CS5691
Pattern Recognition and Machine Learning
Assignment 3

SPAM or HAM

Instructor: Prof. Arun Rajkumar

Student: Janmenjaya Panda

Release Date: 18/04/2024

Submission Deadline: 05/05/2024

Department of Computer Science & Engineering

Indian Institute of Technology Madras

Contents

1 Problem Statement: SPAM or HAM

In this assignment, you will build a spam classifier from scratch. No training data will be provided. You are free to use whatever training data that is publicly available/does not have any copyright restrictions (You can build your own training data as well if you think that is useful). You are free to extract features as you think will be appropriate for this problem. The final code you submit should have a function/procedure which when invoked will be able to automatically read a set a emails from a folder titled test in the current directory. Each file in this folder will be a test email and will be named 'email.txt' ('email1.txt', 'email2.txt', etc). For each of these emails, the classifier should predict +1 (spam) or 0 (non Spam). You are free to use whichever algorithm learnt in the course to build a classifier (or even use more than one). The algorithms (except SVM) need to be coded from scratch. Your report should clearly detail information relating to the data-set chosen, the features extracted and the exact algorithm/procedure used for training including hyperparameter tuning/kernel selection if any. The performance of the algorithm will be based on the accuracy on the test set.

In this assignment we shall build a spam email classifier.

2 Dataset

In this section, we discuss about the dataset used.

2.1 Source

The dataset that is used can be found here. The dataset consists of combined Spam Email CSV of 2007 TREC Public Spam Corpus and Enron-Spam Dataset. This is a csv file containing 83446 records of email which are labelled as either spam or not-spam. The original source is mentioned below.

1. 2007 TREC Public Spam Corpus
   - Original link: here
   - Preprocessed download link: here
2. Enron-Spam Dataset
   - Original link: here

2.2 Description

The description of the dataset goes as follows:

2.3 Columns

1. **label**:

- '1': indicates that the email is classified as spam.
- '0': denotes that the email is legitimate (ham).

2. **text**: This column contains the actual content of the email messages.

2.4 Distribution

The dataset consists of 83448 labelled emails, out of which 39538 are marked 0 (ham) and 43910 are marked 1 (spam). Note that the frequency of each of the mail is 1. The following pie chart depicts the distribution of emails.

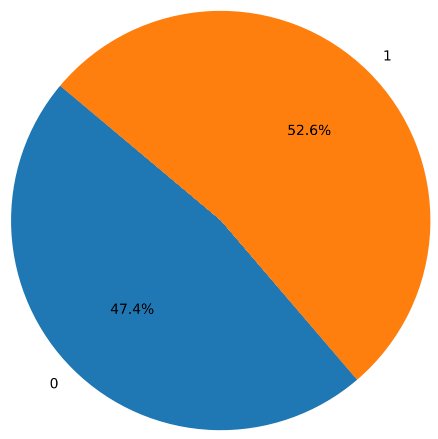

Figure 1: Distribution of emails as per the labels

2.5 Sample mails

Here comes some sample mails from the dataset:

label	**text**
1	ounce feather bowl hummingbird ... divert afterimage
1	wulvob get your medircations online ... visit our website escapenumber
0	computer connection from cnn ... warner company all rights reserved

Table 1: Sample emails and their labels from the dataset

2.6 Train-Test split

We use 80%-20% train-test split method. Thus, we use a set of 66758 of the emails as the training dataset and a set of (distinct) 16690 emails as the test dataset.

3 Feature Extraction

In this section, we discuss about the method followed for the feature extraction, that is — TF-IDF-vectorization.

3.1 Theory

Let's discuss of the feature extraction using TF-IDF vectorization. TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic used in information retrieval and text mining to evaluate the importance of a term in a document relative to a collection of documents. It aims to quantify how relevant a term is to a document within a corpus.

3.1.1 TF (Term Frequency)

TF measures the frequency of a term within a document. It is calculated as the ratio of the number of times a term t appears in a document d to the total number of terms in the document. TF gives a sense of how often a term occurs within a specific document, emphasizing terms that appear frequently within a document.

The formula for TF is:

$$\text{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

3.1.2 IDF (Inverse Document Frequency)

IDF measures the rarity of a term across a collection of documents. It is calculated as the logarithm of the ratio of the total number of documents N to the number of documents containing the term t. IDF helps in identifying terms that are distinctive and less common across the entire corpus, thereby emphasizing their importance.

The formula for IDF is:

$$\text{IDF}(t) = \log\left(\frac{N}{\text{Number of documents containing term } t}\right)$$

3.1.3 TF-IDF Calculation

TF-IDF is the product of TF and IDF. It combines local and global term importance into a single metric. TF-IDF assigns higher weights to terms that are frequent within a document but rare across the entire corpus. Thus, it helps in identifying terms that are both significant within a document and distinctive across the corpus.

The formula for calculating TF-IDF is:

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

3.1.4 Feature Extraction using TF-IDF

The process of feature extraction using TF-IDF involves the following steps:

1. **Tokenization**: The text data is tokenized into individual terms (words or n-grams), discarding punctuation and stop words.

2. **TF Calculation**: For each document, calculate the TF for each term present in that document using the formula mentioned earlier. This step identifies the frequency of each term within its respective document.
3. **IDF Calculation**: Calculate IDF for each term across the entire corpus using the formula mentioned earlier. This step identifies the rarity of each term across the entire corpus.
4. **TF-IDF Calculation**: Multiply TF and IDF for each term in each document to obtain the TF-IDF weight for that term. This step assigns a weight to each term that reflects both its local importance within the document and its global importance across the corpus.
5. **Vectorization**: Represent each document as a vector where each dimension corresponds to a unique term in the entire corpus, and the value of each dimension is the TF-IDF weight of the corresponding term in the document. This results in a high-dimensional sparse matrix representation of the entire document corpus.

TF-IDF is a powerful technique for feature extraction in text data. By capturing both local and global term importance, it helps in identifying the most relevant terms in a document corpus, making it valuable for various natural language processing tasks such as text classification, clustering, and information retrieval.

3.2 Implementation

We implementated using the inbuilt function:

```
feature_extraction = TfidfVectorizer(min_df=1, stop_words='english', lowercase=True)

X_train = feature_extraction.fit_transform(X_train_raw)
X_test = feature_extraction.transform(X_test_raw)

y_train = y_train_raw.astype('int')
y_test = y_test_raw.astype('int')

# Display the shape of X and y
print("Shape of X_train (Vectorized):", X_train.shape)
print("Shape of y_train:", y_train_raw.shape)

print("\nShape of X_test (Vectorized):", X_test.shape)
print("Shape of y_test:", y_test_raw.shape)

with open('models/feature_extraction.pkl', 'wb') as file:
    pickle.dump(feature_extraction, file)
```

The output goes as follows:

```
Shape of X_train (Vectorized): (66758, 261147)
Shape of y_train: (66758,)

Shape of X_test (Vectorized): (16690, 261147)
Shape of y_test: (16690,)
```

Thus, the model is consist of 261147 features.

4 Result Metrics

Different metrics are used to evaluate the performace of the model on the binary classification task. In this section, we discuss about these evalution metrics.

4.1 Precision

Precision measures the proportion of true positive predictions among all positive predictions made by a classifier. It indicates the accuracy of positive predictions. Precision is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Where:

- True Positives (TP): The number of correctly predicted positive instances.
- False Positives (FP): The number of incorrectly predicted positive instances.

4.2 Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of true positive predictions among all actual positive instances in the dataset. It indicates the ability of the classifier to correctly identify positive instances. Recall is calculated as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Where:

- False Negatives (FN): The number of positive instances incorrectly classified as negative.

4.3 F1 Score

F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall. F1 score is calculated as:

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

4.4 Accuracy

Accuracy measures the proportion of correct predictions (both true positives and true negatives) made by the classifier among all predictions. It is the most intuitive evaluation metric but may not be suitable for imbalanced datasets. Accuracy is calculated as:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{False Positives} + \text{True Negatives} + \text{False Negatives}}$$

Where:

- True Negatives (TN): The number of correctly predicted negative instances.

4.5 Evaluation Terminology

- True Positives (TP): Instances that are correctly identified as positive.
- True Negatives (TN): Instances that are correctly identified as negative.
- False Positives (FP): Instances that are incorrectly identified as positive (Type I error).
- False Negatives (FN): Instances that are incorrectly identified as negative (Type II error).

4.6 Interpretation

- Precision: A high precision indicates that the classifier has a low false positive rate, making it reliable when it predicts positive instances.
- Recall: A high recall indicates that the classifier has a low false negative rate, making it sensitive to identifying positive instances.
- F1 Score: A high F1 score indicates that the classifier has both high precision and high recall, striking a balance between the two metrics.
- Accuracy: Accuracy provides an overall measure of how well the classifier performs across all classes, but it may not be suitable for imbalanced datasets.

Different evaluation metrics provide insights into different aspects of classifier performance. It's essential to consider the specific characteristics of the dataset and the problem at hand when choosing the appropriate evaluation metric. Additionally, a combination of multiple metrics can provide a more comprehensive understanding of classifier performance.

5 Naive-Bayes

In this section, we discuss the Naive Baysian method of classification, more specifically, the multinomial Naive-Bayes.

5.1 Theory

Multinomial Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem with the assumption of independence between features. It is commonly used for text classification tasks where features represent the frequencies of words or other tokens in the documents.

5.2 Bayes' Theorem

Bayes' theorem is the foundation of Naive Bayes classifiers. It relates the conditional probability of a class given the features to the conditional probability of the features given the class. Mathematically, Bayes' theorem is expressed as:

$$P(y|x_1, x_2, \dots, x_n) = \frac{P(y) \cdot P(x_1, x_2, \dots, x_n|y)}{P(x_1, x_2, \dots, x_n)}$$

Where:

- $P(y|x_1, x_2, \dots, x_n)$ is the posterior probability of class y given the features $x_1, x_2, \dots, x_n$.
- $P(y)$ is the prior probability of class y.
- $P(x_1, x_2, \dots, x_n|y)$ is the likelihood of the features $x_1, x_2, \dots, x_n$ given class y.
- $P(x_1, x_2, \dots, x_n)$ is the probability of the features $x_1, x_2, \dots, x_n$.

5.3 Multinomial Naive Bayes

In Multinomial Naive Bayes, we model the likelihood of the features given the class as a multinomial distribution. It assumes that each feature represents the frequency count of a term (word or token) in the document. The probability mass function (PMF) of the multinomial distribution is given by:

$$P(x_1, x_2, \dots, x_n|y) = \frac{n!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \cdot \prod_{i=1}^{n} P(w_i|y)^{x_i}$$

Where:

- $x_1, x_2, \dots, x_n$ are the frequencies of terms $w_1, w_2, \dots, w_n$ in the document.
- $P(w_i|y)$ is the probability of term w_i occurring in documents of class y.
- n is the total number of terms in the document.

5.4 Classification Rule

To classify a new document, Multinomial Naive Bayes applies the maximum a posteriori (MAP) decision rule. It assigns the class label that maximizes the posterior probability $P(y|x_1, x_2, \dots, x_n)$:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} P(y) \cdot P(x_1, x_2, \dots, x_n|y)$$

5.5 Smoothing

To handle unseen terms in the test set, Laplace smoothing or other smoothing techniques are often applied to the probability estimates. Laplace smoothing adds a small constant α to each count to avoid zero probabilities:

$$P(w_i|y) = \frac{N_{w_i,y} + \alpha}{N_y + \alpha \cdot V}$$

Where:

- $N_{w_i,y}$ is the count of term w_i in documents of class y.
- N_y is the total count of terms in documents of class y.
- V is the size of the vocabulary (total number of distinct terms).

5.6 Estimation

Let's denote:

- $N_{w_i,y}$ as the count of term w_i in documents of class y.
- N_y as the total count of terms in documents of class y.
- V as the size of the vocabulary (total number of distinct terms).
- α as the Laplace smoothing parameter.

The probability of term w_i occurring in documents of class y, denoted as $P(w_i|y)$, is estimated using Laplace smoothing:

$$P(w_i|y) = \frac{N_{w_i,y} + \alpha}{N_y + \alpha \cdot V}$$

The prior probability of class y, denoted as $P(y)$, is estimated as the proportion of documents in the training set belonging to class y:

$$P(y) = \frac{N_y}{N_{\text{total}}}$$

Where N_{total} is the total number of documents in the training set.

The posterior probability $P(y|x_1, x_2, \dots, x_n)$ for each class y, where $x_1, x_2, \dots, x_n$ are the features (term frequencies) of the document, is computed as:

$$P(y|x_1, x_2, \dots, x_n) = P(y) \cdot \prod_{i=1}^{n} P(w_i|y)^{x_i}$$

The class label for the document is then assigned based on the maximum a posteriori (MAP) decision rule:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} P(y|x_1, x_2, \dots, x_n)$$

Where $\hat{y}$ is the predicted class label.

5.7 Algorithm

Here is the algorithm to implement Multinomial Naive Bayes.

Algorithm 1 : Multinomial Naive Bayes

```
1: Input: Training data $(X, y)$, Laplace smoothing parameter $\alpha$
2: Output: Fitted model with class log priors and feature log probabilities
3: Initialize class_log_prior_ and feature_log_prob_ arrays
4: for each class $c$ in the set of unique classes do
5:     Select samples belonging to class $c$
6:     Compute class log prior:
       $$\text{class\_log\_prior}[c] \leftarrow \log\left(\frac{\text{number of samples in class } c + \alpha}{\text{total number of samples} + \alpha \times \text{number of classes}}\right)$$
7:     Compute feature log probabilities:
       $$\text{feature\_log\_prob}[c] \leftarrow \log\left(\frac{\text{sum of feature occurrences in class } c + \alpha}{\text{sum of feature occurrences in class } c + \alpha \times \text{number of features}}\right)$$
8: end for
9: Function PREDICT_CLASS_PROBABILITIES($X$)
10:     Compute class probabilities for each sample in $X$:
11:     class_probabilities $\leftarrow X \times \text{feature\_log\_prob}^T + \text{class\_log\_prior}$
12:     return class_probabilities
13: Function PREDICT($X$)
14:     Compute predicted class labels for each sample in $X$:
15:     predicted_classes $\leftarrow$ argmax(PREDICT_CLASS_PROBABILITIES($X$))
16:     return predicted_classes
```

5.8 Implementation

Following we mention the implementaiton of Multinomial Naive Bayes:

```python
class MultinomialNaiveBayes:
    def __init__(self, alpha=1.0):
        # Initialize Multinomial Naive Bayes with alpha Laplace smoothing parameter
        self.alpha = alpha
        self.class_log_prior_ = None  # Placeholder for class log priors
        self.feature_log_prob_ = None  # Placeholder for feature log probabilities
        self.classes_ = None  # Placeholder for unique classes

    def fit(self, X, y):
        # Fit the Multinomial Naive Bayes model to the training data X and labels y
        n_samples, n_features = X.shape  # Get the number of samples and features
        self.classes_ = np.unique(y)  # Get the unique classes from the labels

        n_classes = self.classes_.shape[0]  # Number of unique classes
        self.class_log_prior_ = np.zeros(n_classes)  # Initialize class log priors array
        self.feature_log_prob_ = np.zeros((n_classes, n_features))  # Initialize feature
         ↪ log probabilities array

        for i, c in enumerate(self.classes_):
            # Iterate over each class
            X_c = X[y == c]  # Select samples belonging to class c
            self.class_log_prior_[i] = np.log((X_c.shape[0] + self.alpha) / (n_samples +
             ↪  self.alpha * n_classes))
            # Compute class log prior
            self.feature_log_prob_[i] = np.log((np.sum(X_c, axis=0) + self.alpha) /
             ↪  (np.sum(X_c) + self.alpha * n_features))
            # Compute feature log probabilities

    def predict_class_probabilities(self, X):
        # Predict class probabilities for input data X
        return safe_sparse_dot(X, self.feature_log_prob_.T) + self.class_log_prior_

    def predict(self, X):
        # Predict class labels for input data X
        return self.classes_[np.argmax(self.predict_class_probabilities(X), axis=1)]
```

5.9 Result

We obtain the following result on the train dataset:

```
-------------------------------------------------------
Classification_Report (Training):
               precision    recall  f1-score   support

           0      0.9745    0.9890    0.9817     31702
           1      0.9899    0.9766    0.9832     35056

    accuracy                          0.9825     66758
   macro avg      0.9822    0.9828    0.9824     66758
weighted avg      0.9826    0.9825    0.9825     66758
```

The confusion matrix of the train dataset goes as follows:

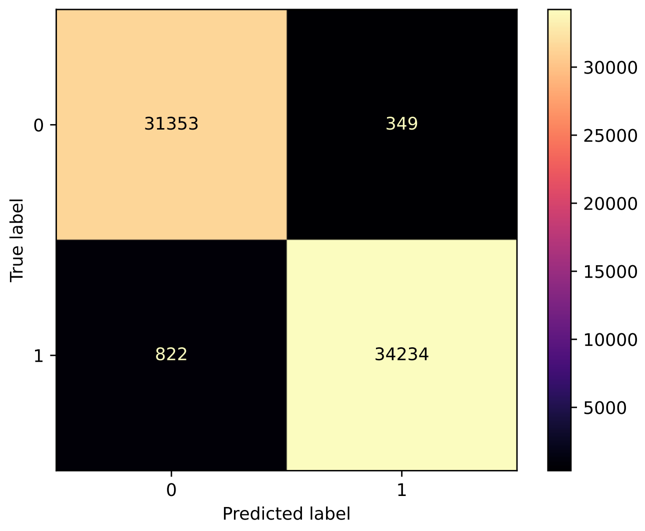

Figure 2: Confusion Matrix (Train)

We obtain the following result on the test dataset:

```
--------------------------------------------------------
Classification_Report (Test):
               precision    recall  f1-score   support

           0      0.9598    0.9890    0.9742      7836
           1      0.9900    0.9633    0.9765      8854

    accuracy                          0.9754     16690
   macro avg      0.9749    0.9762    0.9753     16690
weighted avg      0.9758    0.9754    0.9754     16690
```

The confusion matrix of the test dataset goes as follows:

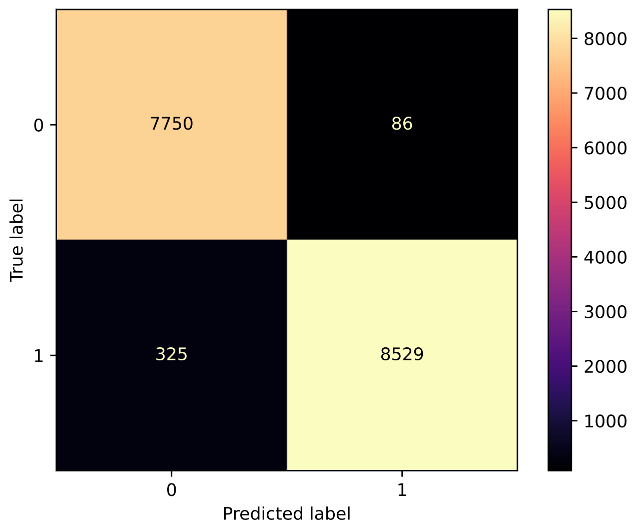

Figure 3: Confusion Matrix (Test)

6 Support Vector Machine

In this section, we shall discuss about support vector machine.

6.1 Theory

Support Vector Machine (SVM) is a powerful supervised machine learning algorithm used for classification and regression tasks. It works by finding the hyperplane that best separates the data points into different classes, maximizing the margin between the classes.

6.1.1 Linear SVM

In its simplest form, SVM constructs a linear decision boundary to separate two classes. Given a training dataset consisting of input-output pairs (x_i, y_i), where x_i represents the input features and y_i represents the class label ($y_i \in \{-1, +1\}$), SVM aims to find the optimal hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ that maximizes the margin between the two classes.

6.1.2 Margin

The margin is defined as the distance between the hyperplane and the nearest data point from either class. The goal of SVM is to maximize this margin.

6.1.3 Optimization Objective

SVM aims to solve the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2} \|\mathbf{w}\|^2$$

subject to the constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geqslant 1, \text{ for all } i$$

Here, $\mathbf{w}$ is the weight vector perpendicular to the hyperplane, and b is the bias term.

6.1.4 Support Vectors

Support vectors are the data points that lie closest to the decision boundary and determine its position. These are the points that have a non-zero slack variable ($\xi_i > 0$).

6.1.5 Dual Optimization:

Given the primal optimization problem for linear SVM:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2$$

subject to the constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geqslant 1, \text{ for all } i$$

We can formulate the Lagrangian $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$ by introducing Lagrange multipliers $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1\right)$$

To find the dual optimization problem, we minimize the Lagrangian with respect to $\mathbf{w}$ and b, and maximize with respect to $\boldsymbol{\alpha}$:

$$\min_{\mathbf{w},b} \max_{\boldsymbol{\alpha}} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$$

Taking derivatives with respect to $\mathbf{w}$ and b and setting them to zero gives us:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \implies \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \implies \sum_{i=1}^{n} \alpha_i y_i = 0$$

Substituting these back into the Lagrangian, we get the dual objective function:

$$\max_{\boldsymbol{\alpha}} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \right)$$

subject to the constraints:

$$0 \leqslant \alpha_i \leqslant C, \text{ for all } i$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

This is the dual formulation of the linear SVM optimization problem, where α_i are the Lagrange multipliers, C is the regularization parameter, and $(\mathbf{x}_i \cdot \mathbf{x}_j)$ denotes the dot product of feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. The solution to this quadratic programming problem yields the optimal values of $\mathbf{w}$ and b, and the decision boundary is defined by the support vectors.

6.1.6 Soft Margin SVM

In real-world scenarios, the data may not be perfectly separable by a linear hyperplane. To handle this, SVM introduces a slack variable ξ_i for each data point, allowing some points to be on the wrong side of the margin or even on the wrong side of the decision boundary. This leads to the formulation of a soft-margin SVM:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \xi_i$$

subject to the constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geqslant 1 - \xi_i, \text{ and } \xi_i \geqslant 0, \text{ for all } i$$

6.1.7 Kernel Trick

In cases where the data is not linearly separable in the original feature space, SVM can be extended to use a kernel function to map the input features into a higher-dimensional space where the classes become separable. Common kernel functions include:

1. Linear Kernel:
$$K(x_i, x_j) = x_i \cdot x_j$$
2. Polynomial Kernel:
$$K(x_i, x_j) = (\gamma x_i \cdot x_j + r)^d$$
3. Radial Basis Function (RBF) Kernel, aka Gaussian Kernel:
$$K(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$$

Here, γ and r are kernel parameters that control the influence of the higher-dimensional space.

6.1.8 Regularization Parameter C

The regularization parameter C in SVM controls the trade-off between maximizing the margin and minimizing the classification error.

6.1.9 Loss Function

For linear SVM, the hinge loss function is commonly used to penalize misclassifications.

$$\ell(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

where $f(x) = (w \cdot x + b)$ is the decision function, and y is the true class label.

Support Vector Machines are versatile and powerful algorithms used for classification and regression tasks. By finding the optimal hyperplane that maximizes the margin between classes, SVM achieves robust and accurate classification, especially in high-dimensional spaces and cases where the classes are not linearly separable. With the flexibility of kernel functions, SVM can handle nonlinear decision boundaries, making it suitable for a wide range of applications in machine learning and pattern recognition.

6.2 Implementation

The model was implemented using sklearn inbuilt function.

6.2.1 Linear Kernel

```
from sklearn.svm import SVC

model_svm_linear = SVC(kernel='linear', random_state=seed)
model_svm_linear.fit(X_train, y_train)

test_prediction_svm_linear = model_svm_linear.predict(X_test)
```

6.2.2 RBF Kernel

```
from sklearn.svm import SVC

model_svm_linear = SVC(kernel='rbf', random_state=seed)
model_svm_linear.fit(X_train, y_train)

test_prediction_svm_linear = model_svm_linear.predict(X_test)
```

6.3 Result

The following results are obtained.

6.3.1 Linear Kernel

We obtain the following result on the train dataset:

```
-------------------------------------------------------
Classification_Report (Training):
              precision    recall  f1-score   support

           0     0.9988    0.9951    0.9970     31702
           1     0.9956    0.9989    0.9973     35056

    accuracy                         0.9971     66758
   macro avg     0.9972    0.9970    0.9971     66758
weighted avg     0.9971    0.9971    0.9971     66758
```

The confusion matrix of the train dataset goes as follows:

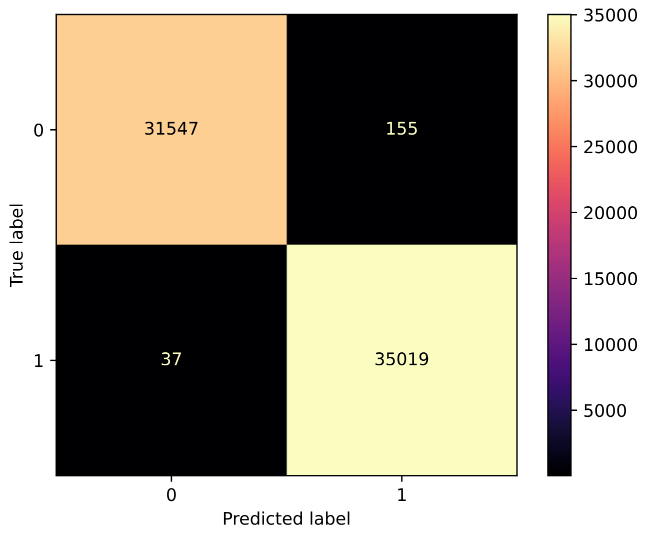

Figure 4: Confusion Matrix (Train)

We obtain the following result on the test dataset:

```
--------------------------------------------------------
Classification_Report (Test):
               precision    recall  f1-score   support

           0     0.9941    0.9877    0.9909      7836
           1     0.9892    0.9948    0.9920      8854

    accuracy                         0.9915     16690
   macro avg     0.9917    0.9913    0.9915     16690
weighted avg     0.9915    0.9915    0.9915     16690
```

The confusion matrix of the test dataset goes as follows:

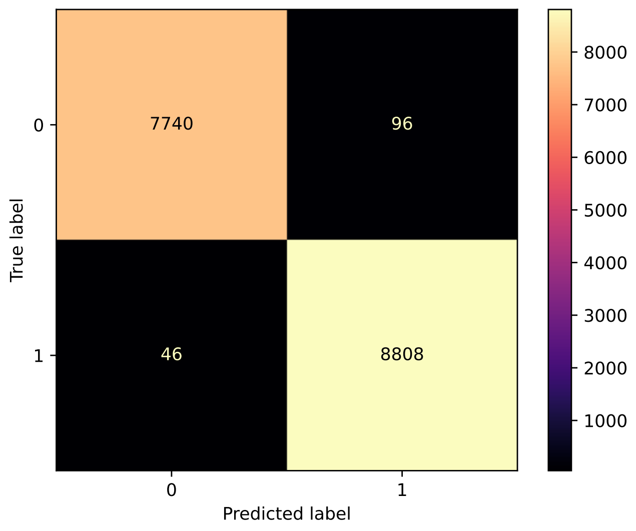

Figure 5: Confusion Matrix (Test)

6.3.2 RBF Kernel

We obtain the following result on the train dataset:

```
-------------------------------------------------------
Classification_Report (Train):
               precision    recall  f1-score   support

           0      0.9995    0.9985    0.9985     31702
           1      0.9978    0.9995    0.9987     35056

    accuracy                          0.9986     66758
   macro avg      0.9986    0.9986    0.9986     66758
weighted avg      0.9986    0.9986    0.9986     66758
```

The confusion matrix of the train dataset goes as follows:

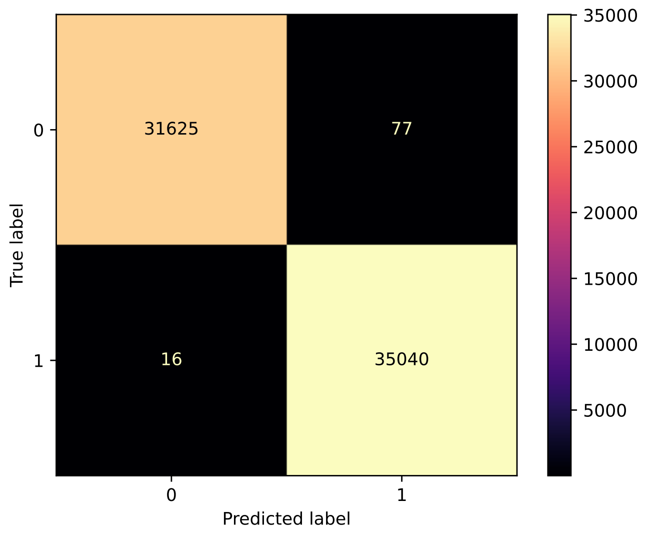

Figure 6: Confusion Matrix (Train)

We obtain the following result on the test dataset:

```
--------------------------------------------------------
Classification_Report (Test):
               precision    recall  f1-score   support

           0      0.9955    0.9857    0.9906      7836
           1      0.9875    0.9960    0.9917      8854

    accuracy                          0.9912     16690
   macro avg      0.9912    0.9912    0.9912     16690
weighted avg      0.9912    0.9912    0.9912     16690
```

The confusion matrix of the test dataset goes as follows:

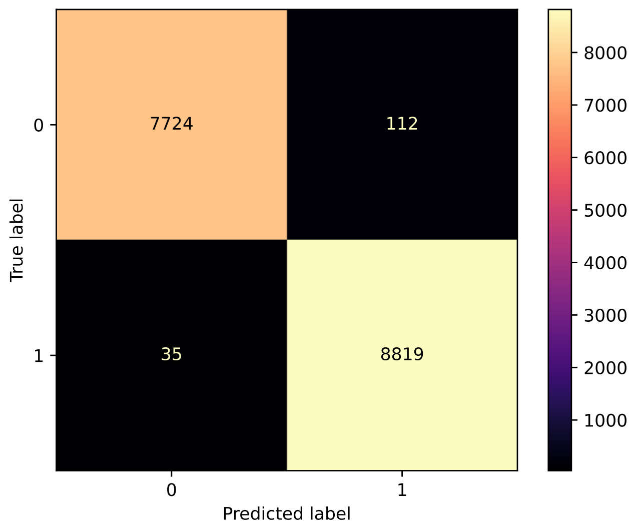

Figure 7: Confusion Matrix (Test)

Clearly, SVM performed much better than the Naive Bayes Method.

7 How to use on test data

Please create a folder named `test` in the current directory and put the test emails (that are: `'email1.txt'`, `'email2.txt'`, etc.) in it. You may run the `.py` file (**main.py**) using the following command (assuming your system uses Python 3):

```
python3 main.py
```

This shall generate a file named `output.txt` which contains the required labeling. A sample content of `output.txt` is shown below:

```
Predicted Classes:
1 - SPAM; 0 - HAM.

email       predicted
file        class
email1.txt: 0
email2.txt: 0
email3.txt: 1
email4.txt: 1
email5.txt: 1
```

We use the simple majority of the result generated by all the models discussed above.