CS5691

Pattern Recognition and Machine Learning

Programming Assignment 3

Rishabh Singh Gaharwar (CS21B067)

Contents

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.

1 Introduction

For this assignment, the training as well as testing data is from **Enron Spam dataset** from HuggingFace. Each e-mail in training as well as testing data has the following attributes

- `message_id` : An identifier for the email
- `text` : The subject and content of the email concatenated
- `label` : The label associated with the email which indicates if it is spam or ham
  - Label 0 corresponds to HAM
  - Label 1 corresponds to SPAM
- `label_text` : The associated label as text (`spam` or `ham`)
- `subject` : The subject that was associated with the email
- `message` : The textual content of the email
- `date` : The date associated with the email

To load the dataset, I used datasets library. The training dataset contains around 31700 emails, out of which around 51% are labelled spam and 49% are labeled ham. Test dataset contains exactly 2000 emails, of which 50.4% are labelled spam and 49.6% are labelled ham.

As per Securelist's "Spam and Phising in 2023" report, around 45.60% of all email sent worldwide are spam emails. Thus this dataset is pretty close to real-world observations as well. The Enron Spam dataset is suitable for training and testing a machine learning model for email classification. With *approximately equal proportions of spam and ham emails in* *both training and test datasets*, the model can be trained effectively to classify emails accurately.

In the following sections, I have explained my code, as well as provided results on the models. Please check confusion matrices of the models and proceeed to Results section to skip code explanation

2 Preprocessing

For feature extraction, I have used the NLTK library, and performed the following preprocessing steps:

1. **Tokenization:** Performed using NLTK's TreebankWordTokenizer, this procedure splits the email-content into individual tokens. This step is required to divide the sentences of the mail into individual words. This is not the same as simply splitting on whitespaces, as tokenization also gets rid of special characters. However, I believe that spam mails are very likely to contain the $ symbol or other currency related symbols, thus I've added a separate word "dollar" to denote $ in the mails.

2. **Stopword Removal**: NLTK has a set of stopwords observed in the English language. Stopwords are words that do not add any information to a body of text. Since these words do not add any information and are not useful in text analysis, we can get rid of these words and reduce the dimensionality of the vectors that we will use to represent these mails.

3. **Lemmatization**: Using NLTK's WordNetLemmatizer, words have been reduced to their root/lemma. This is important to get rid of redundancies among words. To give an example, words like

   - include
   - includes
   - including

   all get reduced to "include" rather than having 3 separate words.

After the email content was processed using the steps mentioned above, a **bag of words** was created. Emails were encoded as a d dimensional vector where d was set to 5000.

For every email, I created two vectors x_F and x_P corresponding to it:

1. `frequency_vector` x_F : $x_{F,i}$ denotes the frequency of the i^{th} word in the email.

2. `presence_vector` x_P : $x_{P,i}$ denotes whether the i^{th} word is present in the email or not.

3 Soft Margin Support Vector Machine

Three SVM classifiers are trained using different input feature representations:

1. SVM with Hinge Loss: Trained using frequency vectors as input features.
2. SVM with Presence Vector: Trained using presence vectors as input features.
3. SVM with Squared Hinge Loss: Trained using frequency vectors as input features.

To do this, I used python's `scikit-learn` library.

- **SVM with Hinge Loss:**
  - Initialize the SVM classifier object using `LinearSVC` from the `svm` module in scikit-learn.
  - Specify parameters such as loss function (`loss='hinge'`), optimization method (`dual='auto'`), and fit intercept (`fit_intercept=True`).
  - Fit the classifier to the training data, where the input features are frequency vectors of processed emails and the labels are the corresponding class labels.
  - The model took **2.5 seconds** to train.
- **SVM with Presence Vector:**
  - Similar to the above steps, initialize another SVM classifier object with the same parameters.
  - Fit this classifier to the training data using presence vectors as input features.
  - The model took **2 seconds** to train.
- **SVM with Squared Hinge Loss:**
  - The motivation behind using this model is that `scikit-learn` by default uses `loss='squared_hinge'` to train SVM models.
  - The classifier is initialized with specific parameters:
  - Loss function: Squared hinge loss (`loss='squared_hinge'`).
  - Optimization method: Auto selection of optimization method (`dual='auto'`).
  - Fit intercept: True, allowing the model to calculate the intercept (`fit_intercept=True`).
  - Maximum number of iterations: 10,000 (`max_iter=10000`). Surprisingly this default parameter had to be increased since the model didn't converge within 1000 iterations
  - The model took **39.1 seconds** to train.

For all three SVM models, evaluation was done using a confusion matrix. The following confusion matrices were obtained

	Actual Spam	Actual Ham
Predicted Spam	988	20
Predicted Ham	20	972

- Accuracy : 98.6 %
- Precision : 98.02 %
- Recall : 98.02 %
- F1-Score : 98.02 %

Table 1: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	16159	37
Predicted Ham	4	15516

- Accuracy : 99.87 %
- Precision : 99.77 %
- Recall : 99.98 %
- F1-Score : 99.87 %

Table 2: Confusion Matrix - **Training** Data

Figure 1: Model - **SVM with Hinge Loss**

	Actual Spam	Actual Ham
Predicted Spam	995	24
Predicted Ham	13	968

- Accuracy : 98.15 %
- Precision : 97.64 %
- Recall : 98.71 %
- F1-Score : 98.17 %

Table 3: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	16159	36
Predicted Ham	4	15517

- Accuracy : 99.87 %
- Precision : 99.78 %
- Recall : 99.98 %
- F1-Score : 99.88 %

Table 4: Confusion Matrix - **Training** Data

Figure 2: Model - **SVM with Presence Vector**

	Actual Spam	Actual Ham
Predicted Spam	990	20
Predicted Ham	18	972

- Accuracy : 98.10 %
- Precision : 98.01 %
- Recall : 98.21 %
- F1-Score : 98.11 %

Table 5: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	16161	24
Predicted Ham	2	15529

- Accuracy : 99.91 %
- Precision : 99.85 %
- Recall : 99.99 %
- F1-Score : 99.92 %

Table 6: Confusion Matrix - **Training** Data

Figure 3: Model - **SVM with Squared Hinge Loss**

Since SVM models reach desirable and sufficiently high accuracy on test set, there was no incentive to kernlize SVM and test.

4 Naive Bayes

To implement the Naive Bayes classifier, I created a class called `NaiveBayes`. I trained this model on presence_vector of training datapoints.

- NaiveBayes class:
  - The `NaiveBayes` class is initialized with processed email data (`processed_mails_training`).
  - It extracts the features (`X`) and labels (`Y`) from the processed emails.
  - It also separates the features and labels of spam and non-spam emails into `ones` and `zeros` arrays, respectively.
- `parameterTuning` method:
  - This method is used to calculate the parameters required for classification.
  - conditional probabilities of each feature are found given that the email is spam (`probability1`) and non-spam (`probability0`). This is equivalent to finding $P(X|Y = +1)$ and $P(X|Y = -1)$ respectively (as done in class)
  - It also calculates the prior probability of spam emails (`p`).
- `predict` method:
  - This method predicts whether an email is spam or not.

  - It calculates the log probabilities of the given email belonging to each class (spam or non-spam) and returns the class with the higher probability. To avoid precision errors, sum of logarithms has been taken instead of multiplying to find the likelihood

- `runNaiveBayes` function:
  - This function initializes a `NaiveBayes` object with processed training emails, tunes its parameters, and then uses it to predict the labels of test emails.
  - It prints the total number of predicted spam emails.
  - It calculates the confusion matrix for the Naive Bayes classifier using the predicted labels and the true labels of test emails.

This model took **8.5 seconds** to train. The following Confusion Matrix was obtained for this model:

	Actual Spam	Actual Ham
Predicted Spam	1000	67
Predicted Ham	8	925

- Accuracy : 96.25 %
- Precision : 93.72 %
- Recall : 99.20 %
- F1-Score : 96.39 %

Table 7: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	16072	821
Predicted Ham	91	14732

- Accuracy : 97.12 %
- Precision : 95.14 %
- Recall : 99.44 %
- F1-Score : 97.24 %

Table 8: Confusion Matrix - **Training** Data

Figure 4: Model - **Naive Bayes**

5 Logistic Regression

To implement the Logistic Regression classifier, I created a class called `LogisticRegression`.

1. **Initialization**:
   - The class `LogisticRegression` is initialized with processed email data.
   - Frequency vectors and labels are extracted from the processed email data.
   - The weight vector **w** is initialized with zeros.
2. **Link Function**:

- The `linkFunction` method calculates the sigmoid function value of the dot product of input features and weights.

3. **Gradient Calculation**:

   - The `gradient` method calculates the gradient of the log-likelihood function.

4. **Gradient Ascent**:

   - The `gradientAscent` method performs gradient ascent to update the weight vector $\mathbf{w}$ over several epochs.
   - The learning rate is inversely proportional to the iteration number ($\frac{\text{learning\_rate}}{i}$), where i is the iteration number.
   - It calculates the difference between consecutive weight vectors to check for convergence.

5. **Prediction**:

   - The `predict` method predicts the label for a given test email based on the learned weights.

This model took **3 minutes, 20 seconds** to train. I would like to mention that I had set number of epochs to be 200 and learning rate to be $\frac{10^{-5}}{i}$ in the i^{th} iteration. Other values of epochs and learning rate might deliver better model faster. Do note that Logistic Regression was trained and tested on **frequency vector** rather than binary vector.

The confusion matrix obtained for this model:

	Actual Spam	Actual Ham
Predicted Spam	970	48
Predicted Ham	38	944

- Accuracy : 95.70 %
- Precision : 95.28 %
- Recall : 96.23 %
- F1-Score : 95.75 %

Table 9: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	15620	794
Predicted Ham	543	14759

- Accuracy : 95.78 %
- Precision : 95.16 %
- Recall : 96.64 %
- F1-Score : 95.89 %

Table 10: Confusion Matrix - **Training** Data

Figure 5: Model - **Logistic Regression**

6 Decision Tree

To implement a decision tree, I first created a `Node` class, to represent nodes of the decision tree.

- `Node` class This class represents each node in the decision tree.
  - `Left`: Reference to the left child node.
  - `Right`: Reference to the right child node.
  - `Value`: Indicates the majority class at the node. If the node is a leaf node, `Value` will be the majority class label in the corresponding subset of data. If the node is not a leaf node, `Value` will be `-1`.
  - `Depth`: Represents the depth of the node in the tree.
  - `Feature`: Represents the feature used for splitting at the node.

Now, using these nodes, I created a `DecisionTree` class

- **Attributes**:
  - `X`: Training features.
  - `Y`: Training labels.
  - `maxDepth`: Maximum depth of the decision tree.
  - `features`: Number of features in the training data.
- **Methods**:
  - `entropy`: Calculates the entropy of a set of labels using the formula:

    $$\text{Entropy} = -p_1 \log_2(p_1) - p_0 \log_2(p_0)$$

    Where p_1 and p_0 are the proportions of positive and negative instances in the dataset.
  - `constructTree`: Recursively constructs the decision tree.
    - Calculates the entropy for each feature and selects the one with the highest information gain (lowest entropy).
    - Constructs child nodes based on the selected feature.
    - Recursively builds the tree until the maximum depth is reached or no more splits can be made.
  - `rootConstructor`: Constructs the root node of the decision tree.

To get predictions from Decision Tree, I created a helper function called `DecisionTreePredict`

- It traverses the decision tree based on the features of the test sample until it reaches a leaf node.

- Returns the majority class at that leaf node as the predicted label.

I trained this model on `presence_vector` of training datapoints. It took **10 minutes** to train a decision tree of height 12. Since construction was computationally expensive for binary datapoints, I did not construct a generalized decision tree and did not cross validate on height of the decision tree.

For decision tree, the following confusion matrix was obtained:

	Actual Spam	Actual Ham
Predicted Spam	859	390
Predicted Ham	149	602

- Accuracy : 73.05 %
- Precision : 68.77 %
- Recall : 85.21 %
- F1-Score : 76.12 %

Table 11: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	13793	6273
Predicted Ham	2370	9280

- Accuracy : 72.72 %
- Precision : 68.73 %
- Recall : 85.33 %
- F1-Score : 76.14 %

Table 12: Confusion Matrix - **Training** Data

Figure 6: Model - **Decision Tree of depth** 12

It is interesting to note that accuracy is higher in case of test datapoints.

7 AdaBoost

To implement boosting, I created a class called `AdaBoost`. I have used decision trees of a single node (i.e. decision stumps) as my weak learners.

- **Initialization**:
  - The constructor `__init__` initializes the AdaBoost classifier with the following parameters:
    - `X_train`: Training features.
    - `Y_train`: Training labels.
    - `gamma` (optional): Learning rate parameter (default value is 0.01).
  - Initializes the weights `D` for each sample and normalizes them.
  - Initializes lists to store weak learners (`learners_`), their weights (`alphas_`), and errors.

  - Calculates the total number of iterations (`T`) based on the training data size and the learning rate. Ideally, if γ is the lowest value such that all weak learners have accuracy $\geq \frac{1}{2} + \gamma$, then $T = \frac{\log 2n}{2\gamma^2}$ where n represents the number of training points.
  - `features` is an array containing indices of features.

- **Weak Learner**:
  - The `weak_learner` method selects a weak learner (a decision stump) based on the weighted error rate.
  - It randomly shuffles the features. This is done to get different weak learners on every invocation.
  - It calculates the error for both possible classifications (0 and 1) and selects the one with the lowest error.
  - Returns the selected weak learner and its error.

- **Boosting**:
  - The `boosting` method trains the AdaBoost model by iteratively selecting weak learners and updating sample weights.
  - It iterates `T` times:
    * Selects a weak learner with the lowest error.
    * Updates the weights of the training samples based on the weak learner's performance.
    * Calculates the learner's weight (`alpha`) based on its error.
    * Stores the weak learner and its weight.

- **Prediction**:
  - The `predict` method predicts the label for a given test sample.
  - It computes the weighted sum of weak learners' predictions using their respective weights (`alphas_`).
  - If sign of the resultant is negative, label 0 is returned, else label 1 is returned.

The model took **33 minutes** to train with $\gamma = 0.01$, which in-turn increased $T = \frac{\log 2n}{2\gamma^2}$
The following confusion matrix was obtained for Adaboost:

	Actual Spam	Actual Ham
Predicted Spam	993	34
Predicted Ham	15	958

- Accuracy : 97.55 %
- Precision : 96.68 %
- Recall : 98.51 %
- F1-Score : 97.59 %

Table 13: Confusion Matrix - **Testing** Data

	Actual Spam	Actual Ham
Predicted Spam	16077	357
Predicted Ham	86	15196

- Accuracy : 98.60 %
- Precision : 97.82 %
- Recall : 99.46 %
- F1-Score : 98.64 %

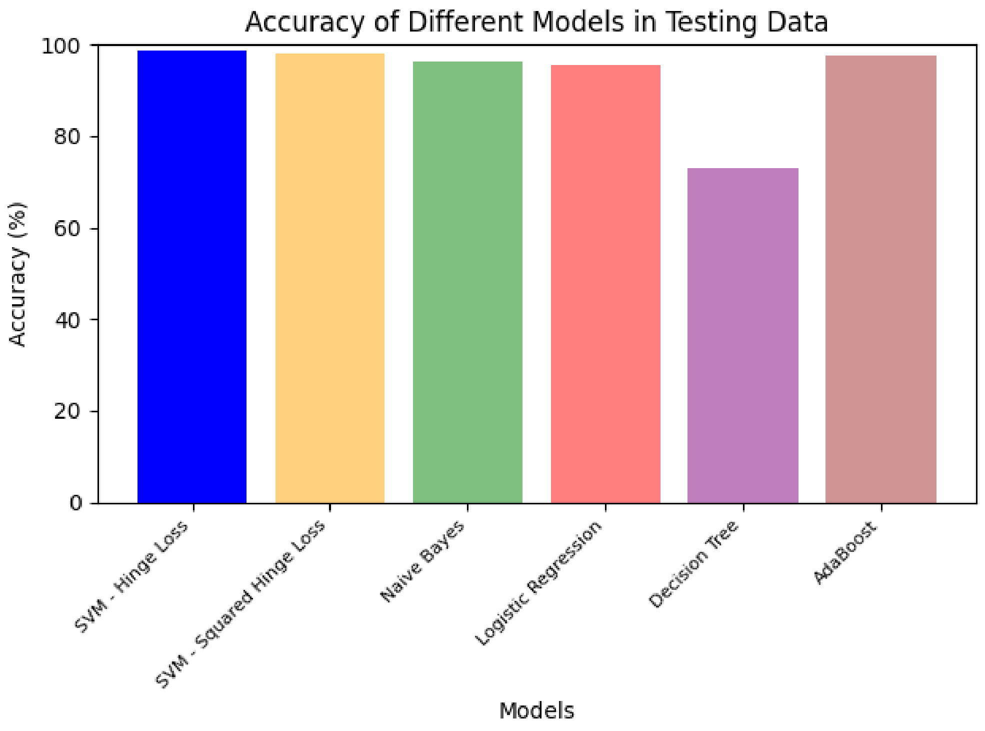
Table 14: Confusion Matrix - **Training** Data

Figure 7: Model - **Adaboost with** 50000 **iterations**

8 Results

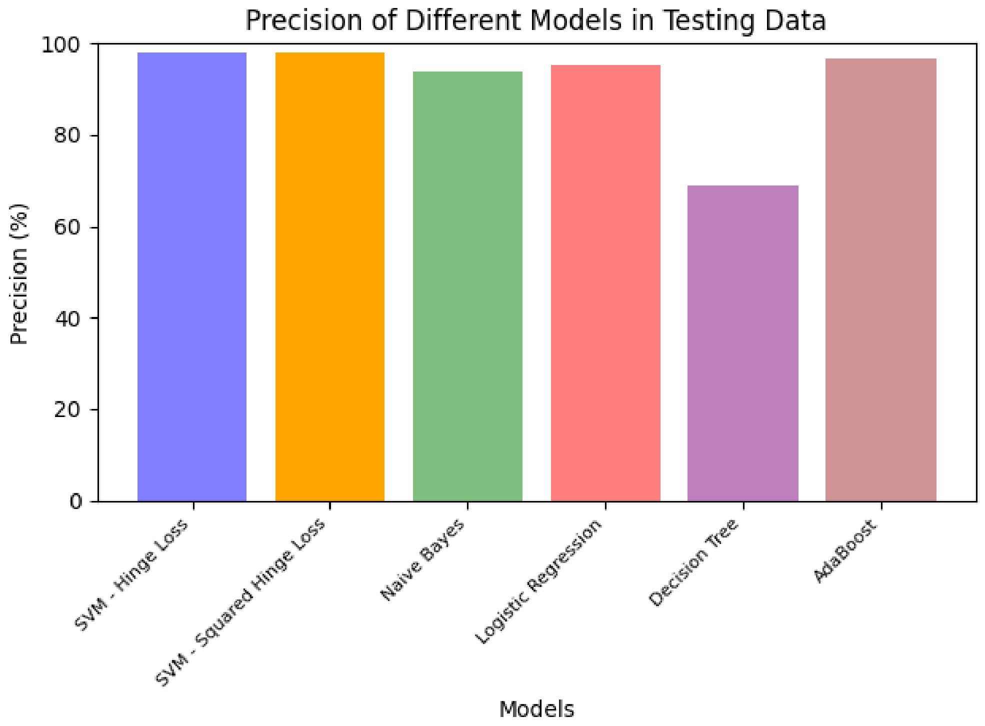
To compare the models, I have plotted the following bar graphs

8.1 Accuracy



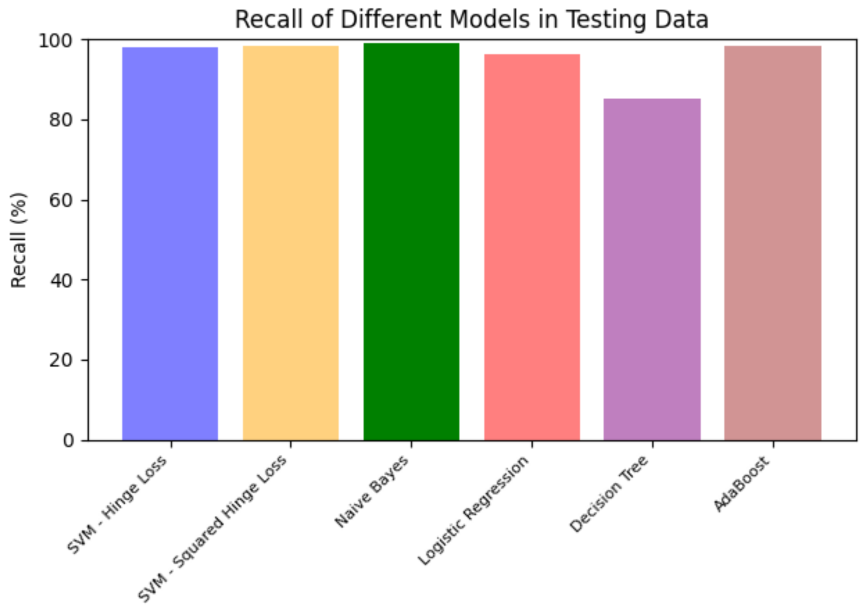
- SVM with Hinge loss has the maximum accuracy
- This is because it is implemented using scikit-learn and Adaboost did not get sufficient number of iterations
- Decision Tree has lowest accuracy
- This is because decision tree has height 12 and is not very general. It is only for binary vectors.

8.2 Precision



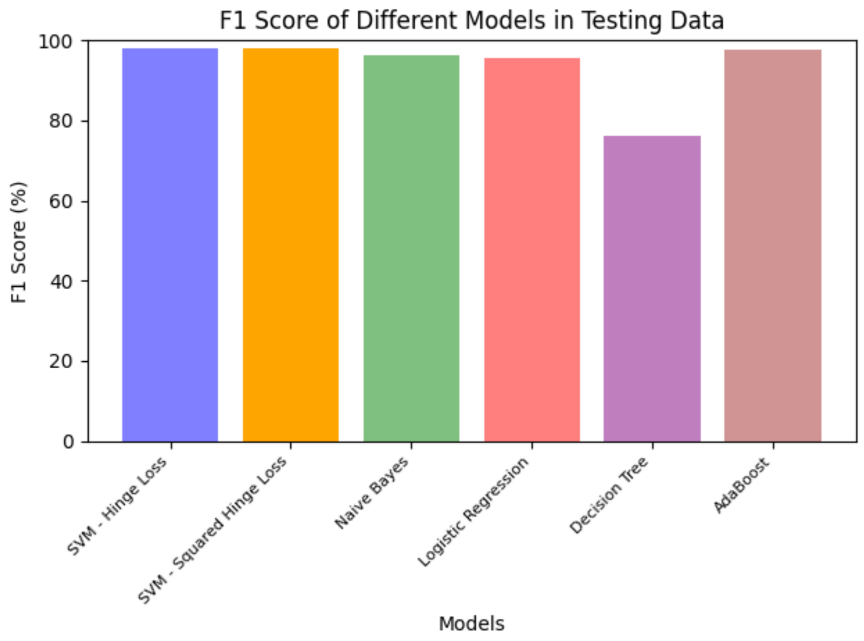
- SVM with Squared Hinge loss has the maximum precision
- Decision Tree has lowest precision
- This is because decision tree has height 12 and is not very general. It is only for binary vectors.

8.3 Recall



- Naive Bayes has the maximum recall
- Decision Tree has lowest recall

8.4 F-1 Score



- SVM with Squared Hinge loss has the highest F-1 score

- Decision Tree has lowest F-1 score

9 Conclusions

- It seems that Decision Tree performs poorly across all metrics.
- This is because its height is only 12
- Also, it is only trained only on presence vectors and not on frequency vectors.
- SVM performs well across all metrics since it is implemented in scikit-learn library
- Adaboost can give strong learner. It appears that Adaboost did not receive enough iterations to return a strong learner.
- There is no overfitting since there isn't a huge difference in values obtained over test and training data for all models.
- Threshold of picking 5000 words with highest cumulative frequency as features is arbitrary.
- Picking words with highest cumulative frequency is a common practice.
- A better way would've been to find the `tf-idf` of all words and choose ones with the highest values.

Please note that the functionality of reading test emails is implemented but not used since I used dataset from HuggingFace.