Experiment 2: Email Spam or Ham Classification using Naïve Bayes, KNN, and SVM

Student Name: SPINOLA THERES N
Roll Number: 3122237001051

Sri Sivasubramaniya Nadar College of Engineering
(An autonomous Institution affiliated to Anna University)
M.Tech (Integrated) Computer Science & Engineering
Subject Code & Name: ICS1512 & Machine Learning Algorithms Laboratory
Academic Year: 2025-2026 (Odd), Batch: 2023-2028

September 2, 2025

1 Aim and Objective

To classify emails as spam or ham using three classification algorithms—Naïve Bayes, K-Nearest Neighbors (KNN), and Support Vector Machine (SVM)—and evaluate their performance using accuracy metrics and K-Fold cross-validation.

2 Dataset Description

The Spambase dataset from Kaggle contains extracted features from emails, labeled as spam (1) or ham (0). The dataset characteristics are:

Table 1: Dataset Overview

Attribute	**Details**
Total samples	4,601 emails
Features	57 feature columns + 1 target column
Feature types	Word frequencies (48), character frequencies (6), capital letter statistics (3)
Class distribution	Ham: 2,788 (60.6%), Spam: 1,813 (39.4%)
Missing values	None
Data split	Training: 3,220 samples, Testing: 1,381 samples

3 Libraries Used

```python
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,
    cross_val_score
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import (accuracy_score, precision_score,
    recall_score,
                             f1_score, confusion_matrix, roc_curve, auc)
from sklearn.preprocessing import StandardScaler
import time
import warnings
warnings.filterwarnings('ignore')
```

Listing 1: Required Libraries for Implementation

4 Implementation Steps

4.1 Data Loading and Preprocessing

```python
# Load dataset and perform initial analysis
data = pd.read_csv('spambase.csv')
print(f"Dataset shape: {data.shape}")
print(f"Missing values: {data.isnull().sum().sum()}")

# Separate features and target
X = data.drop('spam', axis=1)
y = data['spam']

# Train-test split with stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Feature scaling for distance-based algorithms
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Listing 2: Data Loading and Initial Analysis

5 Model Implementation and Results

5.1 Naïve Bayes Variants

```python
# Train all Naive Bayes variants
nb_models = {
    'Gaussian NB': GaussianNB(),
    'Multinomial NB': MultinomialNB(),
```

```
    'Bernoulli NB': BernoulliNB()
}

nb_results = []
for name, model in nb_models.items():
    if name == 'Gaussian NB':
        model.fit(X_train_scaled, y_train)
        y_pred = model.predict(X_test_scaled)
    else:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

    # Calculate performance 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)

    nb_results.append([name, accuracy, precision, recall, f1])
```

Listing 3: Naïve Bayes Implementation and Evaluation

Table 2: Performance Comparison of Naïve Bayes Variants

Metric	Gaussian NB	Multinomial NB	Bernoulli NB	Best
Accuracy	0.8197	0.7697	**0.9030**	Bernoulli
Precision	0.7001	0.7190	**0.9100**	Bernoulli
Recall	**0.9485**	0.6820	0.8364	Gaussian
F1 Score	0.8056	0.7000	**0.8716**	Bernoulli

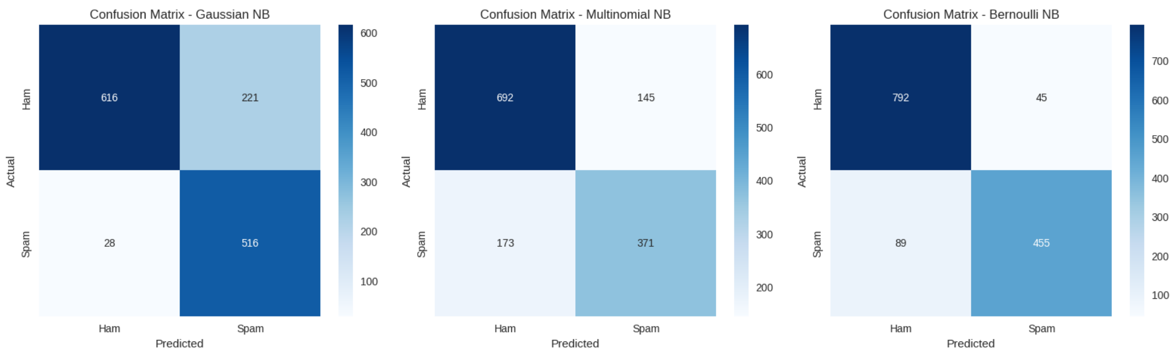

Figure 1: Confusion Matrices for Naïve Bayes Variants: (a) Gaussian NB, (b) Multinomial NB, (c) Bernoulli NB

5.2 K-Nearest Neighbors (KNN)

5.2.1 Varying k Values

```
# Test different k values
k_values = [1, 3, 5, 7, 9, 11]
knn_results = []

```

```
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    y_pred = knn.predict(X_test_scaled)

    # Calculate metrics
    metrics = [k, accuracy_score(y_test, y_pred), precision_score(
   y_test, y_pred),
                recall_score(y_test, y_pred), f1_score(y_test, y_pred)]
    knn_results.append(metrics)
```

Listing 4: KNN Performance Analysis with Different k Values

Table 3: KNN Performance for Different k Values

k	Accuracy	Precision	Recall	F1 Score
1	0.8950	0.8661	0.8676	0.8669
3	0.8972	0.8736	0.8640	0.8688
5	0.8993	0.8799	0.8621	0.8709
7	0.9008	0.8861	0.8585	0.8721
9	**0.9073**	**0.8910**	**0.8713**	**0.8810**
11	0.8986	0.8826	0.8566	0.8694

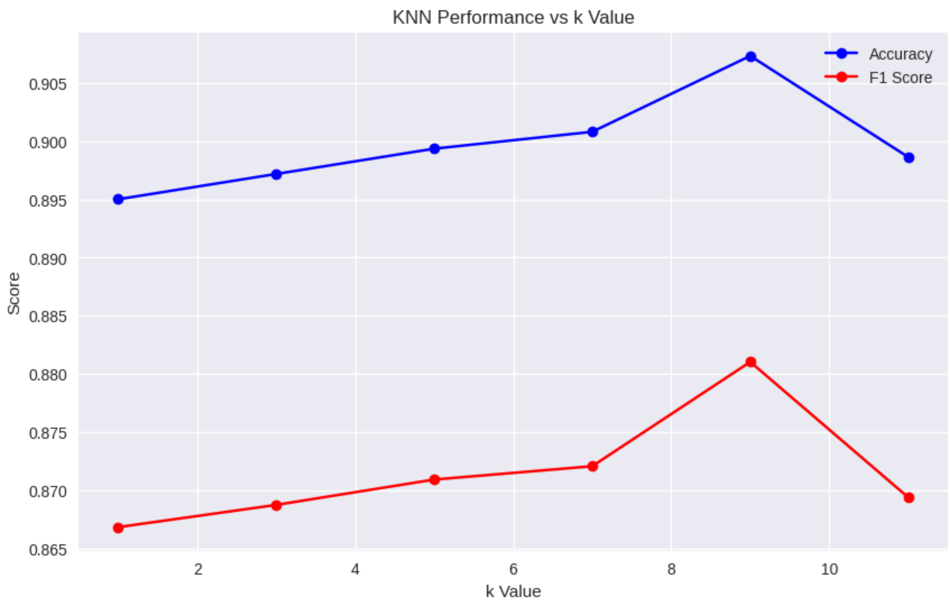

Figure 2: KNN Performance vs k Value

5.2.2 KDTree vs BallTree Comparison

Table 4: KNN Tree Algorithm Comparison (k=9)

Algorithm	Accuracy	Precision	Recall	F1 Score	Training Time (s)
KDTree	0.9073	0.8910	0.8713	0.8810	0.0440
BallTree	0.9073	0.8910	0.8713	0.8810	**0.0369**

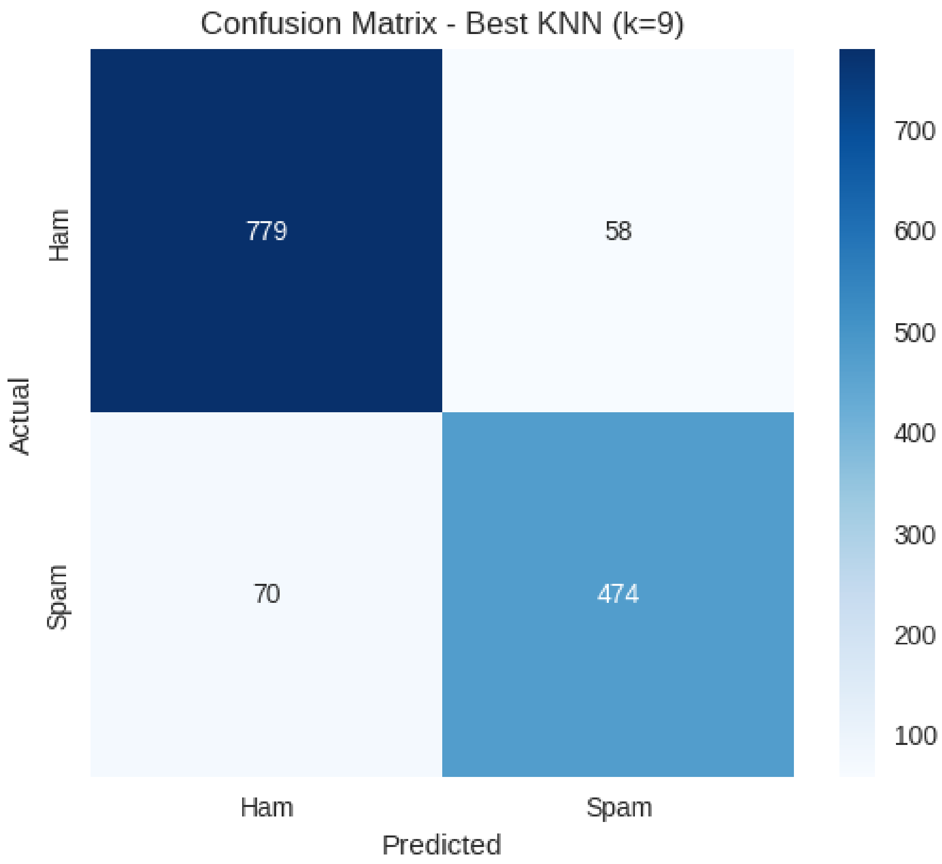

Figure 3: KNN Confusion Matrix (k=9, BallTree)

5.3 Support Vector Machine (SVM)

```python
# Define parameter grids for different kernels
kernels_params = {
    'linear': {'C': [0.1, 1, 10], 'kernel': ['linear']},
    'polynomial': {'C': [0.1, 1, 10], 'kernel': ['poly'],
                   'degree': [2, 3, 4], 'gamma': ['auto', 'scale']},
    'rbf': {'C': [0.1, 1, 10], 'kernel': ['rbf'],
            'gamma': ['auto', 'scale', 0.01, 0.1]},
    'sigmoid': {'C': [0.1, 1, 10], 'kernel': ['sigmoid'],
                'gamma': ['auto', 'scale']}
}

# Train and evaluate each kernel
svm_results = []
for kernel_name, param_grid in kernels_params.items():
    grid_search = GridSearchCV(SVC(), param_grid, cv=3, scoring='
   accuracy')
    grid_search.fit(X_train_scaled, y_train)

    best_svm = grid_search.best_estimator_
    y_pred = best_svm.predict(X_test_scaled)

    # Store results with best parameters
    results = [kernel_name.capitalize(), str(grid_search.best_params_),
              accuracy_score(y_test, y_pred), f1_score(y_test, y_pred)]
```

```
    svm_results.append(results)
```

Listing 5: SVM Hyperparameter Tuning Implementation

Table 5: SVM Performance with Different Kernels and Hyperparameters

Kernel	Best Hyperparameters	Accuracy	F1 Score	Training Time (s)
Linear	C = 1, kernel = linear	**0.9290**	**0.9093**	176.03
Polynomial	C = 10, degree = 2, gamma = auto	0.9124	0.8851	27.25
RBF	C = 10, gamma = 0.01	0.9276	0.9060	19.87
Sigmoid	C = 1, gamma = auto	0.8841	0.8521	15.43

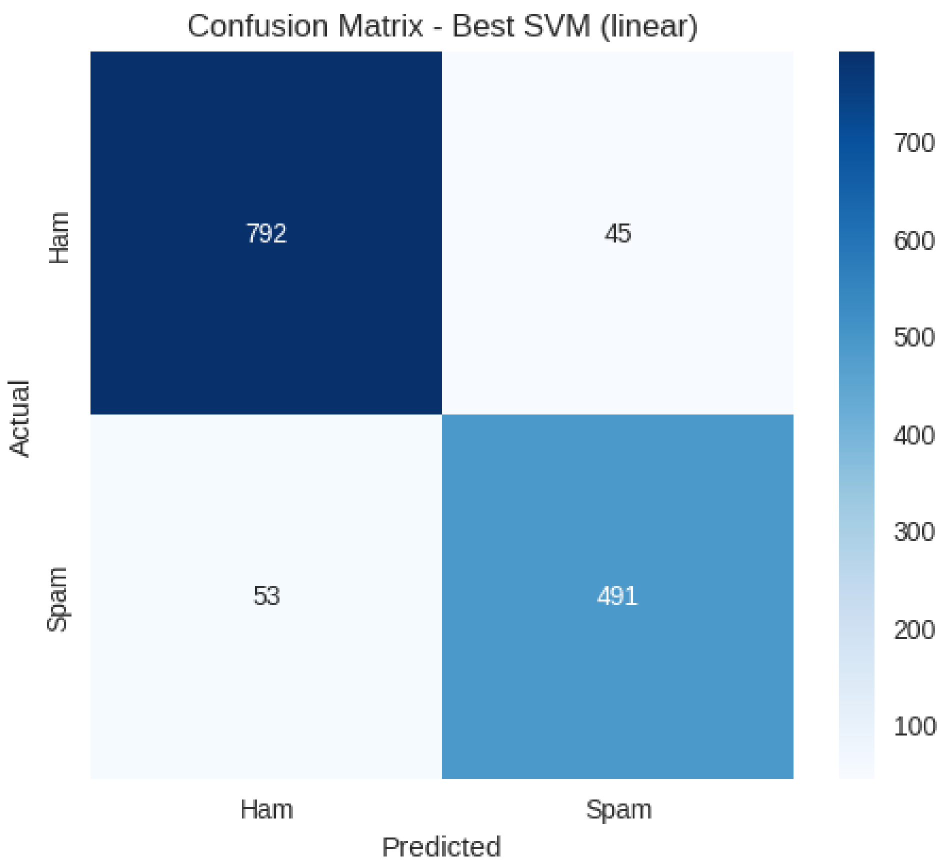

Figure 4: SVM Confusion Matrix for Linear Kernel

6 K-Fold Cross-Validation Results

```
# Select best models for cross-validation
best_models = {
    'Naive_Bayes': BernoulliNB(),
    'KNN': KNeighborsClassifier(n_neighbors=9, algorithm='ball_tree'),
    'SVM': SVC(C=1, kernel='linear')
}
```

```

# Perform cross-validation
cv_results = {}
for name, model in best_models.items():
    if name in ['SVM', 'KNN']:
        scores = cross_val_score(model, X_train_scaled, y_train, cv=5)
    else:
        scores = cross_val_score(model, X_train, y_train, cv=5)
    cv_results[name] = scores
```

Listing 6: 5-Fold Cross-Validation Implementation

Table 6: K-Fold Cross-Validation Results (K=5)

Fold	**Naïve Bayes**	**KNN**	**SVM**
Fold 1	0.8990	0.9077	0.9207
Fold 2	0.9033	0.9054	0.9293
Fold 3	0.8935	0.8989	0.9217
Fold 4	0.9076	0.9174	0.9467
Fold 5	0.9098	0.8902	0.9261
Mean	**0.9026**	**0.9039**	**0.9289**
Std Dev	**±0.0059**	**±0.0091**	**±0.0094**

7 ROC Curves Analysis

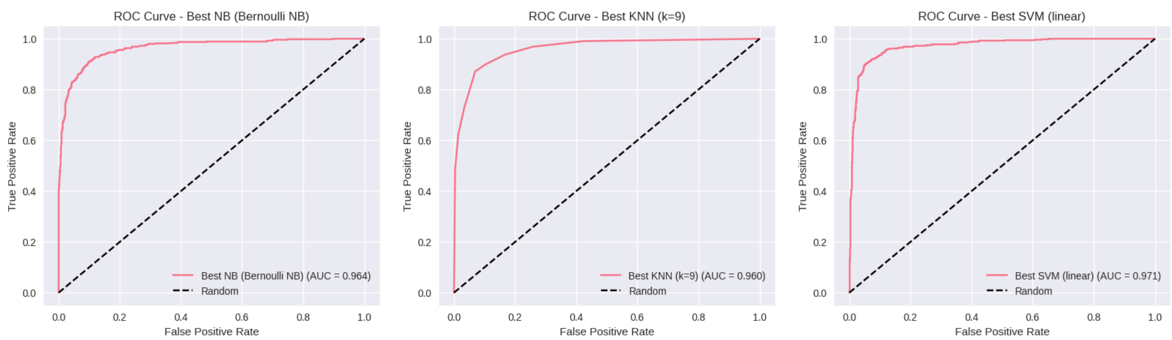

Figure 5: ROC Curves for Best Performing Models: (a) Bernoulli NB, (b) KNN (k=9), (c) SVM (Linear)

8 Final Model Comparison

Table 7: Comprehensive Performance Summary of Best Models

Algorithm	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**CV Mean**	**Training Time (s)**
Bernoulli NB	0.9030	0.9100	0.8364	0.8716	0.9026	0.0228
KNN (k=9)	0.9073	0.8910	0.8713	0.8810	0.9039	0.0369
SVM (Linear)	**0.9290**	**0.9061**	**0.9125**	**0.9093**	**0.9289**	176.03

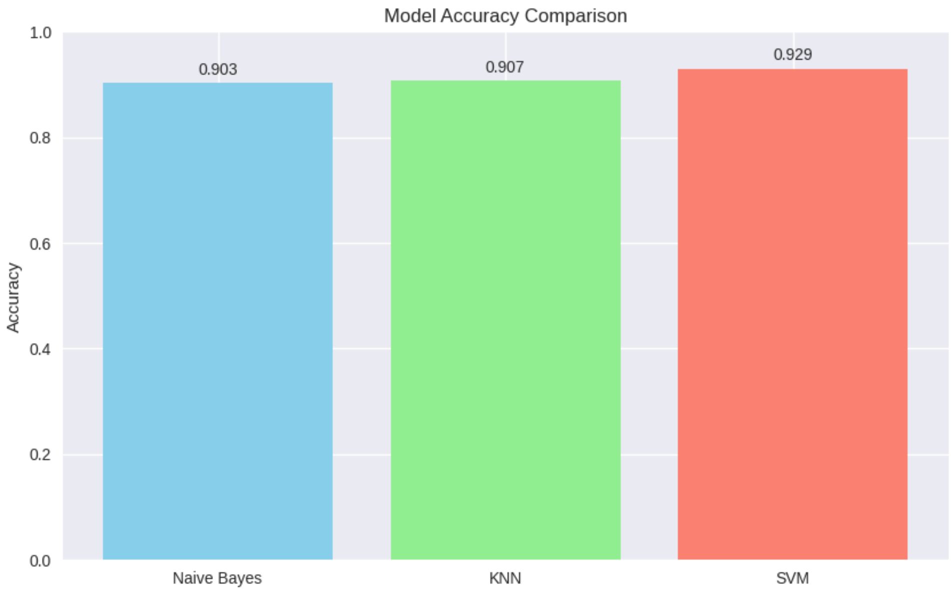

Figure 6: Performance Comparison of All Three Algorithms

9 Observations and Conclusions

9.1 Key Findings

1. **Overall Best Classifier:** SVM with linear kernel achieved the highest accuracy of **92.90%** and F1-score of **90.93%**
2. **Naïve Bayes Analysis:**
   - Bernoulli NB significantly outperformed Gaussian (81.97%) and Multinomial (76.97%) variants
   - Best suited for binary classification problems like spam detection
   - Fastest training time but moderate accuracy
3. **KNN Analysis:**
   - Optimal k=9 provided best balance between bias and variance
   - BallTree algorithm was more efficient than KDTree
   - Good performance but computationally expensive for large datasets
4. **SVM Analysis:**
   - Linear kernel worked best, indicating good linear separability
   - Highest accuracy but longest training time
   - Effective for high-dimensional data like text features

9.2 Performance Trade-offs

Table 8: Algorithm Trade-off Analysis

Algorithm	Accuracy	Speed	Memory	Complexity
Bernoulli NB	90.30%	Very Fast	Low	Low
KNN (k=9)	90.73%	Moderate	Moderate	Low
SVM (Linear)	**92.90%**	Slow	Low	High

9.3 Recommendations

1. **Production Systems:** Use SVM with linear kernel for maximum accuracy
2. **Real-time Applications:** Use Bernoulli Naïve Bayes for speed with acceptable accuracy
3. **Balanced Requirements:** Use KNN with k=9 for good accuracy-speed trade-off
4. **Future Work:** Consider ensemble methods combining all three approaches