PRML

SPAM CLASSIFICATION

Assignement - 3 Report

by

E Naveen Koushik Reddy(CS22B006)

Dr Arun Rajkumar

Computer Science and Engineering

Indian Institute of Technology Madras

Jan - May 2024

Contents

0.1 Datasets

The dataset used is downloaded from Kaggle. You can find the website to download the datsets here. The dataset consists of 5728 rows with first column being the email and the second column indicating whether it is spam(1) or not(0).

0.2 Email Preprocessing

```
1  def preprocess_text(emails):
2      for i in range(len(emails)):
3          emails[i] = emails[i].lower()
4          emails[i] = emails[i].replace('subject', '')
5          emails[i] = emails[i].replace('re', '')
6          emails[i] = emails[i].replace('fw', '')
7          emails[i] = re.sub(r'\d+', '␣', emails[i])
8          emails[i] = re.sub(r'\W', '␣', emails[i])
9          emails[i] = re.sub(r'\s+', '␣', emails[i])
```

Listing 1: Processing the text

- The first step is used to make all the alphabets to lower letters as a letter being capital or not does not indicate much whether the email is spam or not.
- The next step from lines 4-6 is used to remove the words that appear in every mail and replace with empty space.
- The RE library sub (re.sub) in python uses regular expressions to check any whether any sub-sequence matches with the given type of input and replaces with the given symbols.
- r' d' checks whether there is any word with digits and replaces with empty string.
- r' W' checks whether there is any non-word characters and replaces with empty space.
- r' s' checks whether there is any excess spaces and replaces with single empty space.

0.3 Feature Extraction

```
#CountVectorizer - Frequency of words
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(emails)
emails_data_count = X.toarray()
features = vectorizer.get_feature_names_out()

#CountVectorizer - Binary (i.e. presence or absence of words)
vectorizer_naivebayes = CountVectorizer(binary=True)
X = vectorizer_naivebayes.fit_transform(emails)
emails_data_count_naivebayes = X.toarray()


#Term Frequency-Inverse Document Frequency
```

```

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(emails)
emails_data_tfidf = X.toarray()
```

Listing 2: Feature extraction

From the library sklearn.feature extraction.text, I used the fuctions CountVectorizer and TfidfVectorizer. Both these functions divides the given dataset of emails to datapoints with each column corresponding to a unique word appearing in the dataset. These words are arranged in ascending lexographic order.

1. In the **CountVectorizor** function, each email is converted to a vector which indicates the count of a particular word in that email. If the binary is set to true in the countvectorizer, it gives 0(absent) or 1(present) instead of the count.

2. In the **TfidfVectorizer** function , TFIDF corresponds to term frequency - Inverse Document frequency (where term refers to the word and a document refers to an email). That is each mail is converted to a vector with values that indicate the (frequency of that word / total number of words in that mail) times (logarithm of (Total number of emails/ number of emails the word appears)) .

Both features are useful for certain tasks.So, using the best out of the possible features in the following algorithms.

0.4 Splitting the data in train and test

```
emails_data_count_train , emails_data_count_test ,
   labels_count_train , labels_count_test = train_test_split(
   emails_data_count , labels , test_size=0.2)
emails_data_count_naivebayes_train ,
   emails_data_count_naivebayes_test ,
   labels_count_naivebayes_train , labels_count_naivebayes_test =
   train_test_split(emails_data_count_naivebayes , labels ,
   test_size=0.2)
emails_data_tfidf_train , emails_data_tfidf_test ,
   labels_tfidf_train , labels_tfidf_test = train_test_split(
   emails_data_tfidf , labels , test_size=0.2)
```

Listing 3: Splitting into test and train

From the python library sklearn.model selection importing test train split which is used to split the train and test data randomly (in this code , dividing the train test in the ratio 4:1)

0.5 Naive Bayes Algorithm

In Naive Bayes, the features that we use are the binary count vectorizer i.e. where 1 indicates the presence of word and 0 indicates the absence of word.
Naive Bayes is a generative model that focuses on the probability of how the data is also generated.

0.5.1 Training

$\hat{p}$ indicates the probability that an email belongs to spam.

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

The probability $\hat{p_j^y}$ indicates the probability of that word appearing in the spam/ non-spam dataset.

$$\hat{p_j^y} = \frac{\sum_{i=1}^{n} 1(f_j^i = 1, y_i = y)}{\sum_{i=1}^{n} 1(y_i = y)}$$

0.5.2 Prediction of test data point

The label of a test data point is obtained from the application of Bayes rule

$$P(y^{test} = 1|x^{test}) \propto \left(\prod_{k=1}^{d} (\hat{p}_k^1)^{f_k}(1-\hat{p}_k^1)^{1-f_k}\right)\cdot \hat{p}$$

$$P(y^{test} = 0|x^{test}) \propto \left(\prod_{k=1}^{d} (\hat{p}_k^0)^{f_k}(1-\hat{p}_k^0)^{1-f_k}\right)\cdot (1-\hat{p})$$

if $P(y^{test} = 1|x^{test}) > P(y^{test} = 0|x^{test})$ predict $y^{test} = 1$ else $y_{test} = 0$

The values of probabilities are very low and the number of features are greater than 33000. Hence the product will be too small and can lead to underflow in decimals, So we can use logarithms of the values to avoid this issue. Hence logarithm is applied to the above inequality and on simplification the following equation is obtained.

$$\sum_{i=1}^{d} f_i \log\left(\frac{\hat{p}i^1(1-\hat{p}_i^0)}{\hat{p}_i^0(1-\hat{p}_i^1)}\right) + \left(\sum_{i=1}^{d}\log\left(\frac{1-\hat{p}_i^1}{\hat{p}_i^0}\right)\right) + \log\left(\frac{\hat{p}}{1-\hat{p}}\right) \geq 0$$

Hence you notice that the final separator of datapoints is a line with slope $\log\left(\frac{\hat{p}i^1(1-\hat{p}_i^0)}{\hat{p}_i^0(1-\hat{p}_i^1)}\right)$ and with constant (bias) $\left(\sum_{i=1}^{d}\log\left(\frac{1-\hat{p}_i^1}{\hat{p}_i^0}\right)\right) + \log\left(\frac{\hat{p}}{1-\hat{p}}\right)$

$$y_{pred} = \begin{cases} 1 & \text{if } w^T \cdot x_{test} + b \geq 0 \\ 0 & \text{if } w^T \cdot x_{test} + b < 0 \end{cases}$$

0.5.3 Inferences

- The original dataset is used to divide it into two datasets with train and test.
- Initially the model is run on train and tested on the test data. This is done just to check the level of accuracy in Naive Bayes.

- Train data Accuracy $\approx 99\%$. Test data Accuracy $\approx 98\%$.
- Finally the line slope and bias is derived from the whole dataset.

0.6 Logistic Regression Algorithm

This is a slight change in the perceptron algorithms which caters to any dataset even if outliers present.

0.6.1 Training

- The loglikelihood function that is to be maximised in logistic regression is given below.

$$\log \mathcal{L}(w; Data) = \sum_{i=1}^{n} \big((1-y_i)(-w^T x_i) - \log\big(1+\exp\big(-w^T x_i\big)\big)\big)$$

- We can't obtain a closed form solution for w from the above equation hence we need to apply gradient ascent to maximize the log-likelihood function.
- The graident is

$$\nabla \log \mathcal{L}(w) = \sum_{i=1}^{n} x_i \left(y_i - \frac{1}{1+\exp(-w^T x_i)}\right)$$

- Gradient Ascent step:

$$w_{t+1} = w_t + \eta \cdot \nabla \log \mathcal{L}(w_t)$$

$$w_{t+1} = w_t + \eta \cdot \sum_{i=1}^{n} x_i \left(y_i - \frac{1}{1+\exp(-w^T x_i)}\right)$$

- $\sigma(w^T \cdot X)$ yields an matrix of size $n \times 1$, where each element represents the corresponding sigmoid function value.
- The Gradient Ascent step can be modified into:

$$w_{t+1} = w_t + \eta(X^T \cdot (Y - \sigma(w^T \cdot X)))$$

0.6.2 Prediction

- The predication for a test data point is given by the sign of $w^T \cdot x_i$

$$y_{pred} = \begin{cases} 1 & \text{if } w^T \cdot x_{test} \geq 0 \\ 0 & \text{if } w^T \cdot x_{test} < 0 \end{cases}$$

0.6.3 Hyperparameter - Step Size

One thing to notes is that, we are not using count data because the count data can have arbitrary values and there are more chances of overflow occuring due to exponential values and also the time required for such large calculations is exponentially more.
Another thing is that using the step size as some constant divided by the ith iteration is taking a lot more time as compared to just the constant. Hence, we are just using some constant as the step size.
Fixing the number of iterations to 100 so that the execution time is less.

Using the test data to get the step size for which the final accuracy on test data is less.
Below are the graphs on errors on test dataset with respect to each iteration in the gradient descent.
From the figures and the final error from above, it is clear that using a step size of 0.1 gives the best result with minimal execution time.

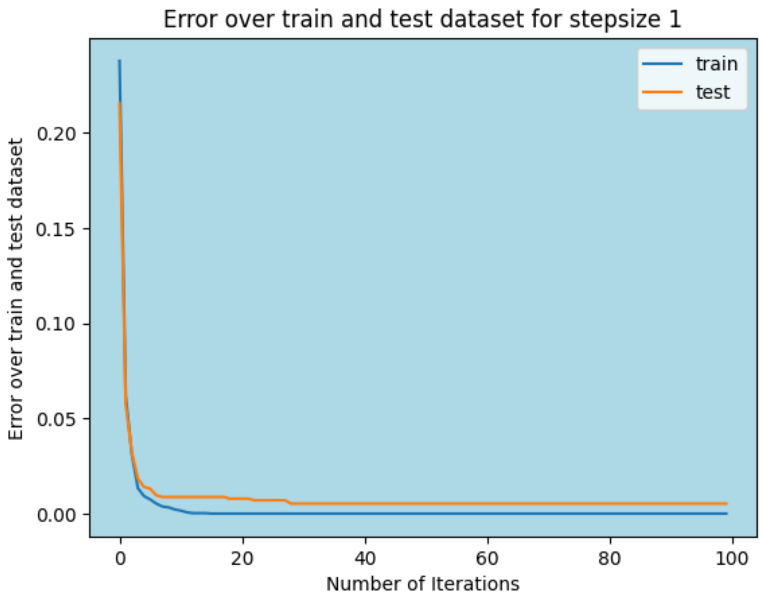

Figure 1: The resulting test error = 0.0052

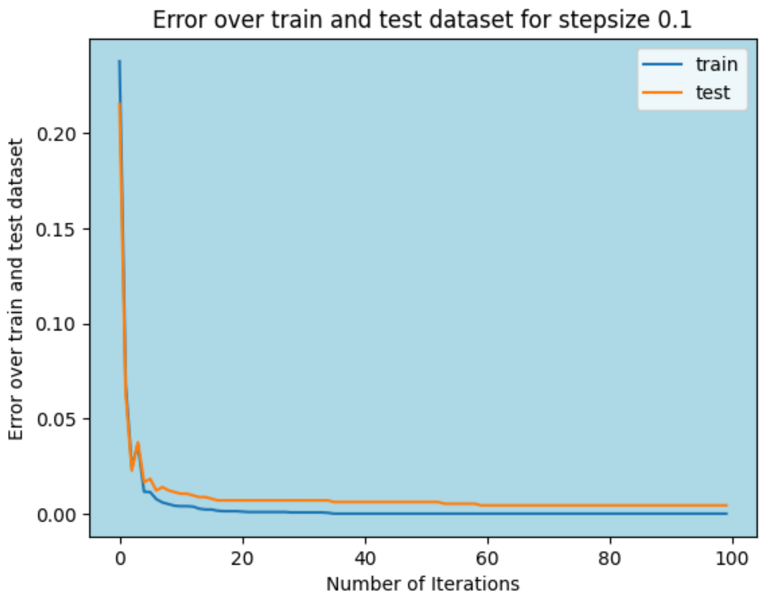

Figure 2: The resulting test error = 0.0043

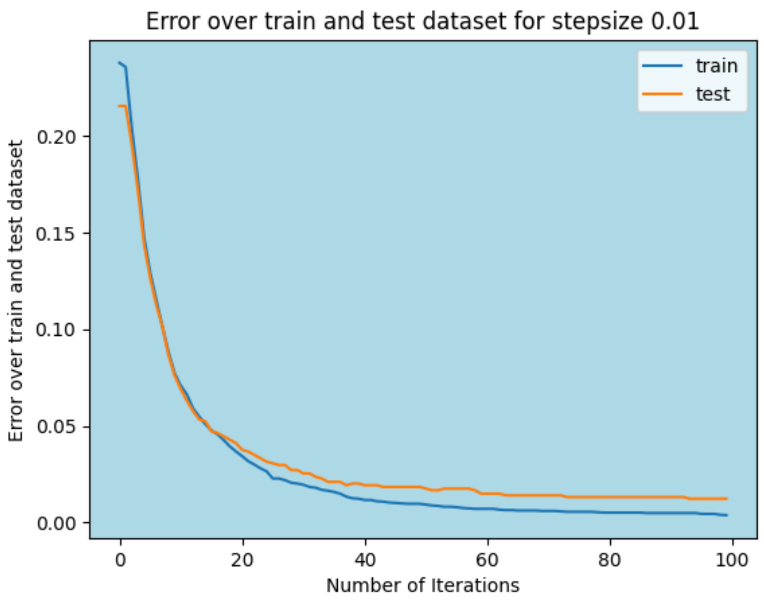

Figure 3: The resulting test error = 0.0122

0.7 SVM - Support Vector Machine Algorithm

0.7.1 Training and Testing

```
    SVClassifier = LinearSVC(dual=True, C = C , max_iter=10000)
    SVClassifier.fit( emails_data_count_train, labels_count_train
       )
    print("Test␣Error␣for␣C␣=",C,SVClassifier.score(
       emails_data_count_test, labels_count_test))
```

Listing 4: SVM

For the SVM Algorithm, I am directly using the library function available in the sklearn module.
The library function fit is used to train the classifier over the train data.
The library function score is used to measure the accuracy of the classifier over the test data.

0.7.2 Hyperparameter - The constant C with Count Features

Test Accuracy for C = 0.001 0.9834205933682374 Train Error for C = 0.001 0.994325621999127
Test Accuracy for C = 0.01 0.9886561954624782 Train Error for C = 0.01 0.9997817546922741
Test Accuracy for C = 0.1 0.9869109947643979 Train Error for C = 0.1 1.0
Test Accuracy for C = 1 0.9860383944153578 Train Accuracy for C = 1 1.0
Test Accuracy for C = 10 0.9851657940663177 Train Accuracy for C = 10 1.0
Test Accuracy for C = 100 0.9851657940663177 Train Accuracy for C = 100 1.0
Test Accuracy for C = 500 0.9851657940663177 Train Accuracy for C = 500 1.0
Test Accuracy for C = 1000 0.9851657940663177 Train Accuracy for C = 1000 1.0
From the values we can see that, the best C would be 1 as for the value 1, we are getting a Train accuracy of 1. For all the values which have train accuracy 1, the best test accuracy is for C = 1.

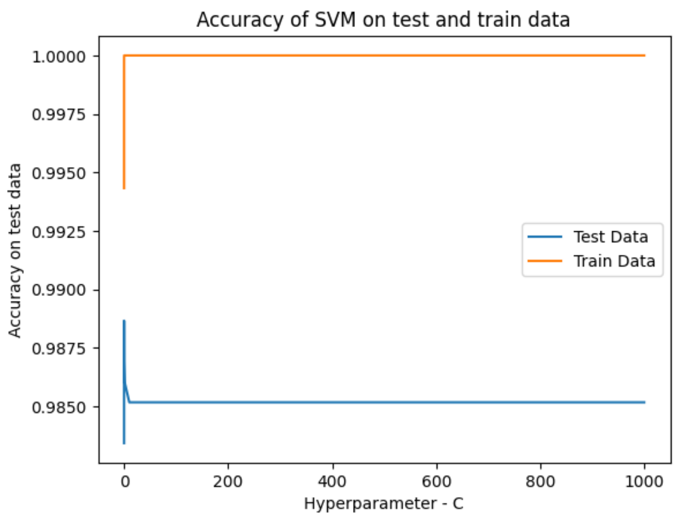

Figure 4: Caption

0.7.3 Hyperparameter - The constant C with TFIDF Features

Test Accuracy for C = 0.001 0.7687 Train Error for C = 0.001 0.784810
Test Accuracy for C = 0.01 0.8315 Train Accuracy for C = 0.01 0.90899
Test Accuracy for C = 0.1 0.988656 Train Accuracy for C = 0.1 0.96377
Test Accuracy for C = 1 0.996509 Train Accuracy for C = 1 0.983195
Test Accuracy for C = 10 0.996509 Train Accuracy for C = 10 0.98646
Test Accuracy for C = 100 0.996509 Train Accuracy for C = 100 0.98646
Test Accuracy for C = 500 0.996509 Train Accuracy for C = 500 0.98646
Test Accuracy for C = 1000 0.996509 Train Accuracy for C = 1000 0.98646

For these values there is no C such that the train accuracy is 1 and the test accuracies are also greater than the count version. Hence, I am using the count version SVM as the final Classifier.

0.8 Final Classifier

The Final Classifier that I am considering is the max of the decision given by all the three above algorithms - Naive Bayes, Logistic Regression , SVM .
That is if any two or more of the classifiers predict that it is spam then it is considered spam or else not.
If we want to be more conservative, we can take the decision that all the three classifiers should predict spam.

0.9 Output

Running the code produces a text file with named test predictions with each classifier predicting the test point and another file named Final Predictions with 1 and 0 where 1 indicates spam and 0 indicates not spam .