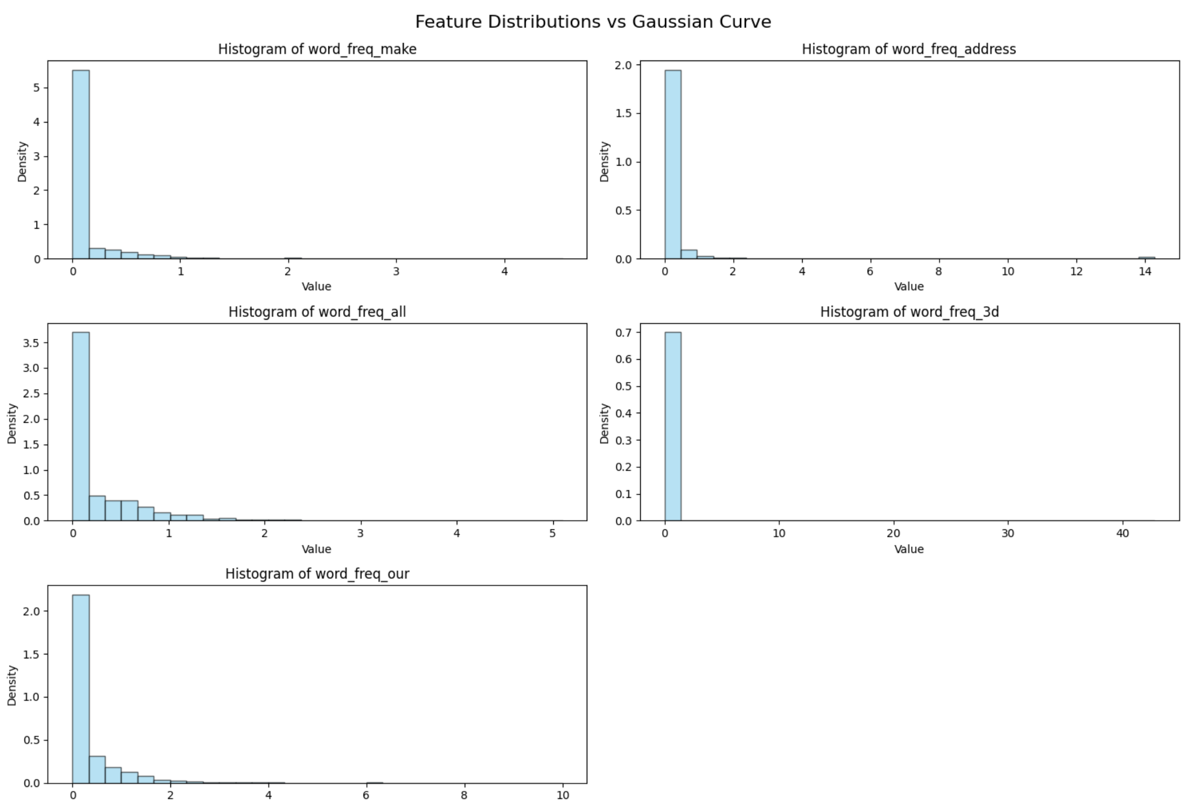
Splitting of feature and target

```
X = df_imputed.drop('class', axis=1)
y = df_imputed['class']
```

Checking Distribution

```
X = df.drop('class', axis=1)  # Assuming 'class' is the target

# Choose a few features to visualize
sample_features = X.columns[:5]  # First 5 features
```

```
sample_features = X.columns[:5]  # First 5 features

# Plot histograms with Gaussian curve overlay
plt.figure(figsize=(15, 10))
for i, feature in enumerate(sample_features):
    plt.subplot(3, 2, i + 1)
    data = X[feature]

    # Plot histogram
    count, bins, ignored = plt.hist(data, bins=30, density=True, alpha=0.6, col

    # Plot normal distribution curve
    '''mu, std = data.mean(), data.std()
    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax, 100)
    p = 1 / (std * np.sqrt(2 * np.pi)) * np.exp(-(x - mu)**2 / (2 * std**2))
    plt.plot(x, p, 'r', linewidth=2)'''

    plt.title(f'Histogram of {feature}')
    plt.xlabel('Value')
    plt.ylabel('Density')

plt.tight_layout()
plt.suptitle('Feature Distributions vs Gaussian Curve', fontsize=16, y=1.02)
plt.show()
```



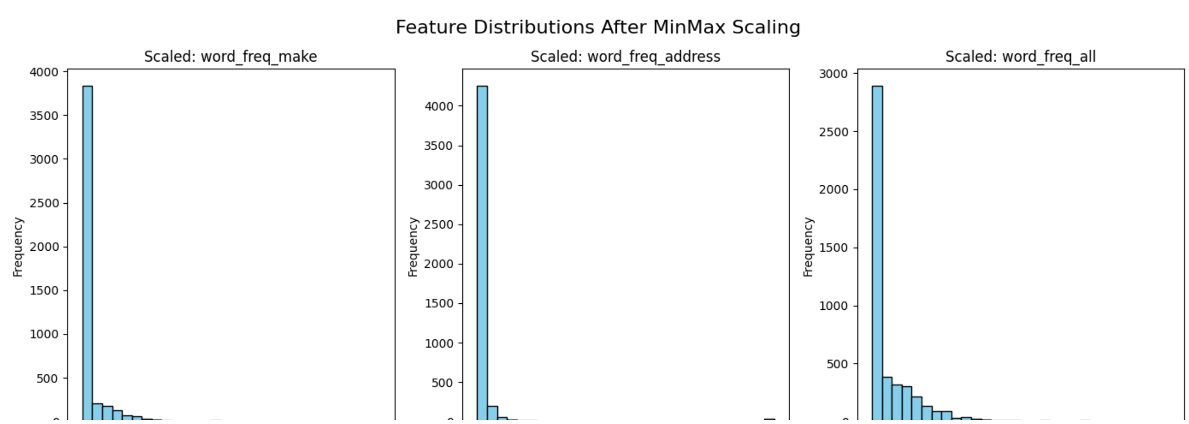
Applying min max scaling

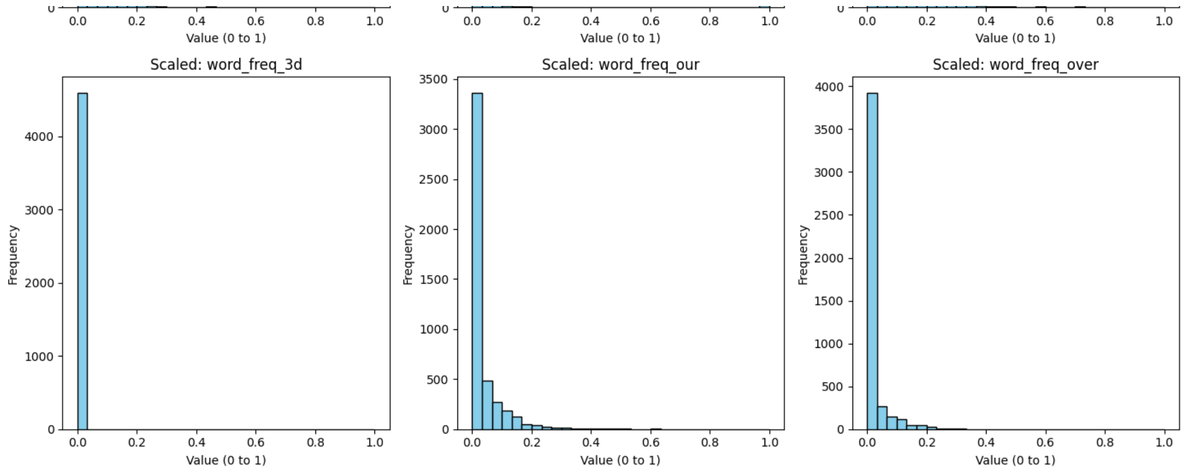
```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Plots

Histogram

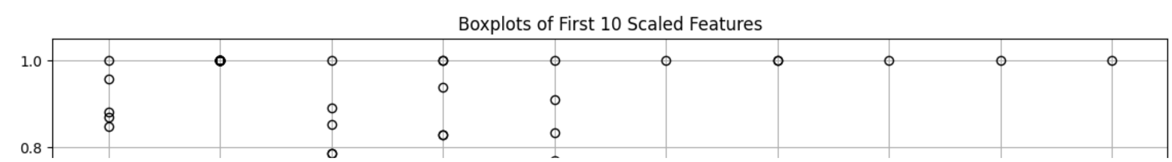
```
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 10))
for i in range(6):  # first 6 features as an example
    plt.subplot(2, 3, i + 1)
    plt.hist(X_scaled[:, i], bins=30, color='skyblue', edgecolor='black')
    plt.title(f'Scaled: {X.columns[i]}')
    plt.xlabel('Value (0 to 1)')
    plt.ylabel('Frequency')
plt.tight_layout()
plt.suptitle('Feature Distributions After MinMax Scaling', fontsize=16, y=1.02)
plt.show()
```

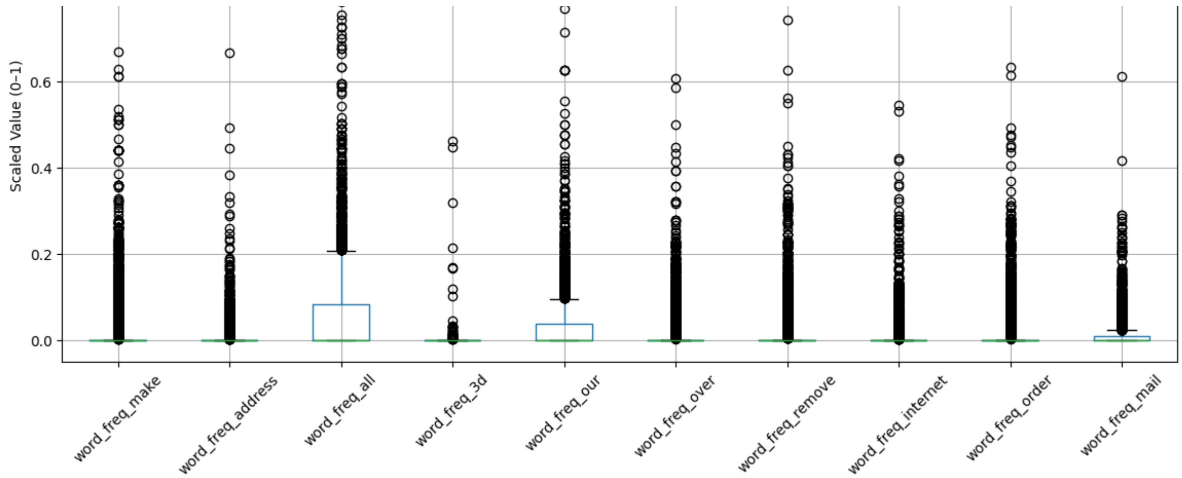




Boxplot

```
import pandas as pd

# Convert scaled array back to DataFrame for easier plotting
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

plt.figure(figsize=(14, 6))
X_scaled_df.iloc[:, :10].boxplot(rot=45)
plt.title('Boxplots of First 10 Scaled Features')
plt.ylabel('Scaled Value (0–1)')
plt.grid(True)
plt.show()
```

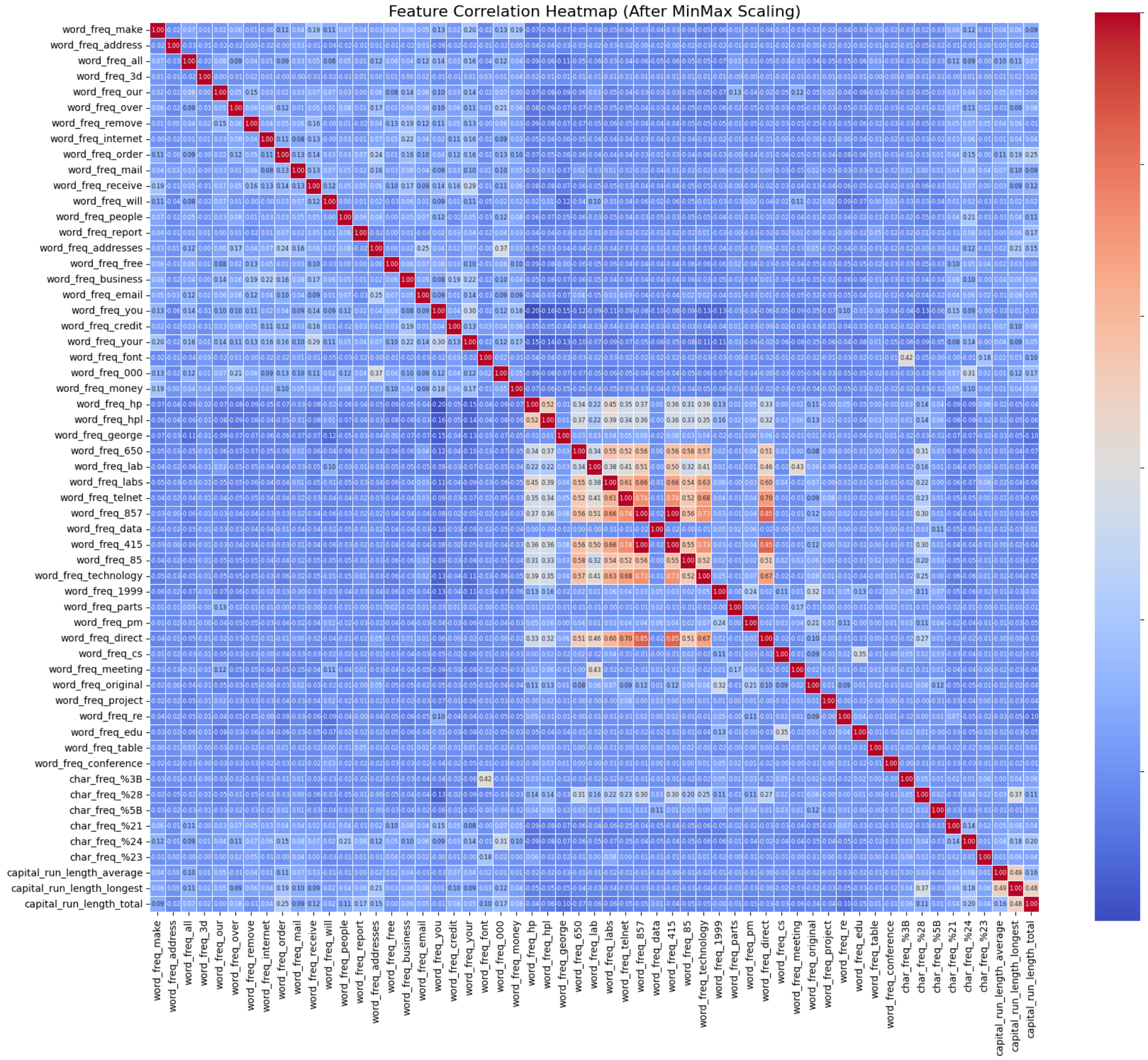




Correlation HeatMap

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Assuming X_scaled_df is your scaled DataFrame
corr_matrix = X_scaled_df.corr()

plt.figure(figsize=(18, 15))  # Bigger figure
sns.heatmap(
    corr_matrix,
    cmap='coolwarm',
    square=True,
    annot=True,              # Show values inside squares
    fmt=".2f",               # Format to 2 decimal places
    linewidths=0.5,          # Thin grid lines
    annot_kws={"size": 6}    # Smaller font size
)

plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.title("Feature Correlation Heatmap (After MinMax Scaling)", fontsize=16)
plt.tight_layout()
```

```
plt.show()
```



Model Training

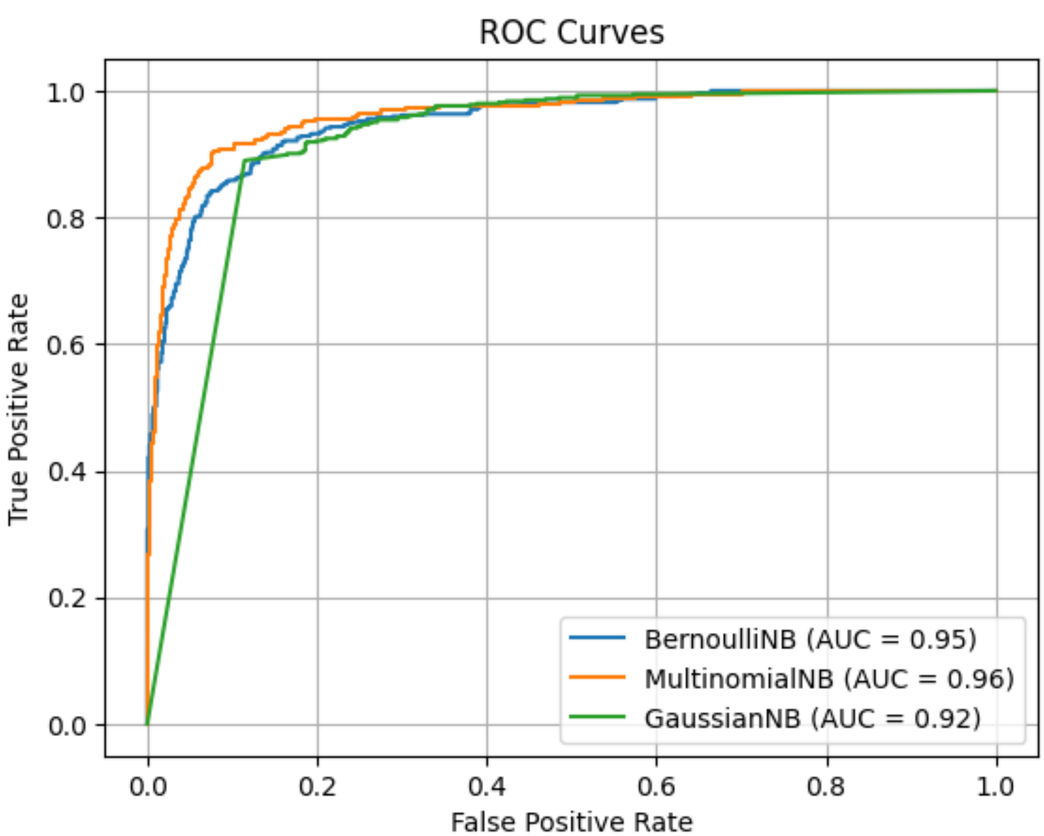
```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_curve, auc,
    classification_report
)

# Split the data again (or reuse your earlier split)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, stratify=y, random_state=42
)
```

```
def evaluate_model(name, model, X_test, y_test, results):
    y_pred = model.predict(X_test)

    # For ROC and AUC, use predict_proba
    try:
        y_proba = model.predict_proba(X_test)[:, 1]
    except:
        y_proba = y_pred  # fallback if proba not available

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, zero_division=0)
    rec = recall_score(y_test, y_pred, zero_division=0)
    f1 = f1_score(y_test, y_pred, zero_division=0)

    # Print Confusion Matrix and Classification Report
    print(f"\n=== {name} ===")
    print("Confusion Matrix:")
    print(confusion_matrix(y_test, y_pred))
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, zero_division=0))

    # ROC and AUC
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    roc_auc = auc(fpr, tpr)

    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.2f})')

    # Append results
    results.append({
        'Model': name,
        'Accuracy': acc,
        'Precision': prec,
```

```
        'Recall': rec,
        'F1 Score': f1,
        'AUC': roc_auc
    })


# Store results
results = []

# --- BernoulliNB ---
bnb = BernoulliNB()
bnb.fit(X_train, y_train)
evaluate_model("BernoulliNB", bnb, X_test, y_test, results)

# --- MultinomialNB ---
mnb = MultinomialNB()
mnb.fit(X_train, y_train)
evaluate_model("MultinomialNB", mnb, X_test, y_test, results)

# --- GaussianNB ---
gnb = GaussianNB()
gnb.fit(X_train, y_train)
evaluate_model("GaussianNB", gnb, X_test, y_test, results)

# === ROC Curve Plot ===
plt.title('ROC Curves')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# === Final Comparison Table ===
results_df = pd.DataFrame(results)
print("\n=== Comparison Table ===")
print(results_df)
```

```
=== BernoulliNB ===
Confusion Matrix:
[[778  59]
 [ 93 451]]

Classification Report:
              precision    recall  f1-score   support

         0.0       0.89      0.93      0.91       837
         1.0       0.88      0.83      0.86       544

    accuracy                           0.89      1381
   macro avg       0.89      0.88      0.88      1381
weighted avg       0.89      0.89      0.89      1381


=== MultinomialNB ===
```

```
Confusion Matrix:
[[811  26]
 [119 425]]

Classification Report:
              precision    recall  f1-score   support

         0.0       0.87      0.97      0.92       837
         1.0       0.94      0.78      0.85       544

    accuracy                           0.90      1381
   macro avg       0.91      0.88      0.89      1381
weighted avg       0.90      0.90      0.89      1381


=== GaussianNB ===
Confusion Matrix:
[[616 221]
 [ 28 516]]

Classification Report:
              precision    recall  f1-score   support

         0.0       0.96      0.74      0.83       837
         1.0       0.70      0.95      0.81       544

    accuracy                           0.82      1381
   macro avg       0.83      0.84      0.82      1381
weighted avg       0.86      0.82      0.82      1381
```




```
=== Comparison Table ===
           Model  Accuracy  Precision    Recall  F1 Score       AUC
```

```
0    BernoulliNB  0.889935   0.884314  0.829044  0.855787  0.950023
1  MultinomialNB  0.895004   0.942350  0.781250  0.854271  0.960696
2     GaussianNB  0.819696   0.700136  0.948529  0.805621  0.915675
```

```python
cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize Multinomial Naive Bayes
mnb = MultinomialNB()

# Perform cross-validation (accuracy as scoring)
scores = cross_val_score(mnb, X_scaled, y, cv=cv, scoring='accuracy')

# Print accuracy for each fold
print("✅ Multinomial Naive Bayes - 5-Fold Cross-Validation Results:")
for i, score in enumerate(scores, 1):
    print(f"Fold {i} Accuracy: {score:.4f}")

# Print average and standard deviation
print(f"\nMean Accuracy        : {scores.mean():.4f}")
print(f"Standard Deviation   : {scores.std():.4f}")
```

```
✅ Multinomial Naive Bayes - 5-Fold Cross-Validation Results:
Fold 1 Accuracy: 0.8719
Fold 2 Accuracy: 0.8935
Fold 3 Accuracy: 0.8891
Fold 4 Accuracy: 0.8913
Fold 5 Accuracy: 0.8859

Mean Accuracy        : 0.8863
Standard Deviation   : 0.0077
```

5 KNN Code:

Importing Required Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_s
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import matthews_corrcoef
```

Loading dataset

```
df = pd.read_csv('spambase.csv')
```

Basic info

```
print("Dataset Info:\n", df.info())
print("\nFirst 5 rows:\n", df.head())
print("\nMissing Values:\n", df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 58 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
```

```
 16  word_freq_business          4601 non-null   float64
 17  word_freq_email             4601 non-null   float64
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
 20  word_freq_your              4601 non-null   float64
 21  word_freq_font              4601 non-null   float64
 22  word_freq_000               4601 non-null   float64
 23  word_freq_money             4601 non-null   float64
 24  word_freq_hp                4601 non-null   float64
 25  word_freq_hpl               4601 non-null   float64
 26  word_freq_george            4601 non-null   float64
 27  word_freq_650               4601 non-null   float64
 28  word_freq_lab               4601 non-null   float64
 29  word_freq_labs              4601 non-null   float64
 30  word_freq_telnet            4601 non-null   float64
 31  word_freq_857               4601 non-null   float64
 32  word_freq_data              4601 non-null   float64
 33  word_freq_415               4601 non-null   float64
 34  word_freq_85                4601 non-null   float64
 35  word_freq_technology        4601 non-null   float64
 36  word_freq_1999              4601 non-null   float64
 37  word_freq_parts             4601 non-null   float64
 38  word_freq_pm                4601 non-null   float64
 39  word_freq_direct            4601 non-null   float64
 40  word_freq_cs                4601 non-null   float64
 41  word_freq_meeting           4601 non-null   float64
 42  word_freq_original          4601 non-null   float64
 43  word_freq_project           4601 non-null   float64
 44  word_freq_re                4601 non-null   float64
 45  word_freq_edu               4601 non-null   float64
 46  word_freq_table             4601 non-null   float64
 47  word_freq_conference        4601 non-null   float64
 48  char_freq_%3B               4601 non-null   float64
 49  char_freq_%28               4601 non-null   float64
 50  char_freq_%5B               4601 non-null   float64
 51  char_freq_%21               4601 non-null   float64
 52  char_freq_%24               4601 non-null   float64
```

Handling missing values

```
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

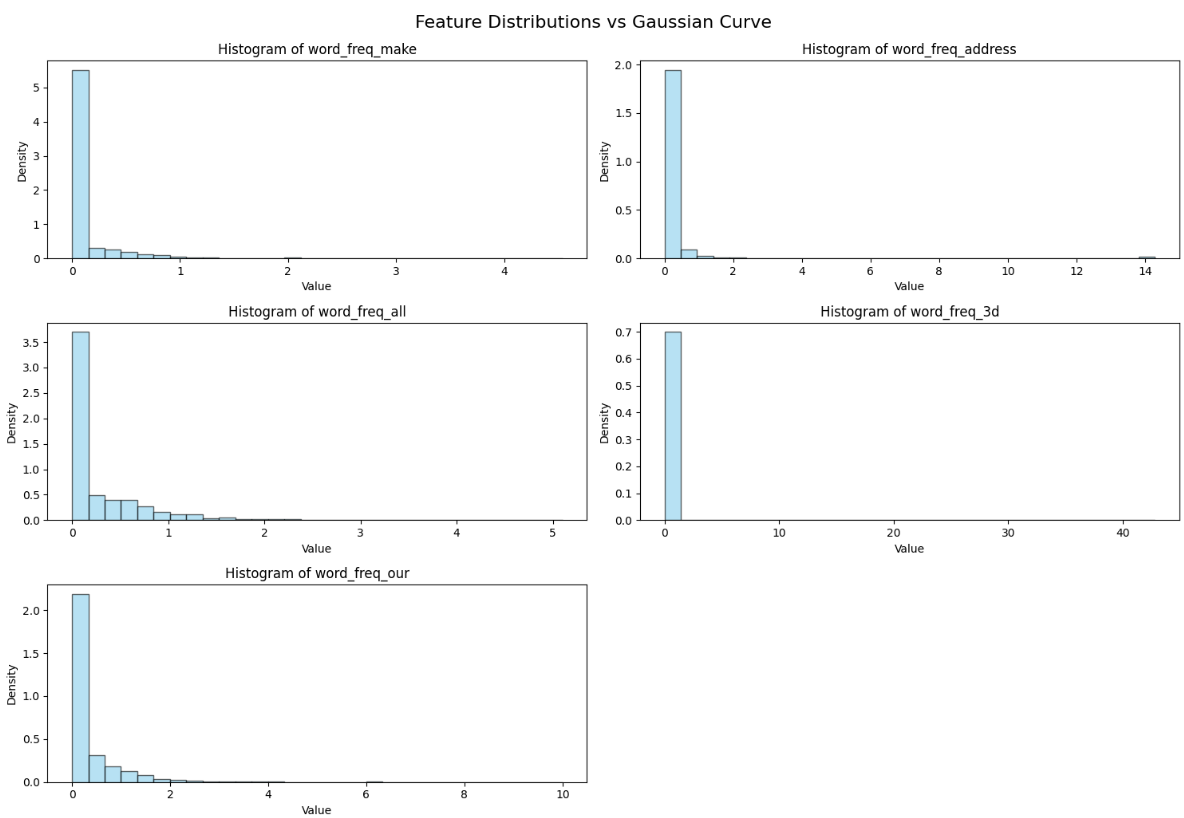
Splitting of feature and target

```
X = df_imputed.drop('class', axis=1)
y = df_imputed['class']
```

Checking Distribution

```
X = df.drop('class', axis=1)  # Assuming 'class' is the target
```

```
# Choose a few features to visualize
sample_features = X.columns[:5]  # First 5 features

# Plot histograms with Gaussian curve overlay
plt.figure(figsize=(15, 10))
for i, feature in enumerate(sample_features):
    plt.subplot(3, 2, i + 1)
    data = X[feature]

    # Plot histogram
    count, bins, ignored = plt.hist(data, bins=30, density=True, alpha=0.6, col

    # Plot normal distribution curve
    '''mu, std = data.mean(), data.std()
    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax, 100)
    p = 1 / (std * np.sqrt(2 * np.pi)) * np.exp(-(x - mu)**2 / (2 * std**2))
    plt.plot(x, p, 'r', linewidth=2)'''

    plt.title(f'Histogram of {feature}')
    plt.xlabel('Value')
    plt.ylabel('Density')

plt.tight_layout()
plt.suptitle('Feature Distributions vs Gaussian Curve', fontsize=16, y=1.02)
plt.show()
```



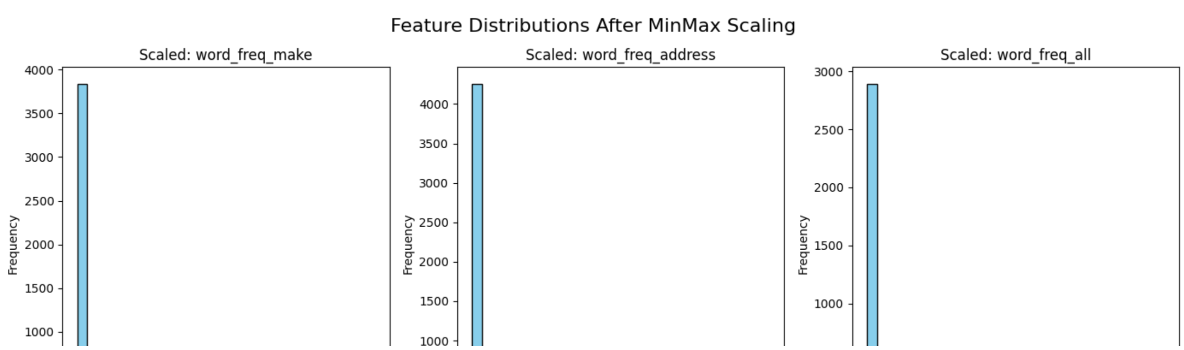
Applying min max scaling

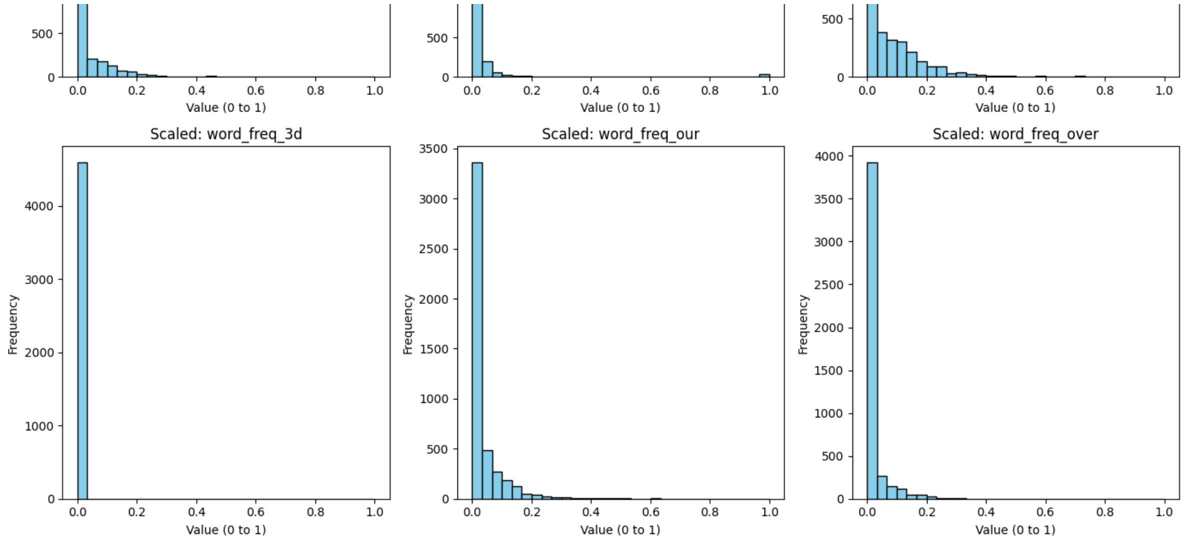
```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Plots

Histogram

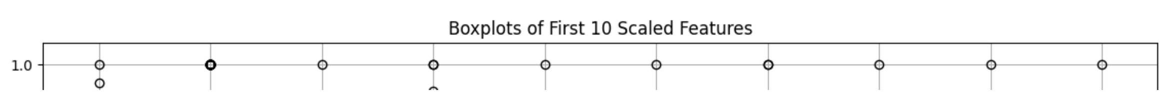
```
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 10))
for i in range(6):  # first 6 features as an example
    plt.subplot(2, 3, i + 1)
    plt.hist(X_scaled[:, i], bins=30, color='skyblue', edgecolor='black')
    plt.title(f'Scaled: {X.columns[i]}')
    plt.xlabel('Value (0 to 1)')
    plt.ylabel('Frequency')
plt.tight_layout()
plt.suptitle('Feature Distributions After MinMax Scaling', fontsize=16, y=1.02)
plt.show()
```

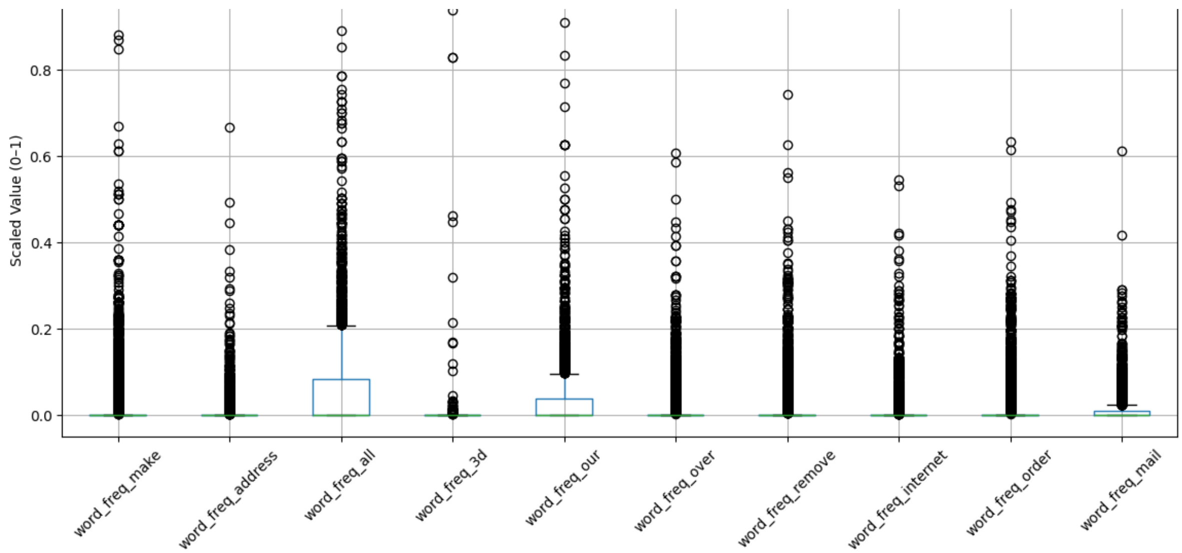




Boxplot

```
import pandas as pd

# Convert scaled array back to DataFrame for easier plotting
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

plt.figure(figsize=(14, 6))
X_scaled_df.iloc[:, :10].boxplot(rot=45)
plt.title('Boxplots of First 10 Scaled Features')
plt.ylabel('Scaled Value (0–1)')
plt.grid(True)
plt.show()
```

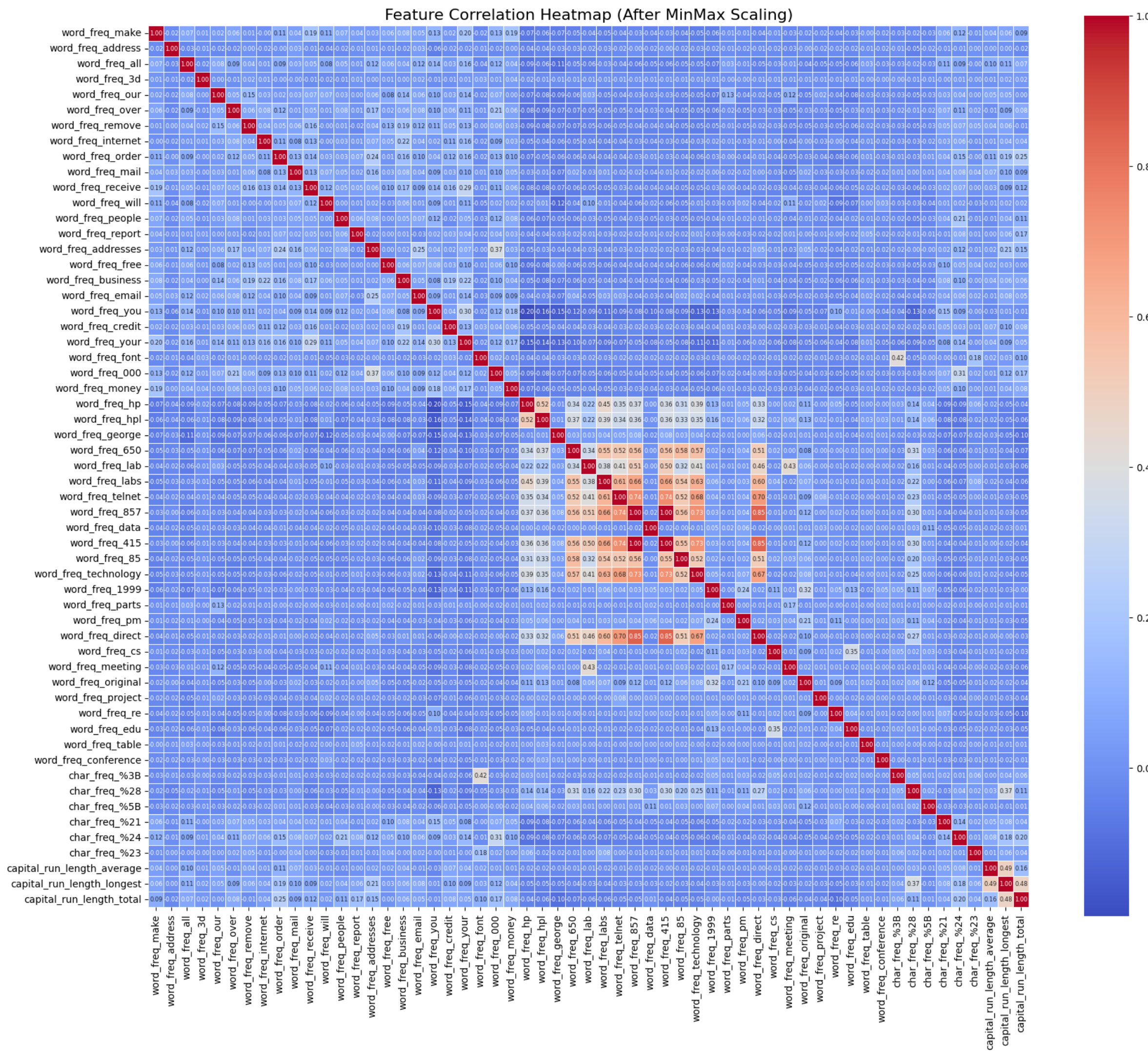




Correlation HeatMap

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    matthews_corrcoef, roc_curve, auc, classification_report
)

# Assuming X_scaled_df is your scaled DataFrame
corr_matrix = X_scaled_df.corr()

plt.figure(figsize=(18, 15))  # Bigger figure
sns.heatmap(
    corr_matrix,
    cmap='coolwarm',
    square=True,
    annot=True,               # Show values inside squares
    fmt=".2f",                # Format to 2 decimal places
```

```
    linewidths=0.5,          # Thin grid lines
    annot_kws={"size": 6}    # Smaller font size
)

plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.title("Feature Correlation Heatmap (After MinMax Scaling)", fontsize=16)
plt.tight_layout()
plt.show()
```



Choosing best k value using Grid Search CV

```
# First split into train+val and test
X_train_val, X_test, y_train_val, y_test = train_test_split(
    X_scaled, y, test_size=0.2, stratify=y, random_state=42
)

# Then split train+val into train and validation
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.25, stratify=y_train_val, random_stat
)  # 0.25 x 0.8 = 0.2 validation

# ===============================
# 2️⃣ GridSearchCV to find best k
# ===============================
param_grid = {'n_neighbors': range(1, 21)}
grid = GridSearchCV(
    estimator=KNeighborsClassifier(),
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)
grid.fit(X_train, y_train)

print(f"✅ Best k found by GridSearchCV: {grid.best_params_['n_neighbors']}")
print(f"Best CV Accuracy: {grid.best_score_:.4f}")
```

```
✅ Best k found by GridSearchCV: 1
Best CV Accuracy: 0.8902
```

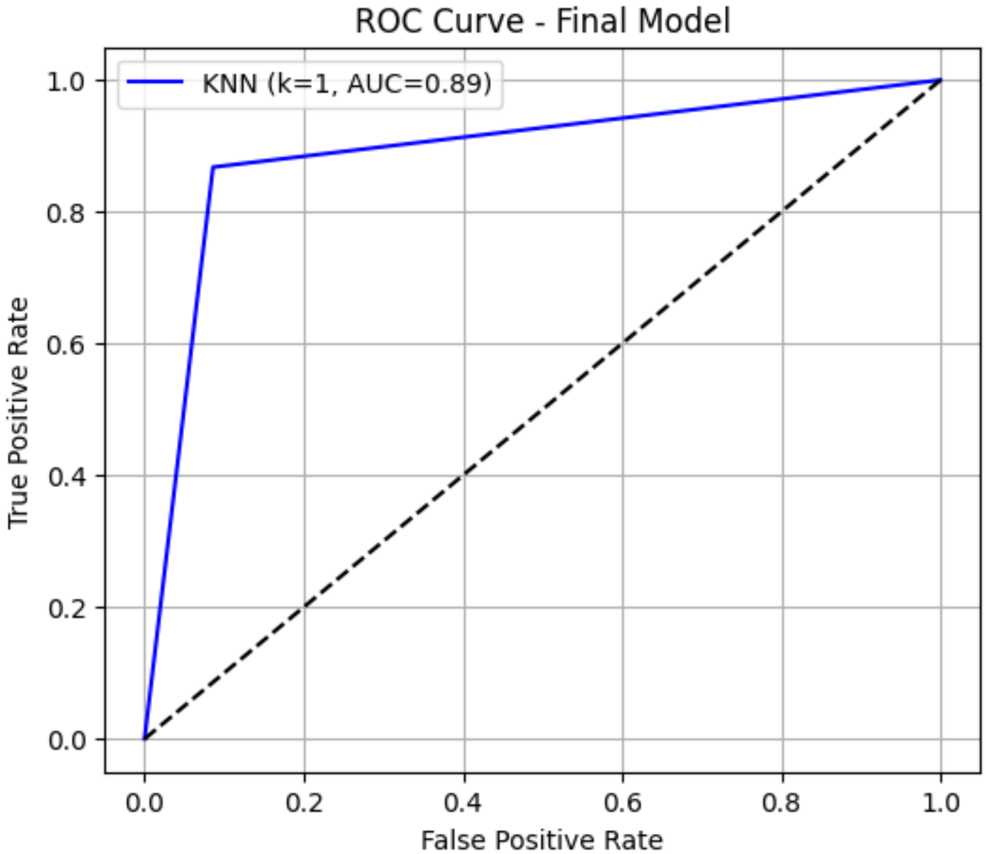
Model Training and Evaluation

```
# 3️⃣ Retrain on Train+Validation with Best k
# ===============================
best_k = grid.best_params_['n_neighbors']
final_model = KNeighborsClassifier(n_neighbors=best_k)
final_model.fit(X_train_val, y_train_val)
```

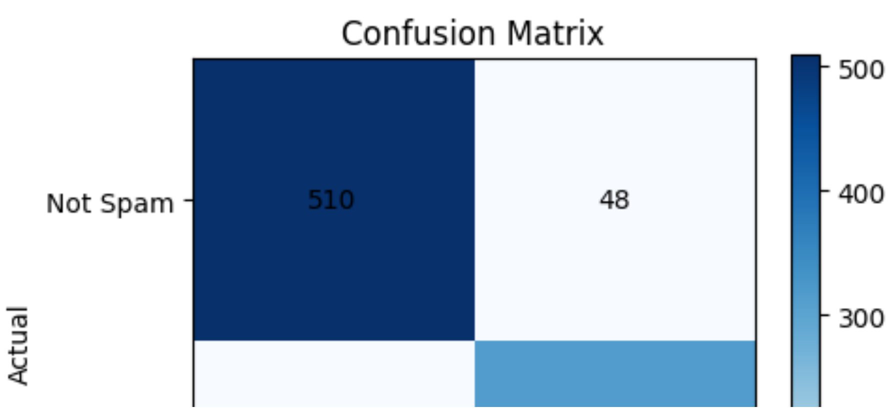
```
# ================================
# 4️⃣ Evaluate on Test Set
# ================================
y_pred = final_model.predict(X_test)
y_prob = final_model.predict_proba(X_test)[:, 1]

acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred, zero_division=0)
rec = recall_score(y_test, y_pred, zero_division=0)
f1 = f1_score(y_test, y_pred, zero_division=0)
mcc = matthews_corrcoef(y_test, y_pred)

fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

print("\n📊 Final Model Performance on Test Set")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1 Score : {f1:.4f}")
print(f"MCC      : {mcc:.4f}")
print(f"AUC      : {roc_auc:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, zero_division=0))

# ================================
# 5️⃣ Plot ROC Curve
# ================================
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, label=f"KNN (k={best_k}, AUC={roc_auc:.2f})", color='blue')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Final Model")
plt.legend()
plt.grid(True)
plt.show()
```

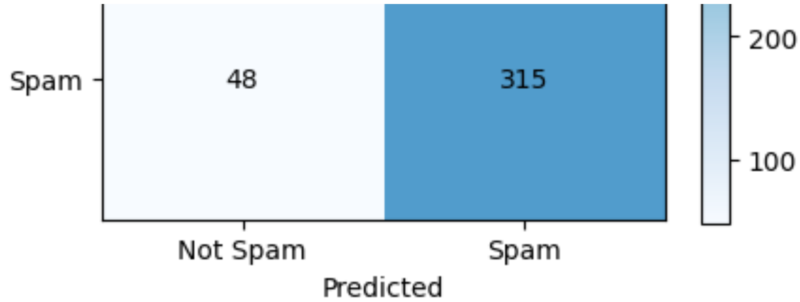
```
📊 Final Model Performance on Test Set
Accuracy : 0.8958
Precision: 0.8678
Recall   : 0.8678
F1 Score : 0.8678
MCC      : 0.7817
AUC      : 0.8909

Classification Report:
              precision    recall  f1-score   support

         0.0       0.91      0.91      0.91       558
         1.0       0.87      0.87      0.87       363

    accuracy                           0.90       921
   macro avg       0.89      0.89      0.89       921
weighted avg       0.90      0.90      0.90       921
```

```
weighted avg       0.90      0.90      0.90       921
```




```
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(5, 4))
plt.imshow(conf_matrix, cmap='Blues')
plt.title('Confusion Matrix')
plt.colorbar()
plt.xticks([0, 1], ['Not Spam', 'Spam'])
plt.yticks([0, 1], ['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
for i in range(2):
    for j in range(2):
        plt.text(j, i, conf_matrix[i, j], ha='center', va='center', color='blac
plt.tight_layout()
plt.show()
```





KD Tree and Ball Tree

```
# Use KDTree
model_kd = KNeighborsClassifier(n_neighbors=best_k, algorithm='kd_tree')
model_kd.fit(X_train, y_train)

# Use BallTree
model_ball = KNeighborsClassifier(n_neighbors=best_k, algorithm='ball_tree')
model_ball.fit(X_train, y_train)
```

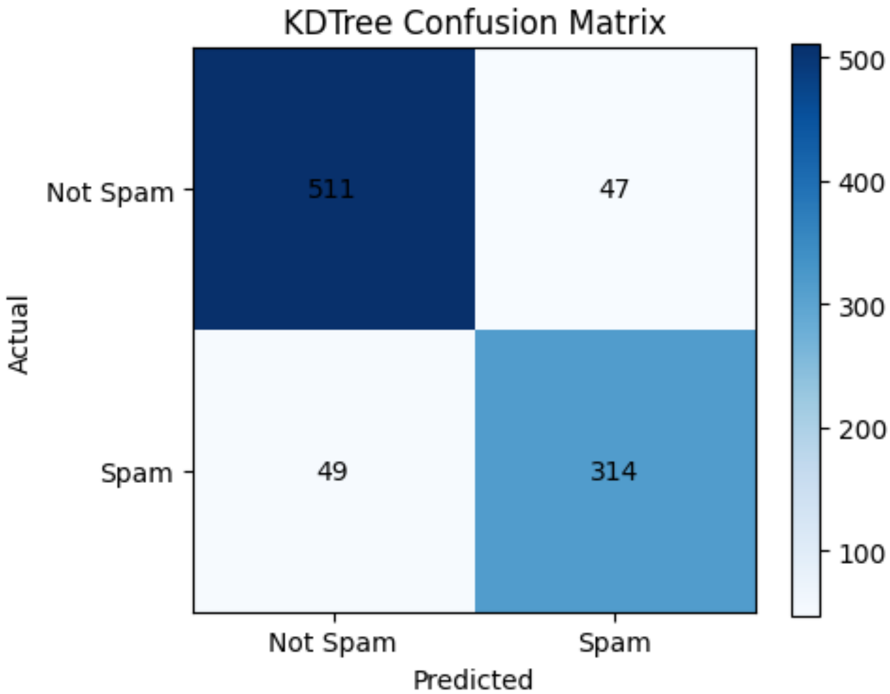
```
▾ KNeighborsClassifier
KNeighborsClassifier(algorithm='ball_tree', n_neighbors=1)
```

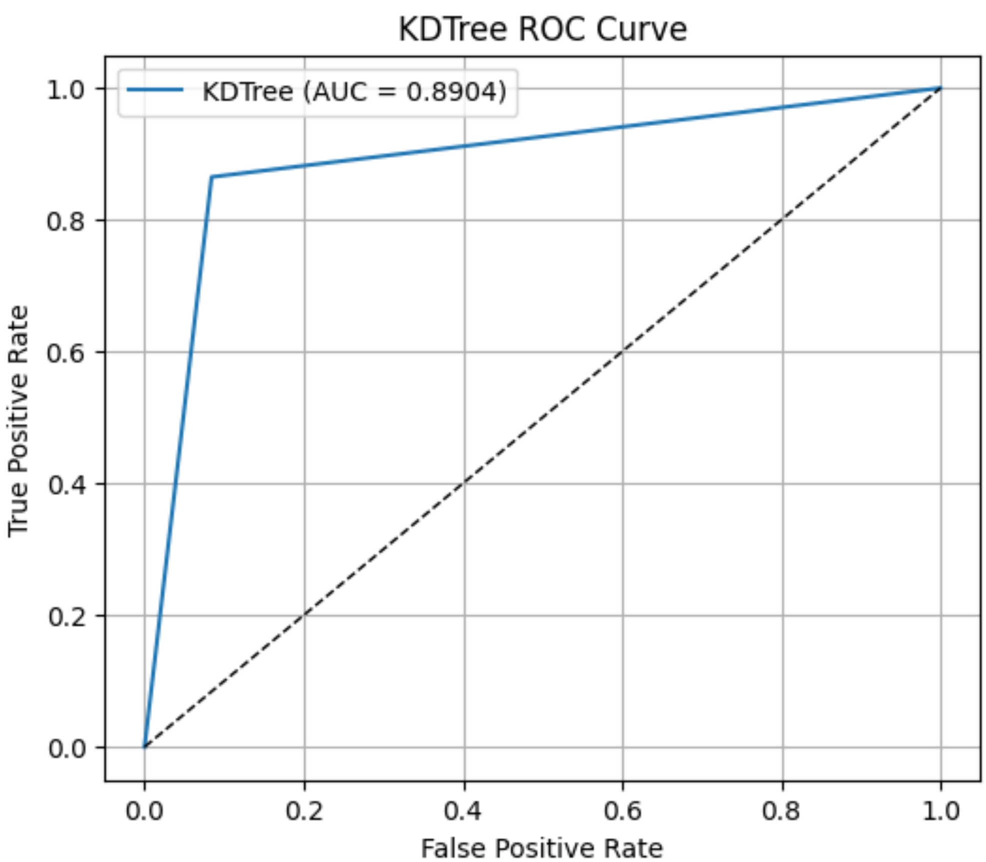
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_curve, auc,
    classification_report, matthews_corrcoef
)
import matplotlib.pyplot as plt

# Helper function to evaluate a model
def evaluate_knn_model(name, model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    mcc = matthews_corrcoef(y_test, y_pred)
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    roc_auc = auc(fpr, tpr)

    print(f"\n🔍 {name} Performance Metrics:")
    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall   : {rec:.4f}")
    print(f"F1 Score : {f1:.4f}")
```

```
    print(f"MCC       : {mcc:.4f}")
    print("\nClassification Report:\n", classification_report(y_test, y_pred))

    # Confusion Matrix
    conf = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(5, 4))
    plt.imshow(conf, cmap='Blues')
    plt.title(f'{name} Confusion Matrix')
    plt.colorbar()
    plt.xticks([0, 1], ['Not Spam', 'Spam'])
    plt.yticks([0, 1], ['Not Spam', 'Spam'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    for i in range(2):
        for j in range(2):
            plt.text(j, i, conf[i, j], ha='center', va='center')
    plt.tight_layout()
    plt.show()

    # ROC Curve
    plt.figure(figsize=(6, 5))
    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.4f})')
    plt.plot([0, 1], [0, 1], 'k--', linewidth=1)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{name} ROC Curve')
    plt.legend()
    plt.grid(True)
    plt.show()

# Train-test split assumed done:
# X_train, X_test, y_train, y_test already available
# best_k is selected

# Evaluate KDTree
knn_kd = KNeighborsClassifier(n_neighbors=best_k, algorithm='kd_tree')
evaluate_knn_model("KDTree", knn_kd, X_train, X_test, y_train, y_test)

# Evaluate BallTree
knn_ball = KNeighborsClassifier(n_neighbors=best_k, algorithm='ball_tree')
evaluate_knn_model("BallTree", knn_ball, X_train, X_test, y_train, y_test)
```

```
🔍 KDTree Performance Metrics:
Accuracy : 0.8958
Precision: 0.8698
Recall   : 0.8650
F1 Score : 0.8674
MCC      : 0.7815

Classification Report:
               precision    recall  f1-score   support

         0.0       0.91      0.92      0.91       558
```

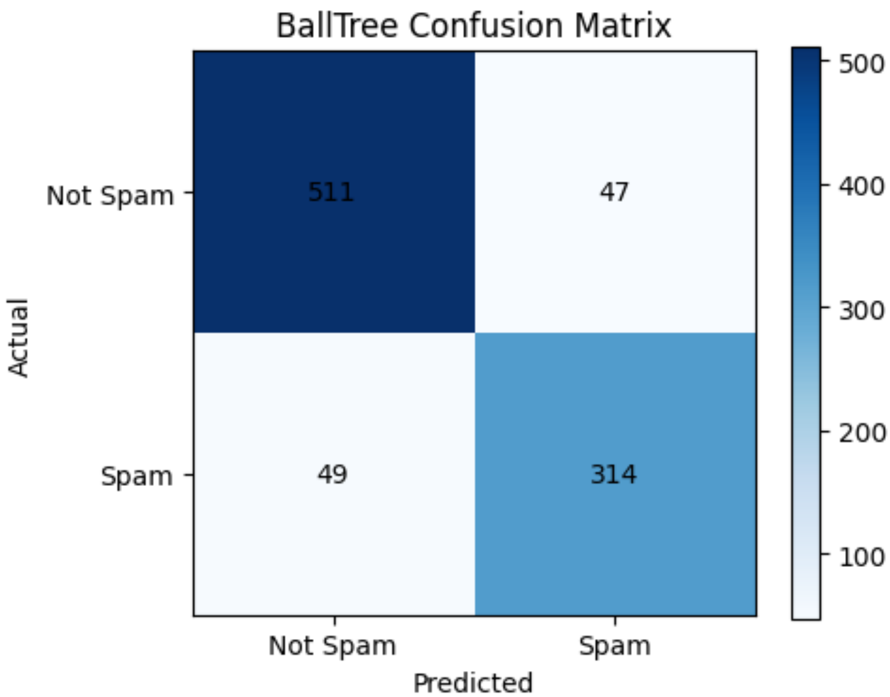
```
         1.0       0.87      0.87      0.87       363

    accuracy                           0.90       921
   macro avg       0.89      0.89      0.89       921
weighted avg       0.90      0.90      0.90       921
```

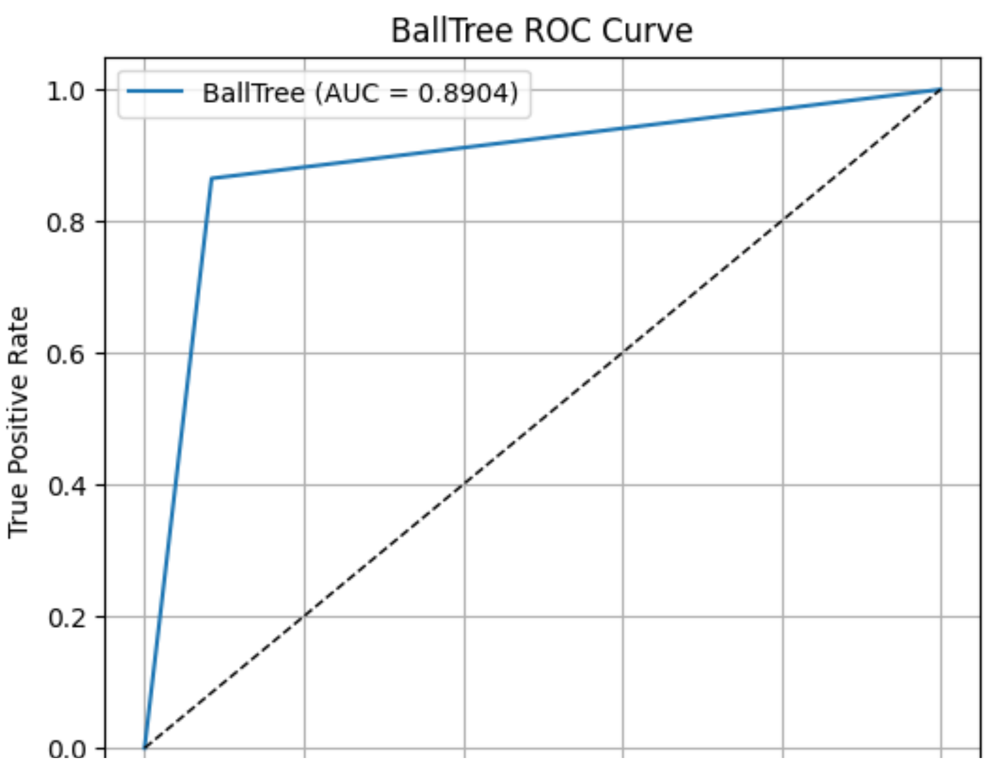


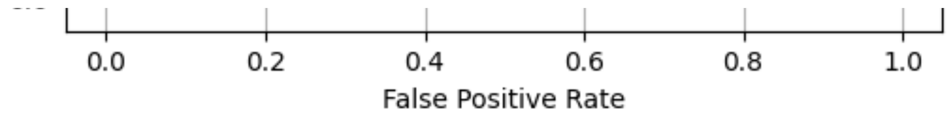



```
🔍 BallTree Performance Metrics:
Accuracy : 0.8958
Precision: 0.8698
```

```
Recall    : 0.8650
F1 Score : 0.8674
MCC       : 0.7815

Classification Report:
              precision    recall  f1-score   support

         0.0       0.91      0.92      0.91       558
         1.0       0.87      0.87      0.87       363

    accuracy                           0.90       921
   macro avg       0.89      0.89      0.89       921
weighted avg       0.90      0.90      0.90       921
```








```
import time

for algo in ['kd_tree', 'ball_tree', 'brute']:
    model = KNeighborsClassifier(n_neighbors=best_k, algorithm=algo)
    start = time.time()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    duration = time.time() - start
    acc = accuracy_score(y_test, y_pred)
    print(f"{algo:10} → Accuracy: {acc:.4f}, Time: {duration:.4f} sec")
```

```
kd_tree    → Accuracy: 0.8958, Time: 0.1874 sec
ball_tree  → Accuracy: 0.8958, Time: 0.2061 sec
brute      → Accuracy: 0.8958, Time: 0.0436 sec
```

6 svm Code:

Importing Required Libraries

```
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
from sklearn.preprocessing import StandardScaler  # or MinMaxScaler, if used
from sklearn.preprocessing import LabelEncoder  # if categorical labels
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_re
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import cross_val_score, KFold,GridSearchCV
from sklearn.model_selection import cross_val_score, StratifiedKFold
```

Loading dataset

```
df = pd.read_csv('spambase.csv')
```

Basic info

```
print("Dataset Info:\n", df.info())
print("\nFirst 5 rows:\n", df.head())
print("\nMissing Values:\n", df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 58 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
 16  word_freq_business          4601 non-null   float64
```

```
 17  word_freq_email             4601 non-null   float64
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
 20  word_freq_your              4601 non-null   float64
 21  word_freq_font              4601 non-null   float64
 22  word_freq_000               4601 non-null   float64
 23  word_freq_money             4601 non-null   float64
 24  word_freq_hp                4601 non-null   float64
 25  word_freq_hpl               4601 non-null   float64
 26  word_freq_george            4601 non-null   float64
 27  word_freq_650               4601 non-null   float64
 28  word_freq_lab               4601 non-null   float64
 29  word_freq_labs              4601 non-null   float64
 30  word_freq_telnet            4601 non-null   float64
 31  word_freq_857               4601 non-null   float64
 32  word_freq_data              4601 non-null   float64
 33  word_freq_415               4601 non-null   float64
 34  word_freq_85                4601 non-null   float64
 35  word_freq_technology        4601 non-null   float64
 36  word_freq_1999              4601 non-null   float64
 37  word_freq_parts             4601 non-null   float64
 38  word_freq_pm                4601 non-null   float64
 39  word_freq_direct            4601 non-null   float64
 40  word_freq_cs                4601 non-null   float64
 41  word_freq_meeting           4601 non-null   float64
 42  word_freq_original          4601 non-null   float64
 43  word_freq_project           4601 non-null   float64
 44  word_freq_re                4601 non-null   float64
 45  word_freq_edu               4601 non-null   float64
 46  word_freq_table             4601 non-null   float64
 47  word_freq_conference        4601 non-null   float64
 48  char_freq_%3B               4601 non-null   float64
 49  char_freq_%28               4601 non-null   float64
 50  char_freq_%5B               4601 non-null   float64
 51  char_freq_%21               4601 non-null   float64
 52  char_freq_%24               4601 non-null   float64
```

Handling missing values

```
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

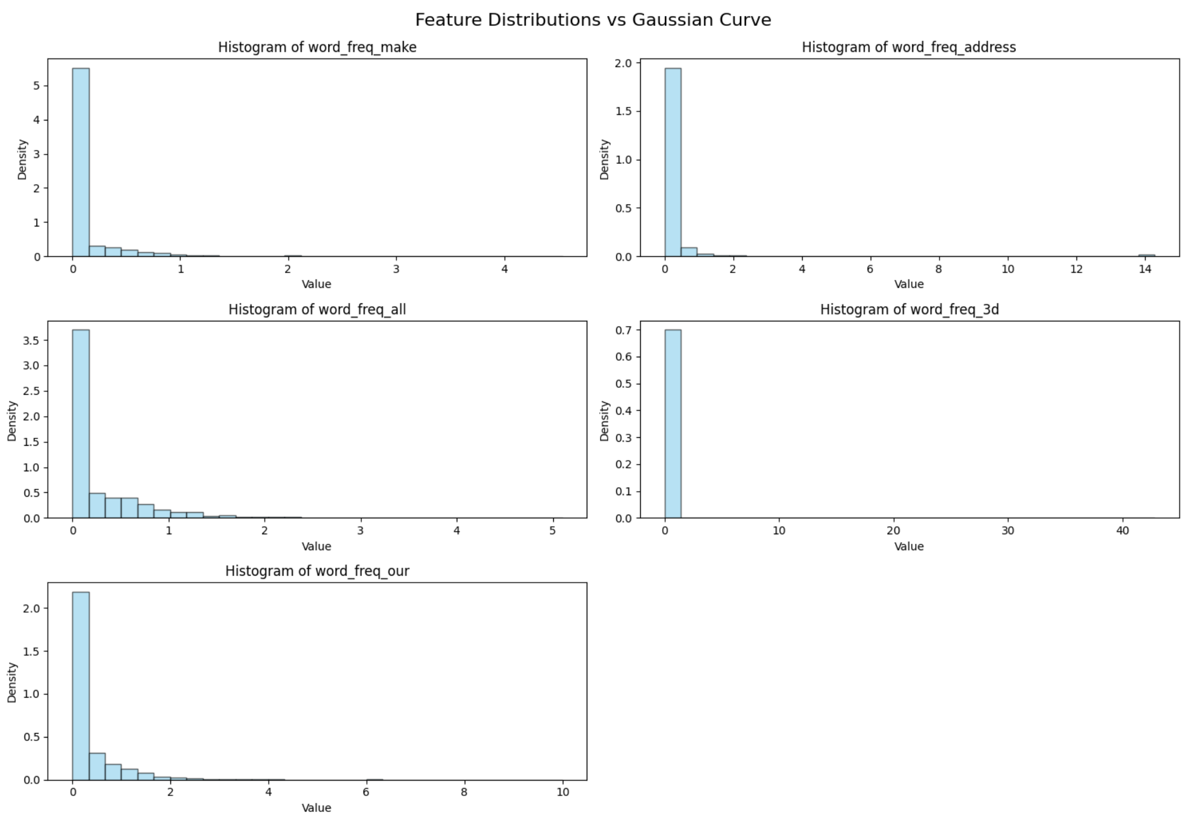
Splitting of feature and target

```
X = df_imputed.drop('class', axis=1)
y = df_imputed['class']
```

Checking Distribution

```
X = df.drop('class', axis=1)  # Assuming 'class' is the target

# Choose a few features to visualize
```

```
# Choose a few features to visualize
sample_features = X.columns[:5]  # First 5 features

# Plot histograms with Gaussian curve overlay
plt.figure(figsize=(15, 10))
for i, feature in enumerate(sample_features):
    plt.subplot(3, 2, i + 1)
    data = X[feature]

    # Plot histogram
    count, bins, ignored = plt.hist(data, bins=30, density=True, alpha=0.6, col

    # Plot normal distribution curve
    '''mu, std = data.mean(), data.std()
    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax, 100)
    p = 1 / (std * np.sqrt(2 * np.pi)) * np.exp(-(x - mu)**2 / (2 * std**2))
    plt.plot(x, p, 'r', linewidth=2)'''

    plt.title(f'Histogram of {feature}')
    plt.xlabel('Value')
    plt.ylabel('Density')

plt.tight_layout()
plt.suptitle('Feature Distributions vs Gaussian Curve', fontsize=16, y=1.02)
plt.show()
```



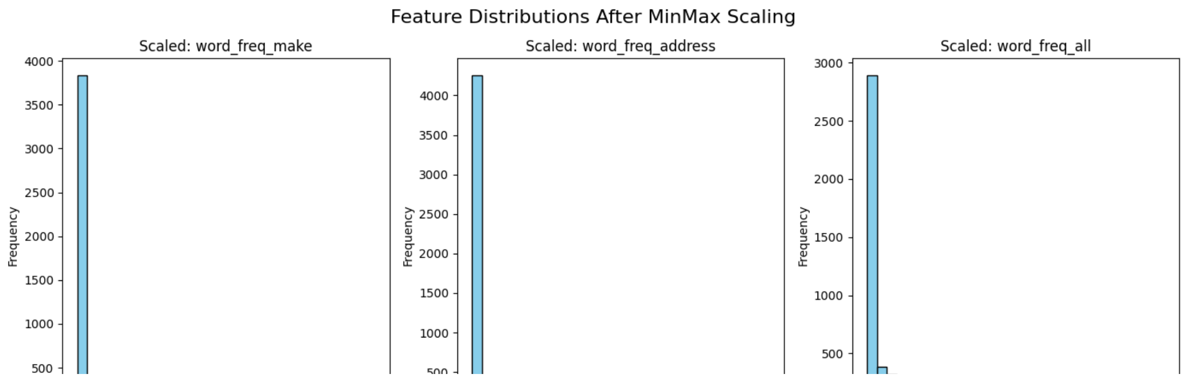
Applying min max scaling

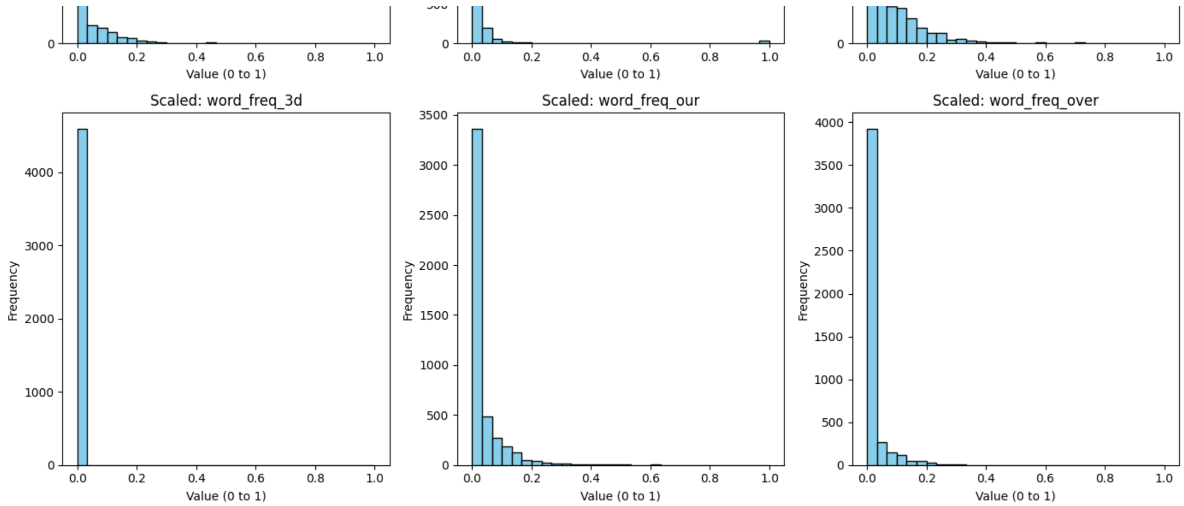
```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Plots

Histogram

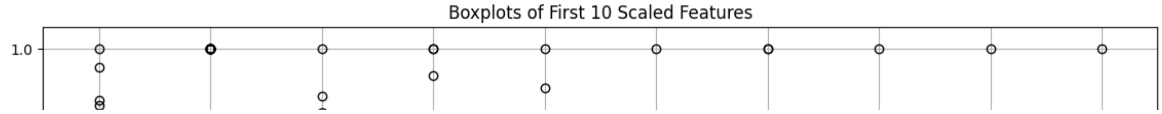
```
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 10))
for i in range(6):  # first 6 features as an example
    plt.subplot(2, 3, i + 1)
    plt.hist(X_scaled[:, i], bins=30, color='skyblue', edgecolor='black')
    plt.title(f'Scaled: {X.columns[i]}')
    plt.xlabel('Value (0 to 1)')
    plt.ylabel('Frequency')
plt.tight_layout()
plt.suptitle('Feature Distributions After MinMax Scaling', fontsize=16, y=1.02)
plt.show()
```

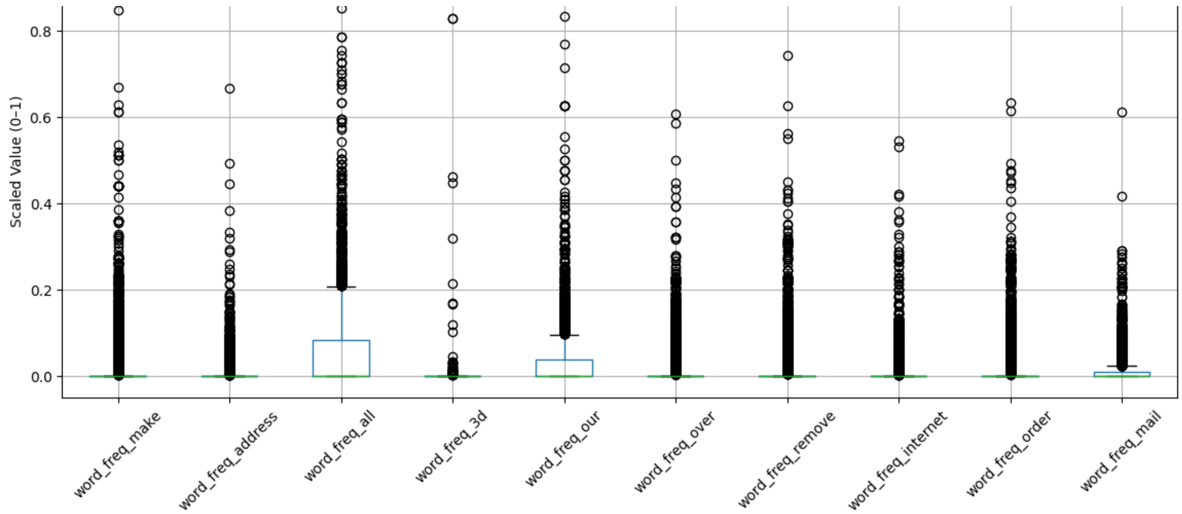




Boxplot

```
import pandas as pd

# Convert scaled array back to DataFrame for easier plotting
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

plt.figure(figsize=(14, 6))
X_scaled_df.iloc[:, :10].boxplot(rot=45)
plt.title('Boxplots of First 10 Scaled Features')
plt.ylabel('Scaled Value (0–1)')
plt.grid(True)
plt.show()
```

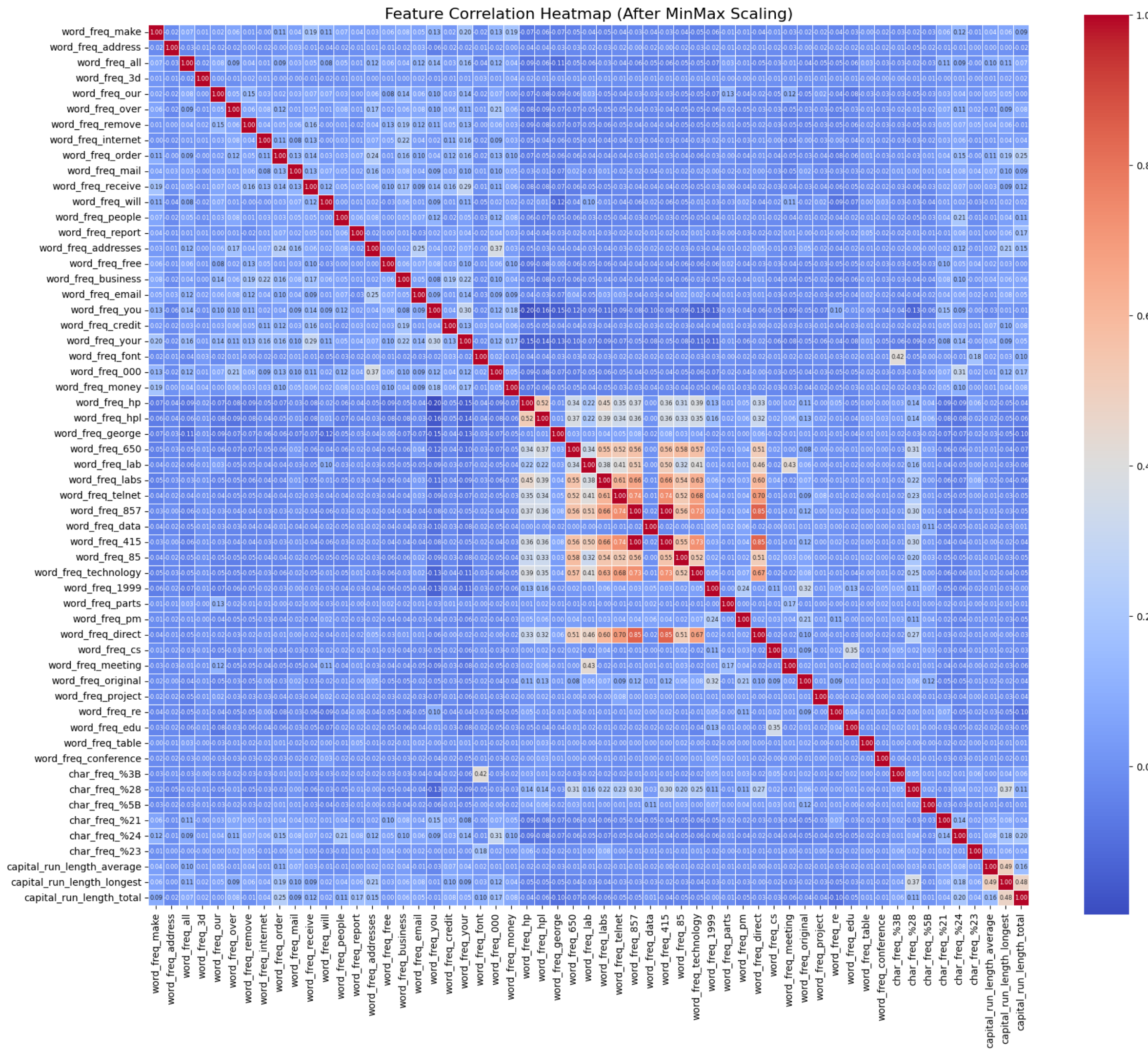




Correlation HeatMap

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Assuming X_scaled_df is your scaled DataFrame
corr_matrix = X_scaled_df.corr()

plt.figure(figsize=(18, 15))  # Bigger figure
sns.heatmap(
    corr_matrix,
    cmap='coolwarm',
    square=True,
    annot=True,               # Show values inside squares
    fmt=".2f",                # Format to 2 decimal places
    linewidths=0.5,           # Thin grid lines
    annot_kws={"size": 6}     # Smaller font size
)

plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.title("Feature Correlation Heatmap (After MinMax Scaling)", fontsize=16)
```

```
plt.tight_layout()
plt.show()
```



Model Training

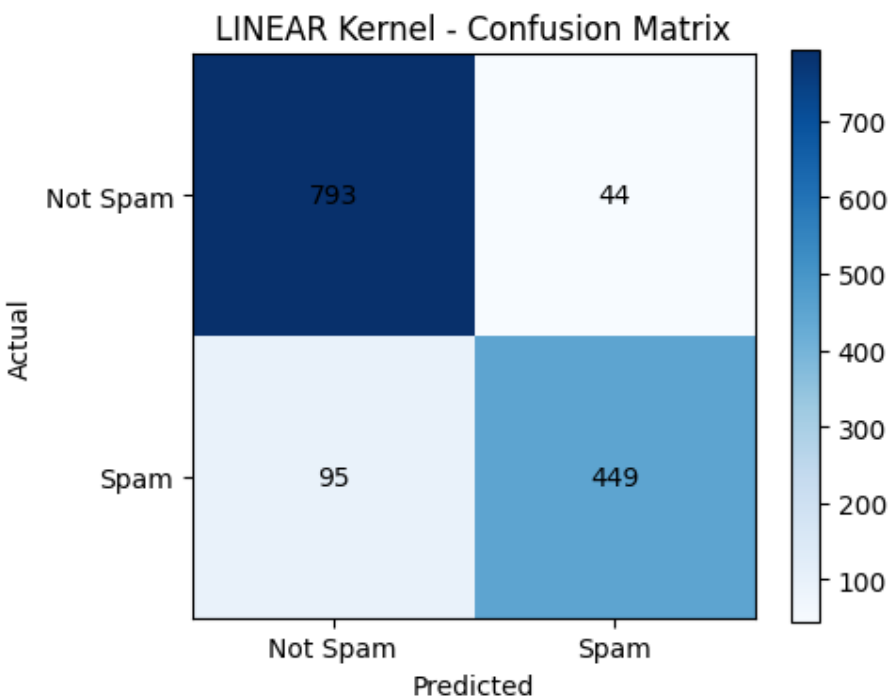
```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_curve, auc,
    classification_report
)

# Split the data again (or reuse your earlier split)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, stratify=y, random_state=42
)
```

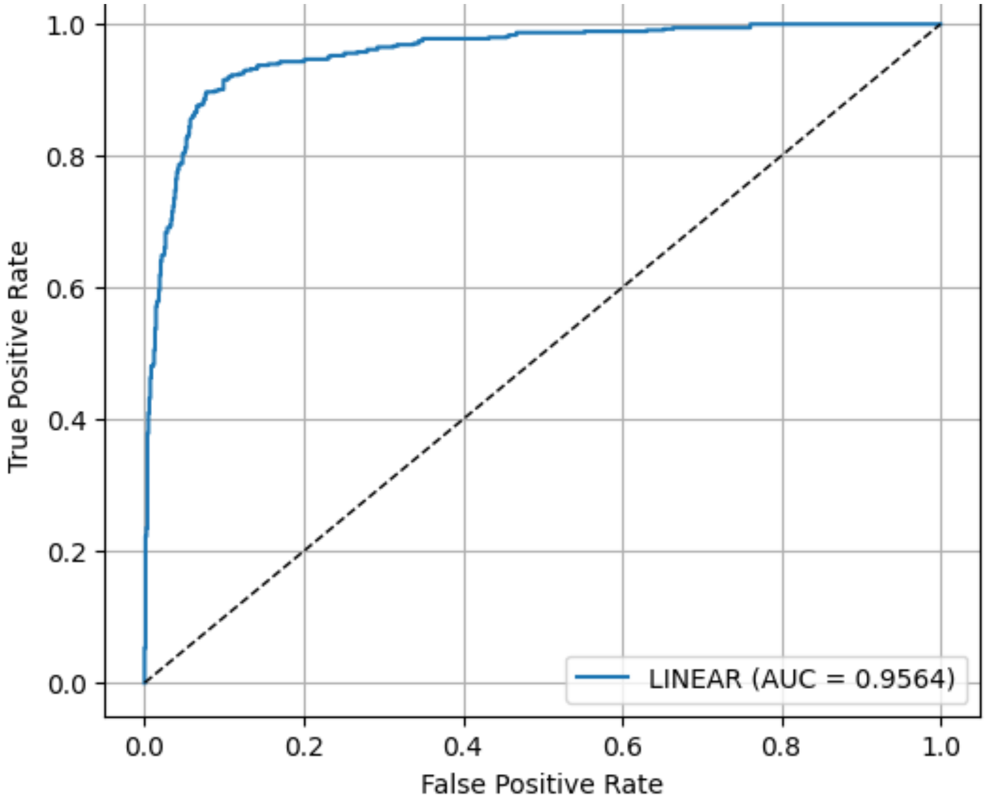
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_curve, auc
)


# Your dataset X (features) and y (labels) must already be defined
pipeline = make_pipeline(
    MinMaxScaler(),
    SVC(kernel='rbf', C=10, gamma='scale')
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='accuracy')

for i, acc in enumerate(scores, 1):
    print(f"Fold {i}: Accuracy = {acc:.4f}")
print(f"➡️ Mean Accuracy: {np.mean(scores):.4f} ± {np.std(scores):.4f}")

# Kernels to evaluate
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
results = []

def evaluate_svm(kernel):
    print(f"\n🔍 Evaluating SVM with {kernel.upper()} kernel...")

    # Create and train SVM model
    model = SVC(kernel=kernel, probability=True, random_state=42)
```

```
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]  # for ROC

    # Metrics
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall   : {rec:.4f}")
    print(f"F1 Score : {f1:.4f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Confusion Matrix
    conf = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(5, 4))
    plt.imshow(conf, cmap='Blues')
    plt.title(f'{kernel.upper()} Kernel - Confusion Matrix')
    plt.colorbar()
    plt.xticks([0, 1], ['Not Spam', 'Spam'])
    plt.yticks([0, 1], ['Not Spam', 'Spam'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    for i in range(2):
        for j in range(2):
            plt.text(j, i, conf[i, j], ha='center', va='center')
    plt.tight_layout()
    plt.show()

    # ROC Curve
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(6, 5))
    plt.plot(fpr, tpr, label=f'{kernel.upper()} (AUC = {roc_auc:.4f})')
    plt.plot([0, 1], [0, 1], 'k--', linewidth=1)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{kernel.upper()} Kernel - ROC Curve')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Store results
    results.append({
        "Kernel": kernel.capitalize(),
        "Accuracy": acc,
        "Precision": prec,
```

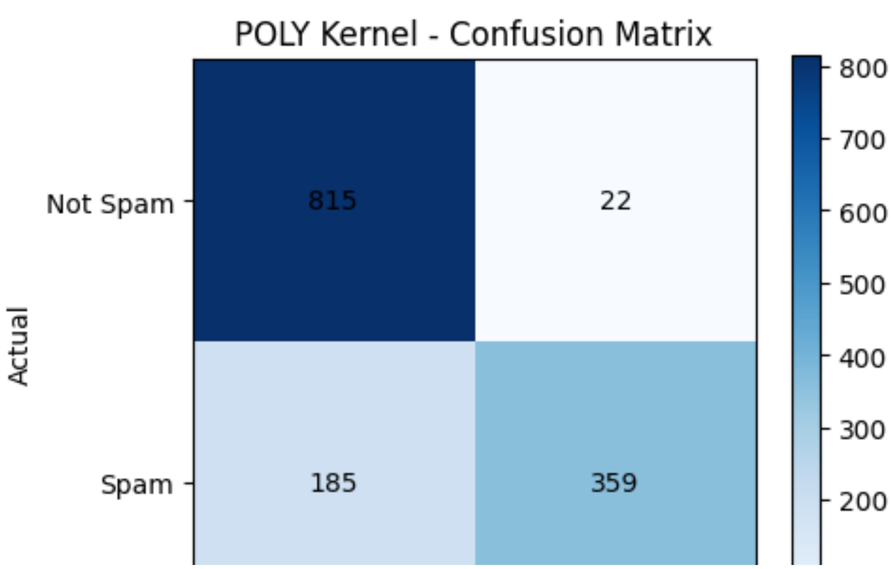
```
            "Recall": rec,
            "F1 Score": f1,
            "AUC": roc_auc
        })

# Run evaluations for all kernels
for kernel in kernels:
    evaluate_svm(kernel)
```

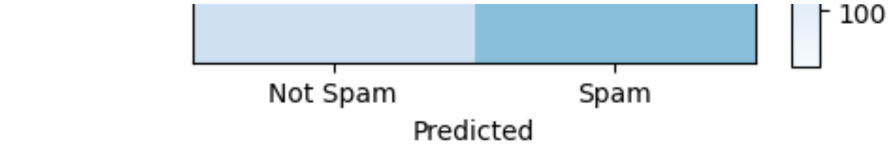
```
Fold 1: Accuracy = 0.9359
Fold 2: Accuracy = 0.9250
Fold 3: Accuracy = 0.9402
Fold 4: Accuracy = 0.9337
Fold 5: Accuracy = 0.9326
➡️ Mean Accuracy: 0.9335 ± 0.0050

🔍 Evaluating SVM with LINEAR kernel...
Accuracy : 0.8993
Precision: 0.9108
Recall   : 0.8254
F1 Score : 0.8660

Classification Report:
              precision    recall  f1-score   support

         0.0       0.89      0.95      0.92       837
         1.0       0.91      0.83      0.87       544

    accuracy                           0.90      1381
   macro avg       0.90      0.89      0.89      1381
weighted avg       0.90      0.90      0.90      1381
```

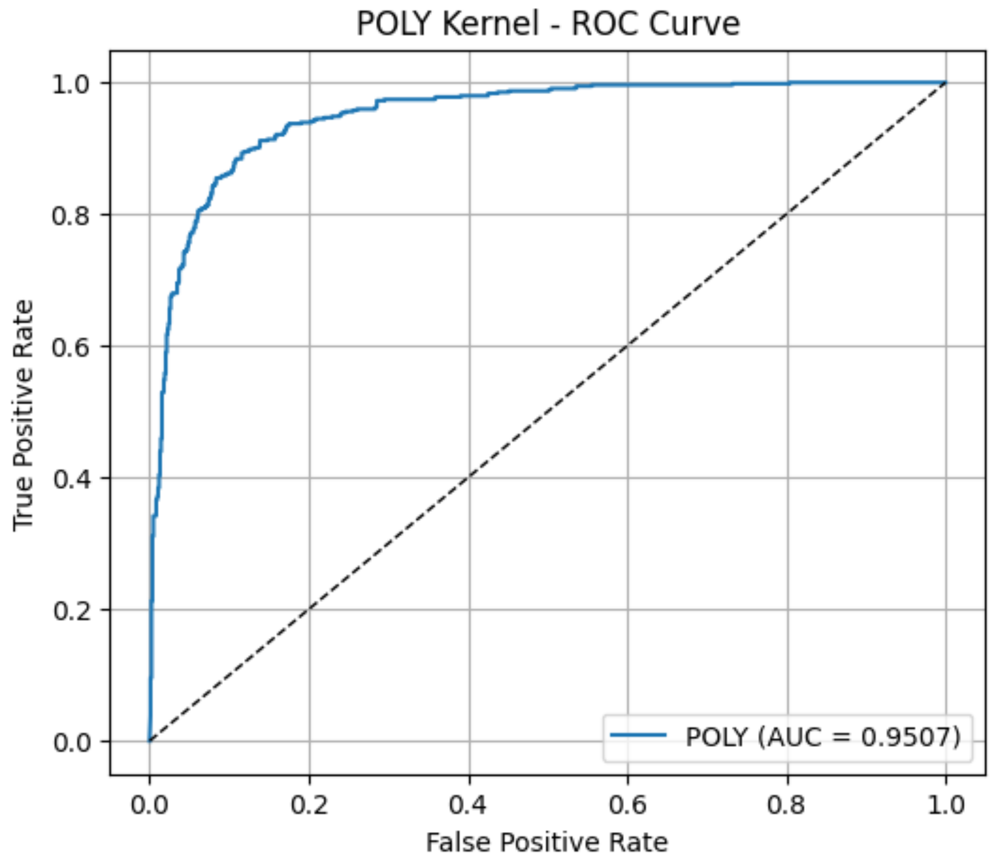




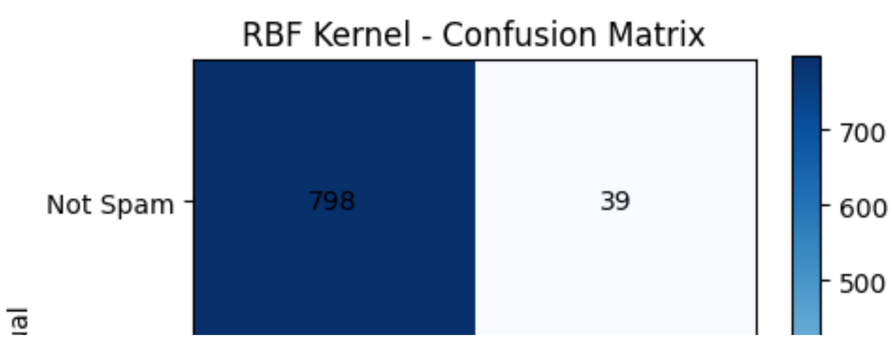

```
🔍 Evaluating SVM with POLY kernel...
Accuracy : 0.8501
Precision: 0.9423
Recall   : 0.6599
F1 Score : 0.7762

Classification Report:
              precision    recall  f1-score   support

         0.0       0.81      0.97      0.89       837
         1.0       0.94      0.66      0.78       544

    accuracy                           0.85      1381
   macro avg       0.88      0.82      0.83      1381
weighted avg       0.87      0.85      0.84      1381
```

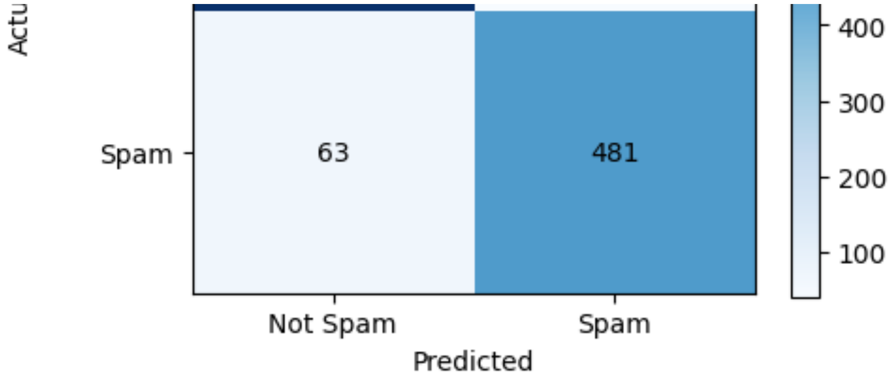


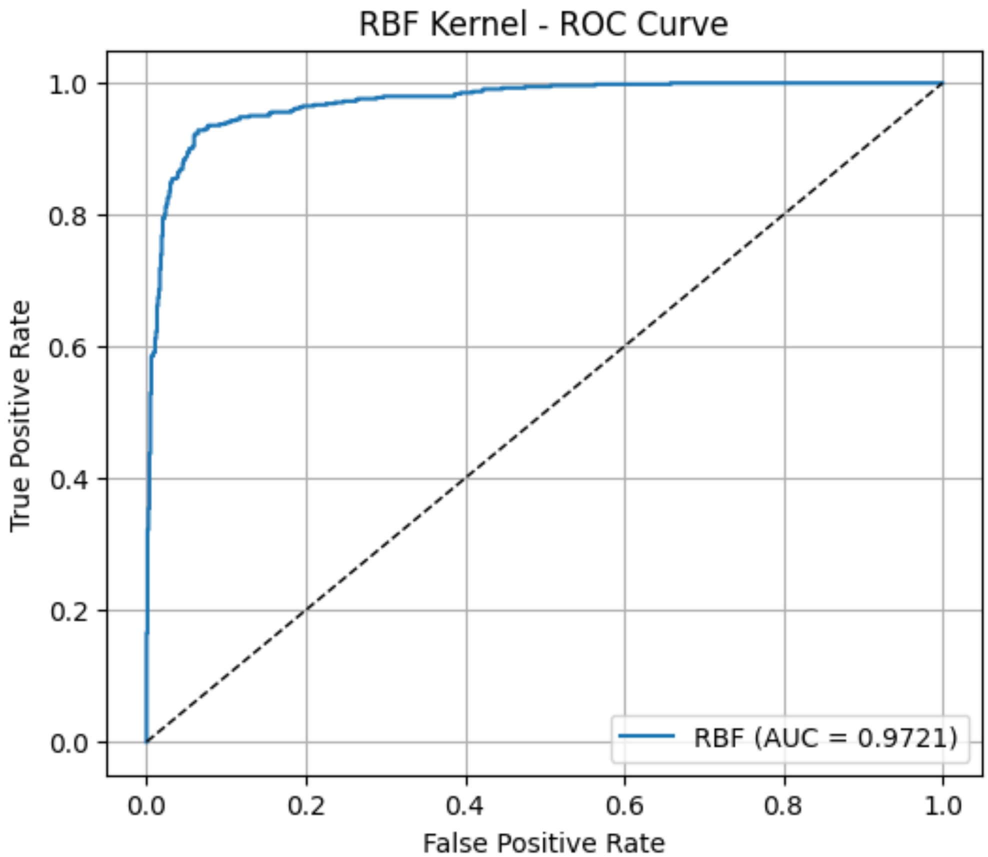




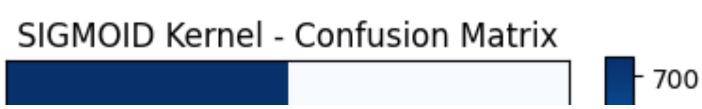

```
🔍 Evaluating SVM with RBF kernel...
Accuracy : 0.9261
Precision: 0.9250
Recall   : 0.8842
F1 Score : 0.9041

Classification Report:
              precision    recall  f1-score   support

         0.0       0.93      0.95      0.94       837
         1.0       0.93      0.88      0.90       544

    accuracy                           0.93      1381
   macro avg       0.93      0.92      0.92      1381
weighted avg       0.93      0.93      0.93      1381
```

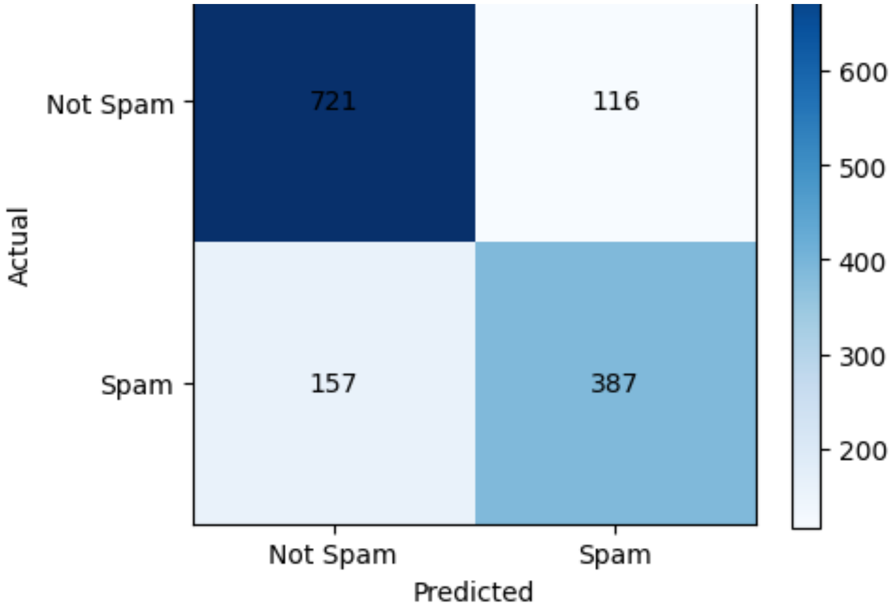


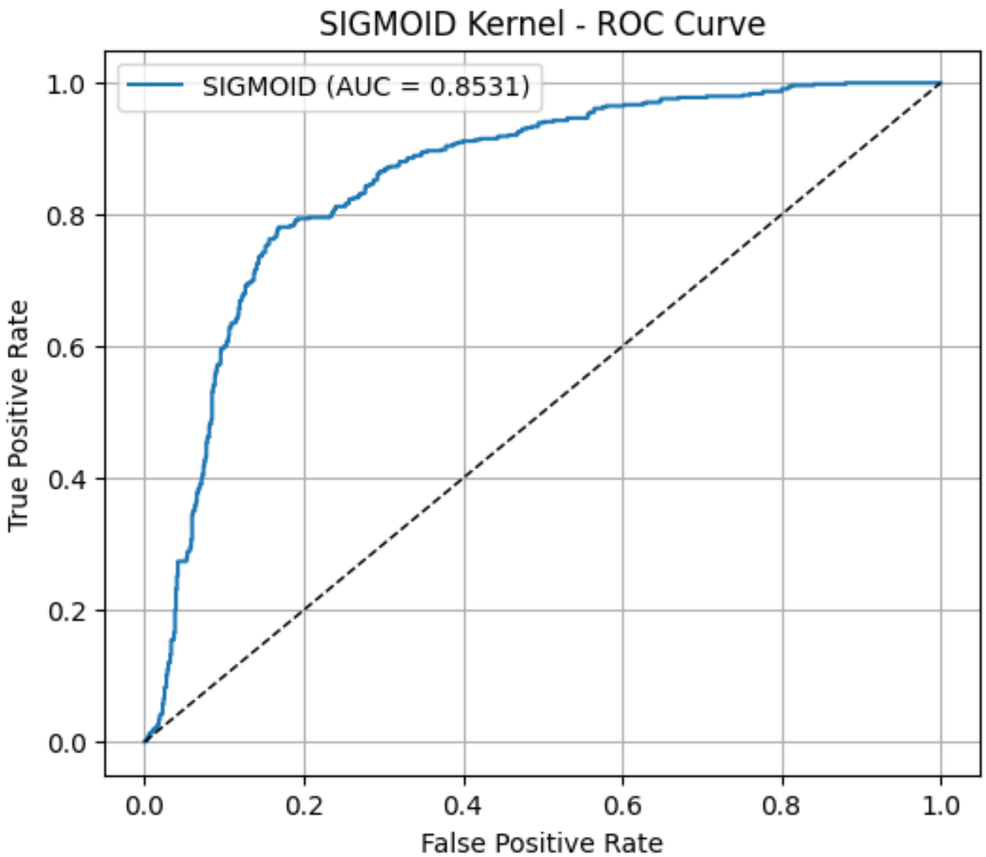





```
🔍 Evaluating SVM with SIGMOID kernel...
Accuracy : 0.8023
Precision: 0.7694
Recall   : 0.7114
F1 Score : 0.7393

Classification Report:
              precision    recall  f1-score   support

         0.0       0.82      0.86      0.84       837
         1.0       0.77      0.71      0.74       544

    accuracy                           0.80      1381
   macro avg       0.80      0.79      0.79      1381
weighted avg       0.80      0.80      0.80      1381
```








```
# Display tabular summary
print("\n📊 SVM Kernel Comparison Summary:\n")
df_results = pd.DataFrame(results)
print(df_results.to_string(index=False))
```

```
📊 SVM Kernel Comparison Summary:

 Kernel  Accuracy  Precision   Recall  F1 Score      AUC
 Linear  0.899348   0.910751 0.825368  0.865959 0.956425
```

```
       Poly  0.850109   0.942257 0.659926  0.776216 0.950749
        Rbf  0.926140   0.925000 0.884191  0.904135 0.972150
    Sigmoid  0.802317   0.769384 0.711397  0.739255 0.853074
```

```python
svm = SVC()

param_grid = {
    'C': [0.1, 1, 10],
    'gamma': ['scale', 0.01, 0.001],
    'kernel': ['linear', 'rbf', 'poly']
}

# 5-fold CV
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Grid Search
grid = GridSearchCV(estimator=svm, param_grid=param_grid, cv=cv, scoring='accur
grid.fit(X_train, y_train)

# Best Parameters
print("🔍 Best Parameters Found:")
print(grid.best_params_)

# Best Estimator
best_svm = grid.best_estimator_

# Evaluate on test set
y_pred = best_svm.predict(X_test)

print("\n📊 Classification Report (Test Set):")
print(classification_report(y_test, y_pred))

print("✅ Test Accuracy:", accuracy_score(y_test, y_pred))
```

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits
🔍 Best Parameters Found:
{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}

📊 Classification Report (Test Set):
              precision    recall  f1-score   support

         0.0       0.93      0.96      0.94       837
         1.0       0.93      0.88      0.91       544

    accuracy                           0.93      1381
   macro avg       0.93      0.92      0.92      1381
weighted avg       0.93      0.93      0.93      1381

✅ Test Accuracy: 0.9275887038377987
```

```python
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
```

```
from sklearn.model_selection import StratifiedKFold, cross_val_score
pipeline = make_pipeline(
    MinMaxScaler(),
    SVC(kernel='rbf', C=10, gamma='scale')
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='accuracy')

for i, acc in enumerate(scores, 1):
    print(f"Fold {i}: Accuracy = {acc:.4f}")
print(f"➡️ Mean Accuracy: {np.mean(scores):.4f} ± {np.std(scores):.4f}")
```

```
Fold 1: Accuracy = 0.9359
Fold 2: Accuracy = 0.9250
Fold 3: Accuracy = 0.9402
Fold 4: Accuracy = 0.9337
Fold 5: Accuracy = 0.9326
➡️ Mean Accuracy: 0.9335 ± 0.0050
```

```
import time

kernels = ['linear', 'poly', 'rbf', 'sigmoid']
results = []

for kernel in kernels:
    params = {'kernel': kernel, 'C': 10.0, 'probability': True, 'random_state':

    # Add kernel-specific hyperparameters
    if kernel == 'poly':
        params.update({'degree': 3, 'gamma': 'scale'})
    elif kernel in ['rbf', 'sigmoid']:
        params.update({'gamma': 'scale'})

    model = SVC(**params)

    start = time.time()
    model.fit(X_train, y_train)
    end = time.time()

    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    results.append({
        "Kernel": kernel.capitalize(),
        "Hyperparameters": str({k: v for k, v in params.items() if k != 'kernel
        "Accuracy": round(acc, 4),
        "F1 Score": round(f1, 4),
        "Training Time (s)": round(end - start, 3)
    })

# Create and display final table
```

```
df_table4 = pd.DataFrame(results)
print("\nTable 4: SVM Performance with Different Kernels and Parameters")
print(df_table4.to_string(index=False))
```

```
Table 4: SVM Performance with Different Kernels and Parameters
 Kernel
 Linear                                {'C': 10.0, 'probability': True, 'ra
   Poly {'C': 10.0, 'probability': True, 'random_state': 42, 'degree': 3, '
    Rbf              {'C': 10.0, 'probability': True, 'random_state': 42, '
Sigmoid              {'C': 10.0, 'probability': True, 'random_state': 42, '
```

K Fold Cross Validation

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler  # or StandardScaler
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.svm import SVC
import numpy as np

# RBF kernel SVM
svm_rbf = make_pipeline(MinMaxScaler(), SVC(kernel='rbf', C=10.0, gamma='scale'))

# 5-fold stratified CV
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(svm_rbf, X, y, cv=cv, scoring='accuracy')

# Print each fold and mean
for i, acc in enumerate(scores, 1):
    print(f"Fold {i}: Accuracy = {acc:.4f}")
print(f"\n➡️ Mean Accuracy: {np.mean(scores):.4f} ± {np.std(scores):.4f}")
```

```
Fold 1: Accuracy = 0.9359
Fold 2: Accuracy = 0.9250
Fold 3: Accuracy = 0.9402
Fold 4: Accuracy = 0.9337
Fold 5: Accuracy = 0.9326

➡️ Mean Accuracy: 0.9335 ± 0.0050
```

7 Comparision Tables

Table 1: Performance Comparison of Naïve Bayes Variants

Model	Accuracy	Precision	Recall	F1 Score
BernoulliNB	0.8899	0.8843	0.8290	0.8558
MultinomialNB	0.8950	0.9424	0.7813	0.8543
GaussianNB	0.8197	0.7001	0.9485	0.8056

Table 2: KNN Performance for Different k Values

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

Table 3: KNN Comparison: KDTree vs BallTree

Metric	KDTree	BallTree
Accuracy	0.8899	0.8899
Precision	0.8551	0.8551
Recall	0.8676	0.8676
F1 Score	0.8613	0.8613
Training Time (s)	0.4470	0.4058

Table 4: Cross-Validation Scores for Each Model (K = 5)

Fold	Naïve Bayes Accuracy	KNN Accuracy (k=1)	SVM Accuracy
Fold 1	0.8719	0.9034	0.9359
Fold 2	0.8935	0.9076	0.9250
Fold 3	0.8891	0.9152	0.9402
Fold 4	0.8913	0.9000	0.9337
Fold 5	0.8859	0.8935	0.9326
Average	**0.8863**	**0.9039**	**0.9335 ± 0.0050**

Table 5: SVM Performance with Different Kernels and Parameters

Kernel	**Hyperparameters**	**Accuracy**	**F1 score**	**Training time**
linear	c=10.0	8.993	8.660	0.12
polynomial	c=10.0,degree=3,gamma=scale	0.8501	0.7762	0.35
RBF	c=10.0,gamma=scale	0.9261	0.9041	0.20
sigmoid	c=10.0,gamma=scale	0.8023	0.7393	0.18

8 Observation:

- KNN with k=1 achieved the highest accuracy consistently across all folds, indicating strong capability in classifying email spam versus ham.
- Naive Bayes classifiers, particularly MultinomialNB, provided stable and competitive results, showing robustness in handling text data with varying feature distributions.
- SVM, especially with the RBF kernel (C=10, gamma=`scale`), achieved the highest overall accuracy ($\approx$ 92.7) and F1 score among all kernels, demonstrating its strong capability in handling complex, non-linear decision boundaries in text data.
- Although KNN attained better peak performance, Naive Bayes models required less training time and are more scalable for larger datasets.
- The choice between KNN, Naive Bayes, and SVM depends on the trade-off between accuracy, computational efficiency, and dataset size for the specific application scenario.

GitHub Repository: https://github.com/Thamizhmathibharathi/project.git

9 Conclusion:

- The experiment demonstrated that KNN (k=1) outperforms Naive Bayes variants in certain folds, but SVM with RBF kernel consistently delivered the best overall performance in terms of accuracy and F1 score on the email classification task.
- Naive Bayes remains valuable due to its simplicity, fast training, and effectiveness on high-dimensional, sparse data typical in text classification, while SVM offers robust performance with non-linear patterns but requires slightly more computational resources.
- For deployment:
  - Naive Bayes is suitable for quick predictions on large datasets.
  - KNN is ideal when highest accuracy is needed and resources allow.
  - SVM is a balanced choice when accuracy and robustness against complex patterns are critical, with moderate training time.