Pattern Recognition and Machine Learning

Assignment Report

submitted by

Karthikeya P (CS22B026)

Under the supervision
of

Dr.Arun Rajkumar
Assistant Professor

Computer Science and Engineering

Indian Institute of Technology Madras

Jan - May 2024

Contents

List of Figures

1 Data Preprocessing

1.1 Removing Numbers and Special Characters

We don't to want to avoid numbers and special characters to affect our data. For this we use the re library to remove them from our data. The below code snippet shows the preprocessing implementation.

```
def preprocess_text(text):
    text = text.replace("Subject", "")
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    text = text.lower()

    return text
```

The above code replaces the word 'Subject' with an empty string(") and only takes only English alphabet into consideration neglecting other characters and numbers if present.

1.2 Vectorisation

To train the data we need to convert the emails into vectors. The following piece of code is used to do that.

```
from sklearn.feature_extraction.text import TfidfVectorizer
# let train_data" is the list containing all the emails.
vectoriser = TfidfVectorizer()
vectorised_train_data = vectoriser.fit_transform(train_data)
# here vectorised_train_data is a sparse matrix.
```

1.3 Internal working of TF - IDF

- **TF(Term Frequency):** This is related to words in a particular document. It is the measure of what fraction of the words is the given word in that particular document. It is calculated as the number of times the word appeared in the document divided by the total number of words in the document.

  For a given word "A" and document "B".

  $$TF_{A,B} = \frac{\text{\# of A in B}}{\text{\# of words in B}}$$

- **IDF(Inverse Document Frequency):** This is related to a word in the dataset. It is measure of importance of a word in the entire dataset. This is calculated as the logarithm of the total number of documents divided by the number of documents that contain this word. This helps in decreasing the importance of the terms that frequently appear across different documents of the same set. For example words like "is", "was", "the".. these won't help us in classification so we want to give them least importance.

  For a given word "A" and dataset "D".

  $$IDF_{A,D} = \log\left(\frac{\text{\# documents in dataset}}{\text{\# documents containing A}}\right)$$

- **TF-IDF(Term Frequency - Inverse Document Frequency):** This is defined for a word in a dataset for a document. This is calculated as the product of the TF and IDF values. What this basically does is that multiplies the importance of that particular word in the document times the importance of the word over the entire dataset. So what happens is that common words have very less importance when seen over the entire dataset. Even if they are present in a larger fraction in a particular document, the word only present in that document a few times gets higher preference as its importance over the dataset is more. This is what is also desirable.

  For a given word "A", document "B" and dataset "D"
  $$TF\_IDF_{A,B,D} = TF_{A,B} * \times IDF_{A,D}$$

- **TF-IDF Matrix:** The matrix is an $m \times n$ matrix where "m = number of emails" and "n = number of words" that are present in the entire dataset. Where the $\{i, j\}^{th}$ gives the importance of the j^{th} word for the i^{th} email. Since there will be many words which are present in only one document there will be many zeros in the matrix. So the TfidfVectorizer returns a sparse matrix rather than a normal matrix.

1.4 Binary Data for Naive-Bayes

While implementing Naive-Bayes, we should convert the tfidf data into binary data indicating the presence or absence of the word. This binary data is used to implement the Naive-Bayes algorithm.

1.5 CountVectoriser

The count vectoriser stores the frequencies of the word in the document this doesn't take the dependencies across the emails so it is equivalent to having the TF term of the TFIDF vectoriser.

2 About Dataset

2.1 Description

This dataset contains a collection of email text messages, labeled as either spam or not spam. Each email message is associated with a binary label, where "1" indicates that the email is spam, and "0" indicates that it is not spam. The dataset is intended for use in training and evaluating spam email classification models.

2.2 Columns

- **text(Text):** This column contains the text content of the email messages. It includes the body of the emails along with any associated subject lines or headers.
- **spam(Binary):** This column contains binary labels to indicate whether an email is spam or not. "1" represents spam, while "0" represents not spam.

2.3 Source

This dataset is taken from kaggle named Spam email Dataset. The primary language in this email is English so the model trained by this data set best works on English test data.

3 Different Classifier Models

3.1 Naive-Bayes Classifier

Implemented a Naive-Bayes classifier which is a constructive model for the classification problem for this the data is converted into binary data if the tf-idf value is non-zero then it is replaced as 1 else 0. This model has a prior p and likelihood vectors π_0, π_1. The labels are predicted as given below.

$$label = \begin{cases} 1 & \text{if } p\prod_{i=1}^{d} \pi_{1i}^{y_i}\pi_{0i}^{1-y_i} \geq (1-p)\prod_{i=1}^{d} \pi_{0i}^{y_i}\pi_{1i}^{1-y_i} \quad (1) \\ 0 & \text{if } p\prod_{i=1}^{d} \pi_{1i}^{y_i}\pi_{0i}^{1-y_i} < (1-p)\prod_{i=1}^{d} \pi_{0i}^{y_i}\pi_{1i}^{1-y_i} \quad (2) \end{cases}$$

Applying log to the above classifier we get a linear classifier whose prediction is as follows.

$$label = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} y_i \log\frac{\pi_{1i}(1-\pi_{0i})}{\pi_{0i}(1-\pi_{1i})} + \sum_{i=1}^{d} \log\frac{1-\pi_{1i}}{1-\pi_{0i}} + \log\frac{p}{1-p} \geq 0 \quad (3) \\ 0 & \text{otherwise} \quad (4) \end{cases}$$

3.2 Gaussian Naive-Bayes Classifier

It is a variant of the Naive-Bayes Classifier but with multi-dimensional Gaussian distribution. Here we will be assuming that both the probability distributions are having the same covariance. This model is not advisable for classification of this data because it is computationally heavy as the data is having around 33,000 features making the calculation of the inverse of the matrices very difficult.

- The Gaussian Naive-Bayes classifier can have Linear Decision Boundary when the covariance of both the labels are equal.
- The Gaussian Naive-Bayes classifier can have Quadratic Decision Boundary when the covarianve of both the labels are unequal.

3.3 Perceptron Classifier

Perceptron is a basic yet a powerful classifier. The update rule of perceptron is very simple.
Update Rule: If $(w_t^T.x_i).y_i < 0$ (assuming the labels are -1, 1) then $w_{t+1} = w_t + x_i y_i$.

This can be implemented in two ways one is traverse through the dataset completely keep on updating without breaking and the other is breaking from the loop and start over whenever and error is encountered. The below plot shows how the number of errors varies as the Perceptron progresses in both kind of updation.

There are a few important conclusions that we can derive from the below plots.

- The plot to the left shows the errors including the initialisation whereas the plot to the right shows the errors after the first iteration.

- The error without any iteration is too high be cause we haven't used any data and predicted 1 for all the data points.

- Since we used the information provided by the data to some extent for the first time there is a large drop from around 3500 errors to 37 and 1200 errors.

- The decrease from this step with iterations is not as much as the first decrease and keeps on decreasing as the iterations increases.

- The fluctuations in figure 2 are more because the updation for a particular point may cause many other points as errors which is less likely if we did not break as such will be fixed in the same iteration.

- Observe Fig2 and Fig4 the fluctuations in Fig4 is more compared to Fig2 this is because the TFIDF captures more information compared to the Count so the effect of one error is not as drastic as count.

- If we observe the error convergence, that is also better for the TFIDF compared to the count vectoriser. So it would be better to use the TFIDF data for predicting labels.

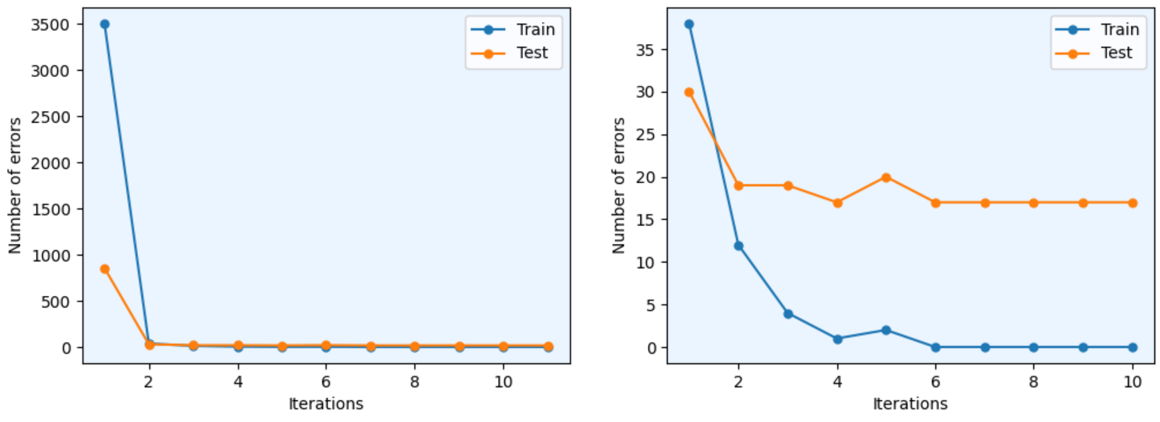

Figure 1: Perceptron Errors vs Iterations without breaking tfidf

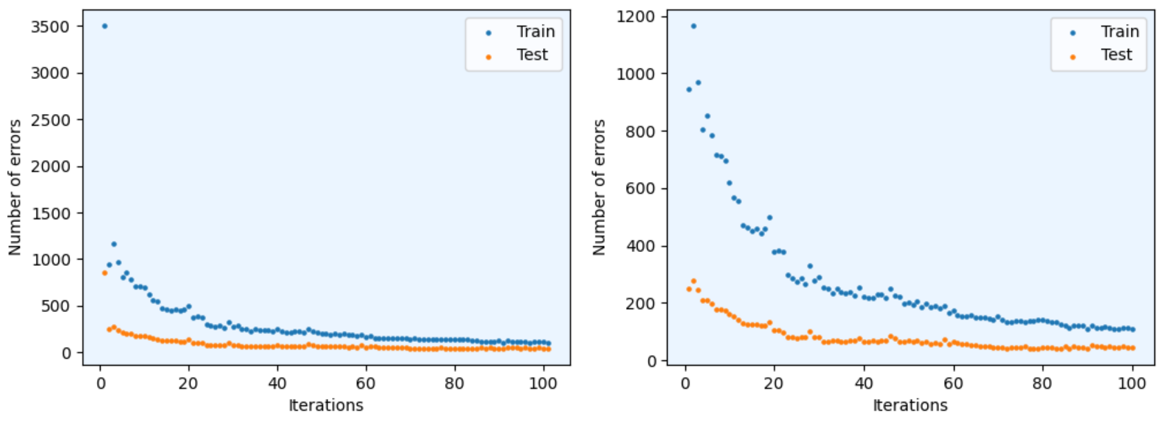

Figure 2: Perceptron Errors vs Iterations with breaking tfidf

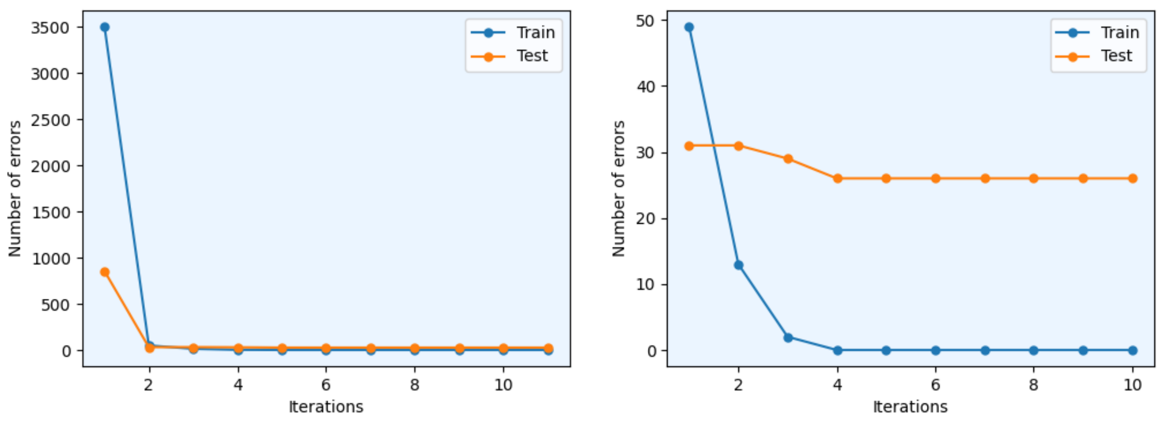

Figure 3: Perceptron Errors vs Iterations without breaking count

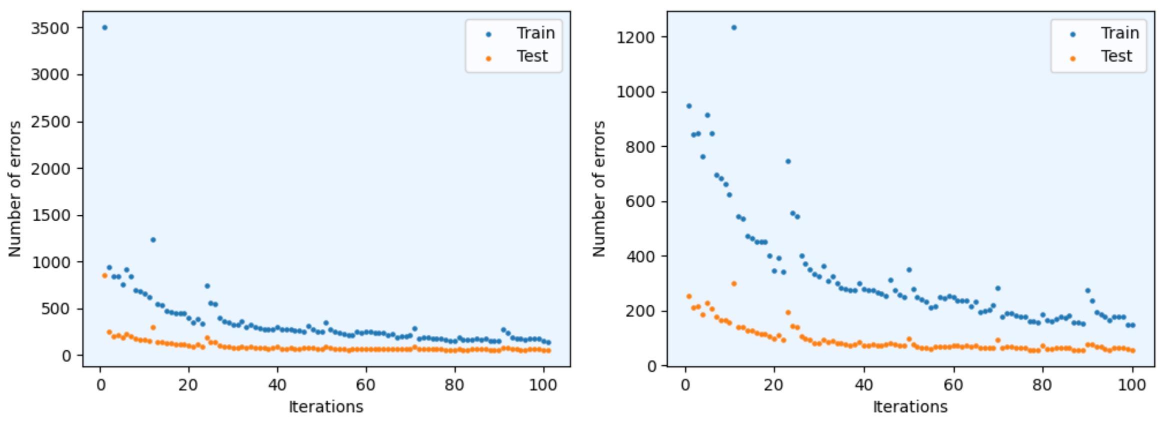

Figure 4: Perceptron Errors vs Iterations with breaking count

3.4 Logistic Regression Classifier

The logisitic regression is a function where we try to maximise the likelihood of the prediction. The likelihood function is given by

$$\mathcal{L}(w, data) = \prod_{i=1}^{n} (g(x_i)^{y_i}).(1 - g(x_i))^{1-y_i} \quad ,\text{where } g(x) = \frac{1}{1 + e^{-w^T x}}$$

$$\log \mathcal{L}(w, data) = \sum_{i=1}^{n} (1 - y_i)(-w^T x_i) - log(1 + e^{-w^T x_i})$$

$$\nabla \log \mathcal{L}(w) = \sum_{i=1}^{n} x_i \left(y_i - \frac{1}{1 + e^{w^T x_i}} \right) \Rightarrow w_{t+1} = w_t + \eta_t \nabla \log \mathcal{L}(w)$$

We will take the loglikelihood and we will maximise it using the gradient ascent. The below plots shows variation of errors with iterations for different step value of values 0.01, 0.1 and 1. Among these the least error on test data is observed when step size is 0.1 with only 5 errors. Intuitively this is like weighting the importance of the points based on how close they are to the actual label.

Few important observations can be made from the below plots.

- Here also we can observe a similar dip in the first iteration as seen in perceptron also. But this is more closer to the optimal compared to perceptron because we are preferrably adding based on their weights where as in perceptron we are directly adding $x_i y_i$ so which is capturing less information so more Logistic Regression works better.
- Also the decrement is smoother in case of smaller step-size we can see from the plots. This gradient is more smoother than compared to perceptron updation.
- Number of errors is minimum in case of step-size 0.1 which is 5 errors
- If we observe the plots in Fig5 as the step size is very less the classifier didn't reach close to the optimum and is still in the process of converging to the best classifier.
- From this we can observe that the step size should be chosen optimally so that the converging happens and at the same time overfitting should not happen on the train data so 0.1 is chosen as the step size for the Logistic Regression Classifier.

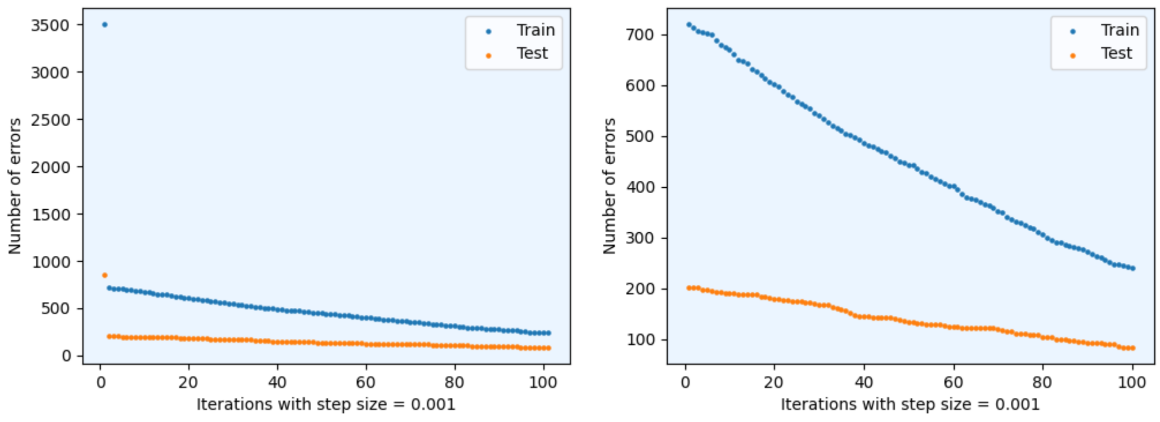

Figure 5: Logistic Regression Errors vs Iterations on TFIDF

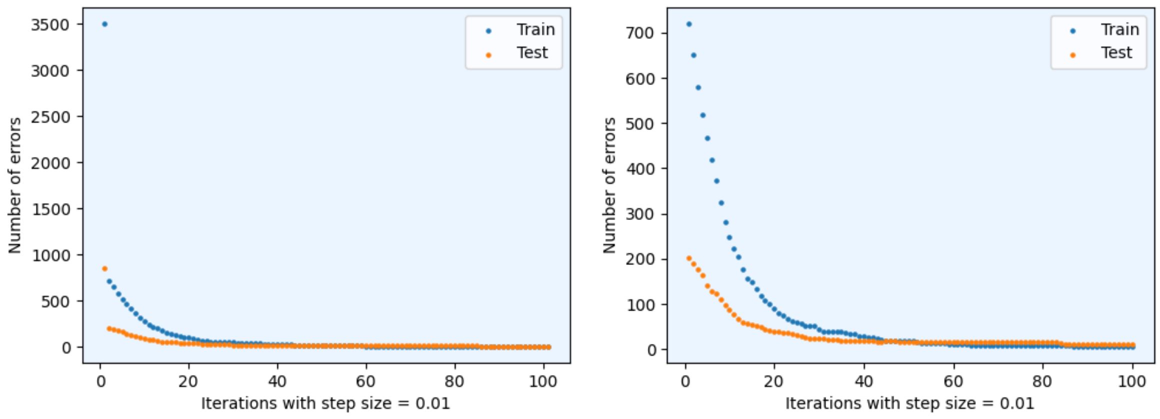

Figure 6: Logistic Regression Errors vs Iterations on TFIDF

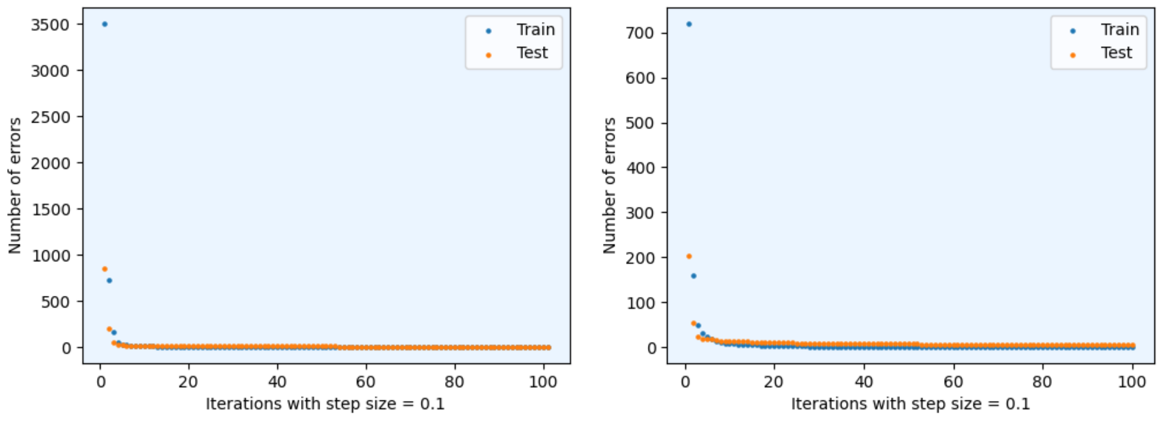

Figure 7: Logistic Regression Errors vs Iterations on TFIDF

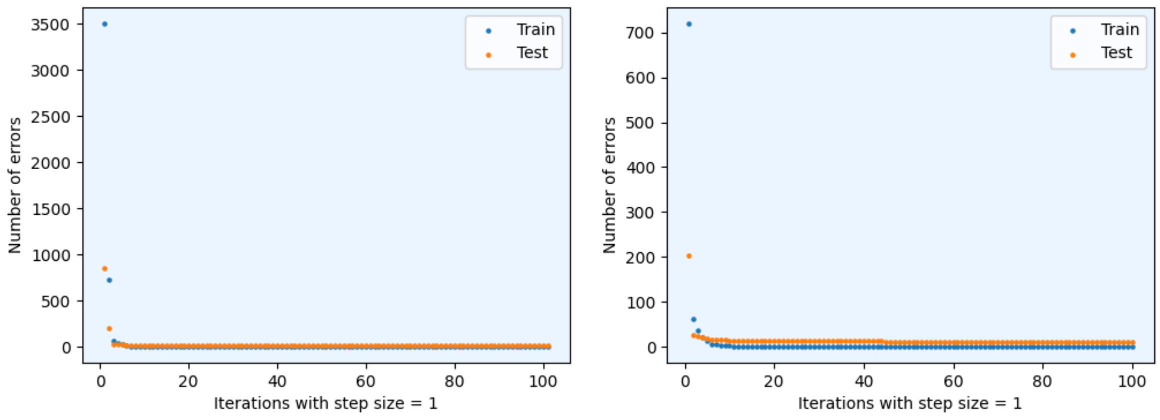

Figure 8: Logistic Regression Errors vs Iterations on TFIDF

3.5 Support Vector Machine

This is implemented using python libraries sklearn with the below piece of code.

```
SVM_classifier = svm.LinearSVC(dual='auto')
SVM_classifier.fit(vectorised_train_data, train_labels)
SVM_predicted_labels = SVM_classifier.predict(vectorised_test_data)
print(calculate_error(SVM_predicted_labels, test_labels))
```

This is giving SVM classifier with the train data and the train labels and predicting the labels for the test data.

3.6 Mixed Majority Classifier

After getting the Naive-Bayes, Perceptron and Logistic Regression the label for a data point is the majority of these 3. This gave a better predicition on the test data. This tells us having more decision classifiers increases the chances of better predictions. The below code shows how the labels are chosen.

```
Best_predicted_labels = 1*(np.sum(np.array([Perceptron_predicted_labels,
LR_predicted_labels, NB_predicted_labels, SVM_predicted_labels]), axis=0) >= 2)
test_error = calculate_error(Best_predicted_labels, test_labels)
print(test_error)
```

4 Results and Inferences

4.1 Logistic Regression

The below table gives the summary of Logistic Regression when ran for 100 iterations of the gradient ascent.

Data	η (step size)	Time	Accuracy	
			Train split	Test split
tfidf_data	1	17s	100%	99.12%
	10^{-1}	17s	100%	99.56%
	10^{-2}	17s	99.86%	98.95%
	10^{-3}	17s	94.76%	92.67%
count_data	10^{-3}	–	–	–
	10^{-4}	56.4s	99.60%	98.77%
	10^{-5}	58.1s	98.29%	97.55%
	10^{-6}	57.1s	89.65%	89.17%

- The time for training is less in case of TFIDF data compared to the count data.
- Accuracy is also better in case of TFIDF as it captures more information of the data set including the dependencies across the files.
- If we vary the step size then there is a maximum accuracy somewhere around 0.1 and then decreases with decrease in step size, which is observed on both the test and train and while using the TFIDF and the Count data.
- The conclusion is that we should use TFIDF vectoriser with step size of 0.1 for our final classifier.

4.2 Perceptron

The below table gives the summary of Perceptron when ran for 10, 100 iterations without and with breaking at error encounter.

Data	Breaking Status	Time	Accuracy	
			Train split	Test split
tfidf_data	with breaking	10.2s	97.62%	96.07%
	without breaking	1.9s	100%	98.51%
count_data	with breaking	13.2s	96.81%	95.02%
	without breaking	2.5s	100%	97.73%

- The training time and accuracy is better in case of TFIDF data compared to the count data as it captures more information regarding the dataset.

- The algorithm performs better if we implement the classifier without breaking when an error is encountered. This might be because if we loop from the start there will be a biased updation at the earlier datapoints.
- So here it is better to use Perceptron on TFIDF data without breaking when error is encountered.

4.3 Overall Analysis

Data	Perceptron	Logistic Regression	Naive-Bayes	SVM	Mixed
train_data	100%	100%	98.21%	100%	100%
test_data	98.51%	99.56%	97.38%	99.30%	99.56%

- Expect Naive-Bayes others are performing well on the train data. This is beacuse Naive-Bayes just uses the existence factor of a word which loses a lot of information compared to the other algorithms.
- Perceptron being a simple classifier is performing better than Naive yet a bit less accurate compared to the Logistic Regression and SVM.
- Logistic Regression and SVM are almost performing on the same level.
- The majority classifier performs as good as Logistic Regression as this will take the different possible classifiers into consideration.
- From the above analysis it will be better if we use the **Mixed Majority Classifier**.

5 Testing the Data

5.1 How data is used for the above plots?

From the data that is collected 80% of it is used as the train data and the remaining 20% is used as the test data. This is to analyse the performance of the classifiers. But when it comes to predicting a new email more the training data better would be the prediction so the entire data is used to train the Classifier that runs on the new email.

5.2 How to test the data? and See the implementation?

The testing and analysis can be seen in the "A3.ipynb" where as to test on emails, take all the emails in a directory named "test" with emails named as "email0.txt, email1.txt, . . . ". Run the python file "testing.py" the predicted values various classifiers on test data can be found in "predicted_labels.csv".

5.3 What is the preferred data for testing?

Since the train data is in English this classifier will better work on emails that are in English. Emails of other languages might not be predicted properly as the transform ignores non-english words.