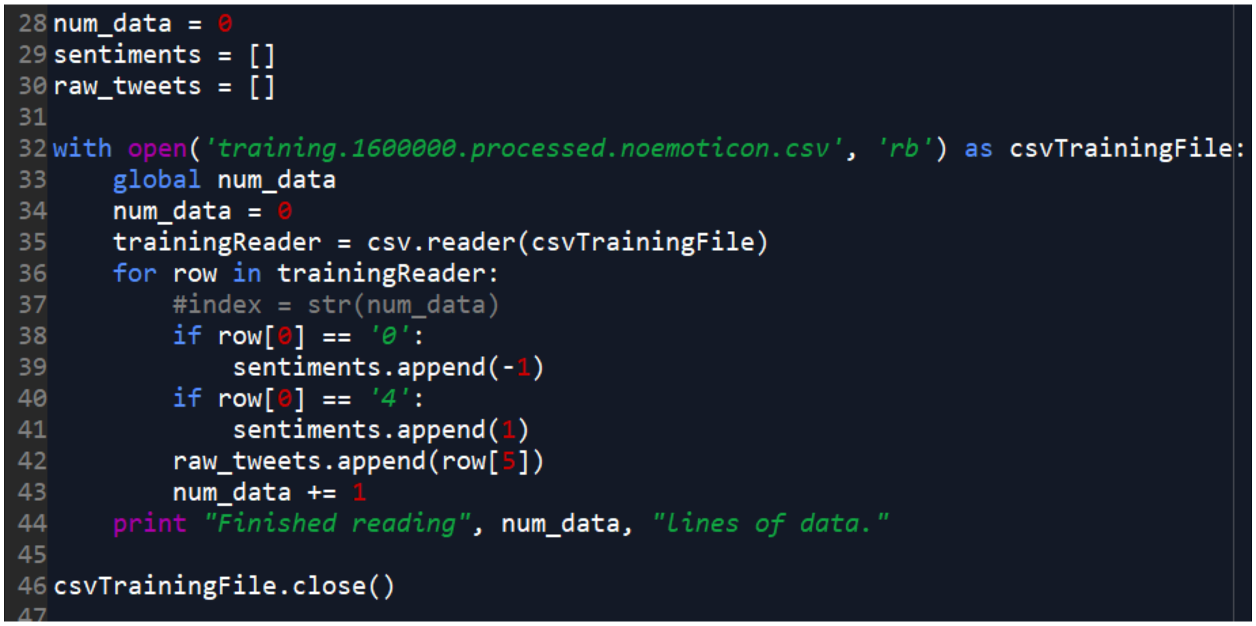
Team Members: **Pedro Rizo, Wei Wu** Oct. 31th, 2017

Project 2 Report

Please see the code for implementation on each of the different questions.
It's useful to note that the numpy files corresponding to the trained SVM's are included in the zipfile in case they are needed. Also, the error information over number of iterations is also included.

Question 1.1 – Reading the training data

We used a csv.reader to read the data from the training data set. While reading from each row we created to lists. The first one is named *raw_tweets*. This list basically contains all the raw tweets as read from the csv file. The *sentiments* list is basically just storing a '-1' if the value of the sentiment for the tweet is '0' or a '1' if the value of the sentiment for the tweet is '4'.



```python
num_data = 0
sentiments = []
raw_tweets = []

with open('training.1600000.processed.noemoticon.csv', 'rb') as csvTrainingFile:
    global num_data
    num_data = 0
    trainingReader = csv.reader(csvTrainingFile)
    for row in trainingReader:
        #index = str(num_data)
        if row[0] == '0':
            sentiments.append(-1)
        if row[0] == '4':
            sentiments.append(1)
        raw_tweets.append(row[5])
        num_data += 1
    print "Finished reading", num_data, "lines of data."

csvTrainingFile.close()
```

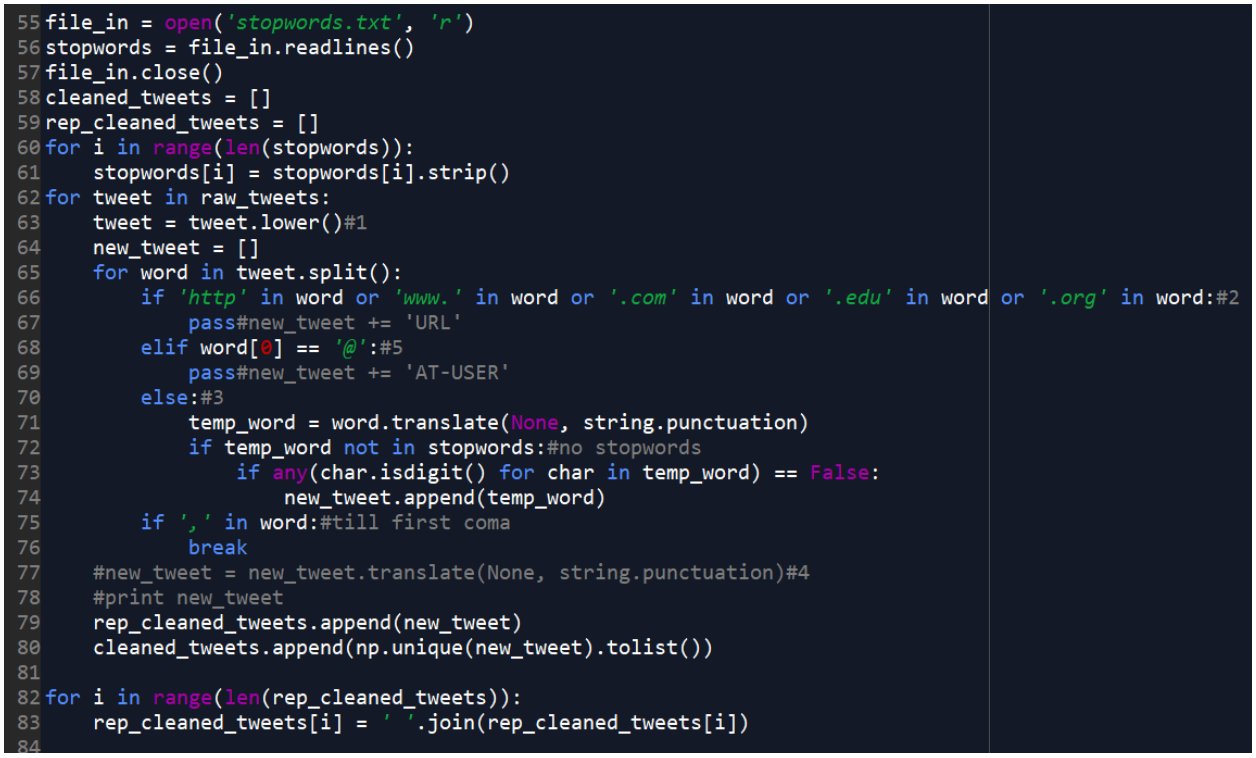
Question 1.2 – Cleaning up the data

For this question, we start with reading in all the *stopwords* file, after which we manage to clean the tweets.

For each tweet in all the tweets we collected, we first convert all its characters to lowercase, and then split it to a series of words. Then we create an empty list called *new_tweet* to store the words in the original tweet. For each word, we determine a word to be a URL if it contains any of the patterns such as 'http', 'www.', '.com' and so forth. If it is a URL, it is removed by not being appended to the variable *new_tweet*, for that URL is included in *stopwords* and its conversion to

‘URL’ sill lead to a removal anyways. If a word starts with ‘@’, we consider it as ‘AT-USER’, which is also included in *stopwords* and is thus not appended to the variable *new_tweet*.

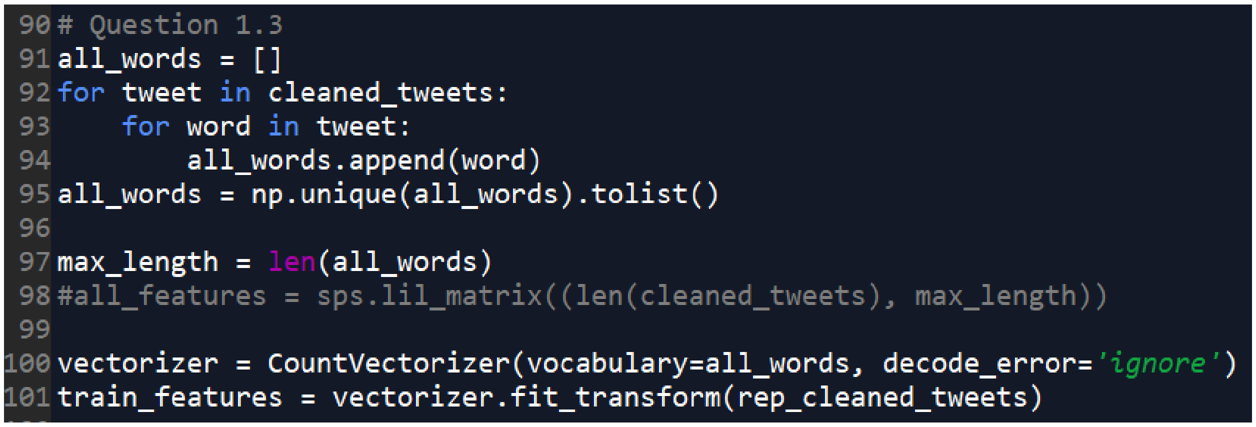
If a word is not a URL or AT-USER, we remove all its punctuations in its copy and append it to the *new_tweet* if it is not in the *stopwords*. After which we check if there is a coma in this word, if yes, then we stop appending words from this tweet. This collection of tweets is called rep_cleaned_tweets because it has repetitive. We at the same time obtain a collection called cleaned_tweets by appending the unique words of each new_tweet.
The last thing we did for this question is to join all the words in each tweet from rep_cleaned_tweet to make rep_cleaned_tweet a list of strings for the convenience of following steps.



```python
file_in = open('stopwords.txt', 'r')
stopwords = file_in.readlines()
file_in.close()
cleaned_tweets = []
rep_cleaned_tweets = []
for i in range(len(stopwords)):
    stopwords[i] = stopwords[i].strip()
for tweet in raw_tweets:
    tweet = tweet.lower()#1
    new_tweet = []
    for word in tweet.split():
        if 'http' in word or 'www.' in word or '.com' in word or '.edu' in word or '.org' in word:#2
            pass#new_tweet += 'URL'
        elif word[0] == '@':#5
            pass#new_tweet += 'AT-USER'
        else:#3
            temp_word = word.translate(None, string.punctuation)
            if temp_word not in stopwords:#no stopwords
                if any(char.isdigit() for char in temp_word) == False:
                    new_tweet.append(temp_word)
        if ',' in word:#till first coma
            break
    #new_tweet = new_tweet.translate(None, string.punctuation)#4
    #print new_tweet
    rep_cleaned_tweets.append(new_tweet)
    cleaned_tweets.append(np.unique(new_tweet).tolist())

for i in range(len(rep_cleaned_tweets)):
    rep_cleaned_tweets[i] = ' '.join(rep_cleaned_tweets[i])
```

Question 1.3 – Extracting the features

For this step, we first create a list called all_words which will contain all the valid words that ever appeared in the cleaned tweets and eventually determine the length of our features.
Then we called in the sklearn module and used its feature extractor to extract our features which is then store in the form of a sparse matrix.



```python
# Question 1.3
all_words = []
for tweet in cleaned_tweets:
    for word in tweet:
        all_words.append(word)
all_words = np.unique(all_words).tolist()

max_length = len(all_words)
#all_features = sps.lil_matrix((len(cleaned_tweets), max_length))

vectorizer = CountVectorizer(vocabulary=all_words, decode_error='ignore')
train_features = vectorizer.fit_transform(rep_cleaned_tweets)
```

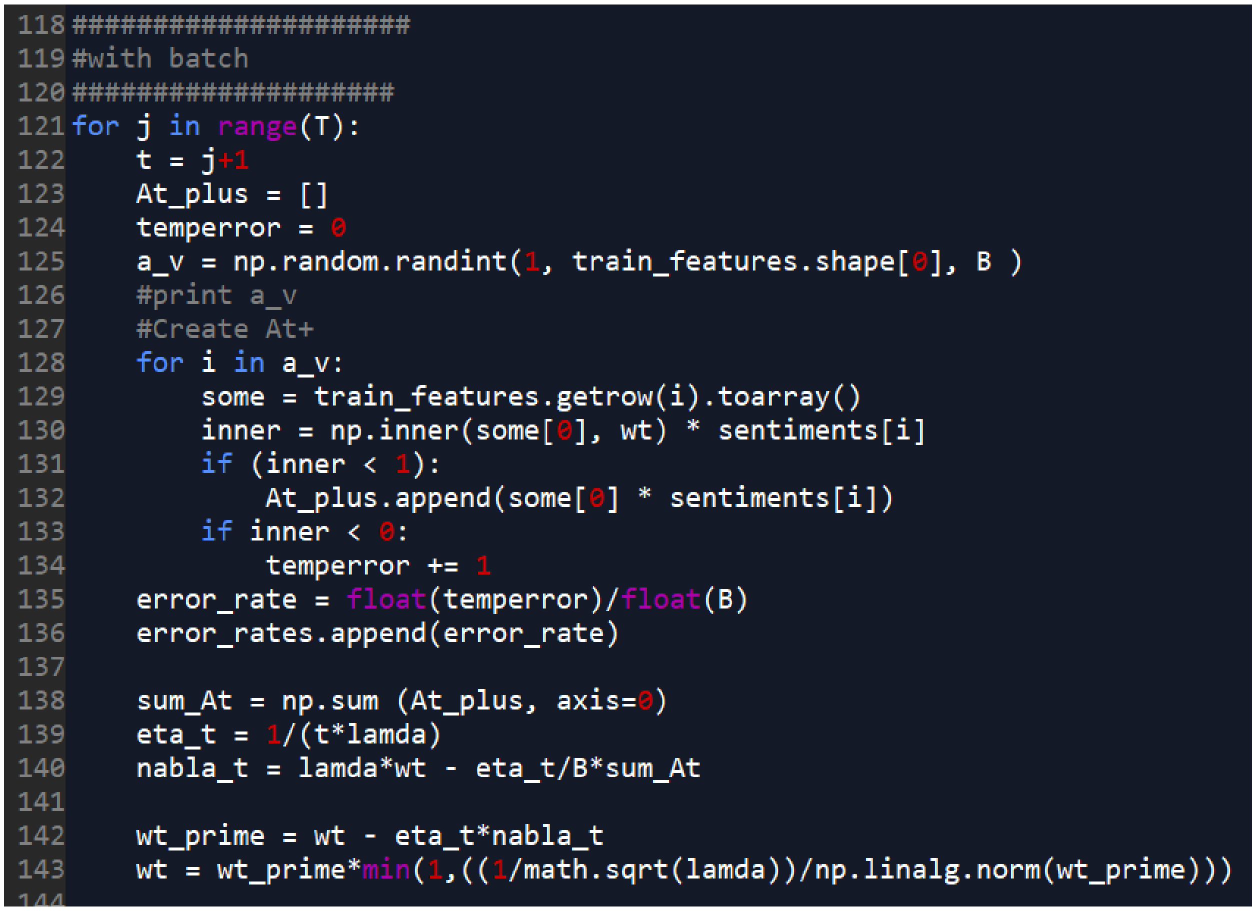
Question 1.4 Use PEGASOS to train an SVM on the features extracted above. Make a plot of training error vs number of iterations.

As seen in class, we implemented the PEGASOS algorithm with the use of a mini-batch to train an SVM for the features extracted from the dataset of tweets.
The results were obtained using 2000 iterations (T), each with a batch size of 200 tweets (B) and a value of lambda equal to 0.001.

The implementation is very straightforward. In each iteration we extract B random sample tweets form the entire set of features (variable named '*train_features*'). Then, for each tweet in the mini-batch, we check if: $y\langle w_t, x\rangle < 1$ to create A_t^+. Also, for each member of the mini-batch we are checking if the prediction is correct, and if not, adding to the error for that iteration.

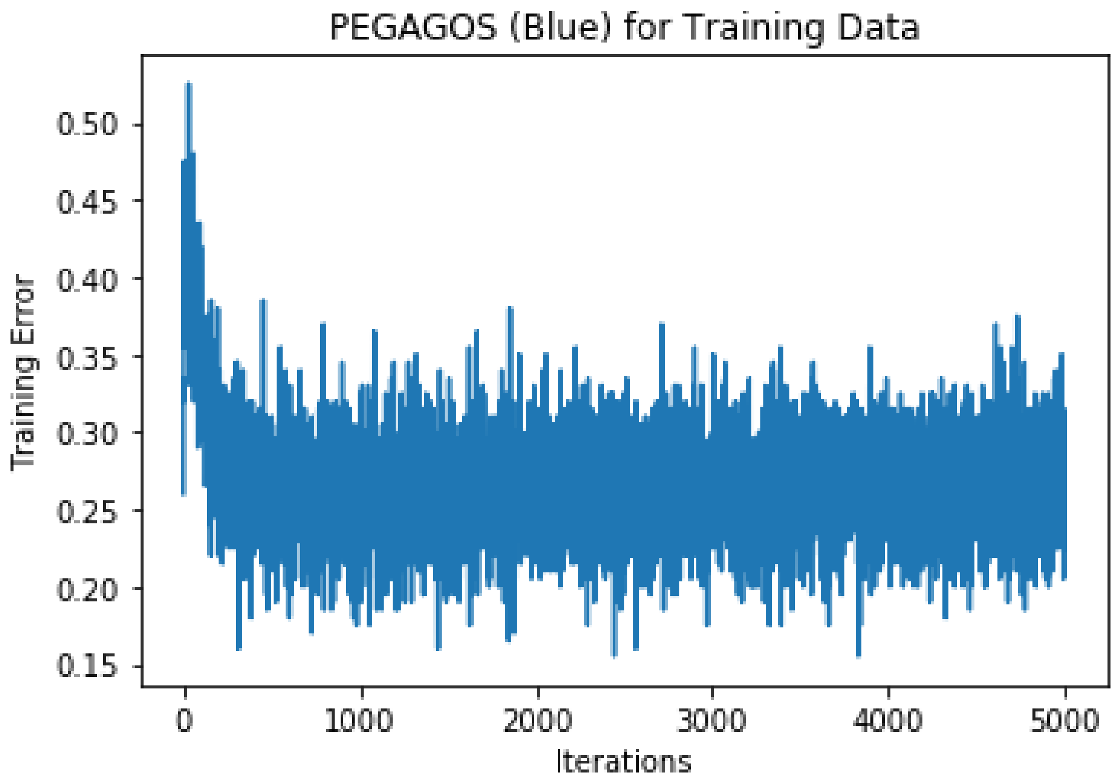
Then, we are just adding all the elements in A_t^+ and this result is used to calculate ∇_t.
Then we just perform the update and the projection of w_t and perform the next iteration.
At the end of all the iterations we obtain our trained algorithm.

The code for this is shown below.



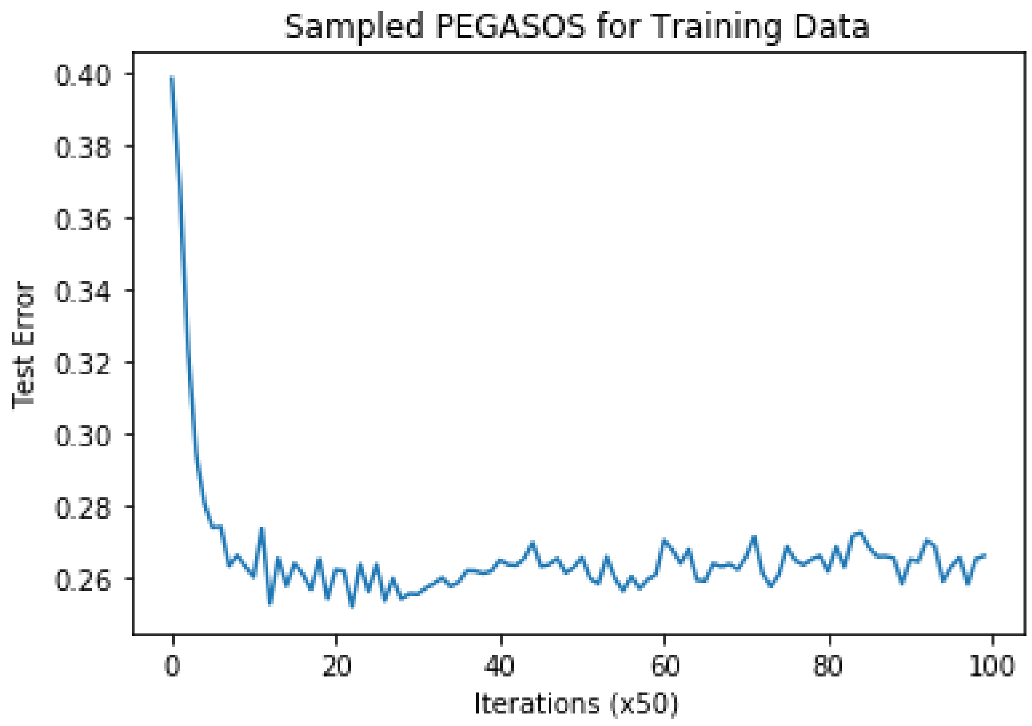
```
####################
#with batch
###################
for j in range(T):
    t = j+1
    At_plus = []
    temperror = 0
    a_v = np.random.randint(1, train_features.shape[0], B )
    #print a_v
    #Create At+
    for i in a_v:
        some = train_features.getrow(i).toarray()
        inner = np.inner(some[0], wt) * sentiments[i]
        if (inner < 1):
            At_plus.append(some[0] * sentiments[i])
        if inner < 0:
            temperror += 1
    error_rate = float(temperror)/float(B)
    error_rates.append(error_rate)

    sum_At = np.sum (At_plus, axis=0)
    eta_t = 1/(t*lamda)
    nabla_t = lamda*wt - eta_t/B*sum_At

    wt_prime = wt - eta_t*nabla_t
    wt = wt_prime*min(1,((1/math.sqrt(lamda))/np.linalg.norm(wt_prime)))
```

The results of the trained SVM is shown in the plot shown below.



We can see from the plot that at the very start of the iterations, the values go to the randomness of 0.5. But then the plot starts to lower and converge to a point around 0.25 of error rate. It is worth noting that this error at each iteration was sampled from the same mini-batch for each iteration, and hence the amount of variation in the error.

If we take the average of the error every 50 iterations, we can obtain a clearer picture of the behavior of the algorithm.



Also, we can extract some useful information from the values of our '*error_rates*' variable.

Error for last batch: 0.225

Average error: 0.26645

Finally, we tested the SVM with the entire dataset of 1.6 million tweets. The obtained results is appropriate considering the time constraints and scope of the current project.

Error over the entire training dataset = 0.26027

It is important to note that in our code, the final trained SVM for this part is called "wt".

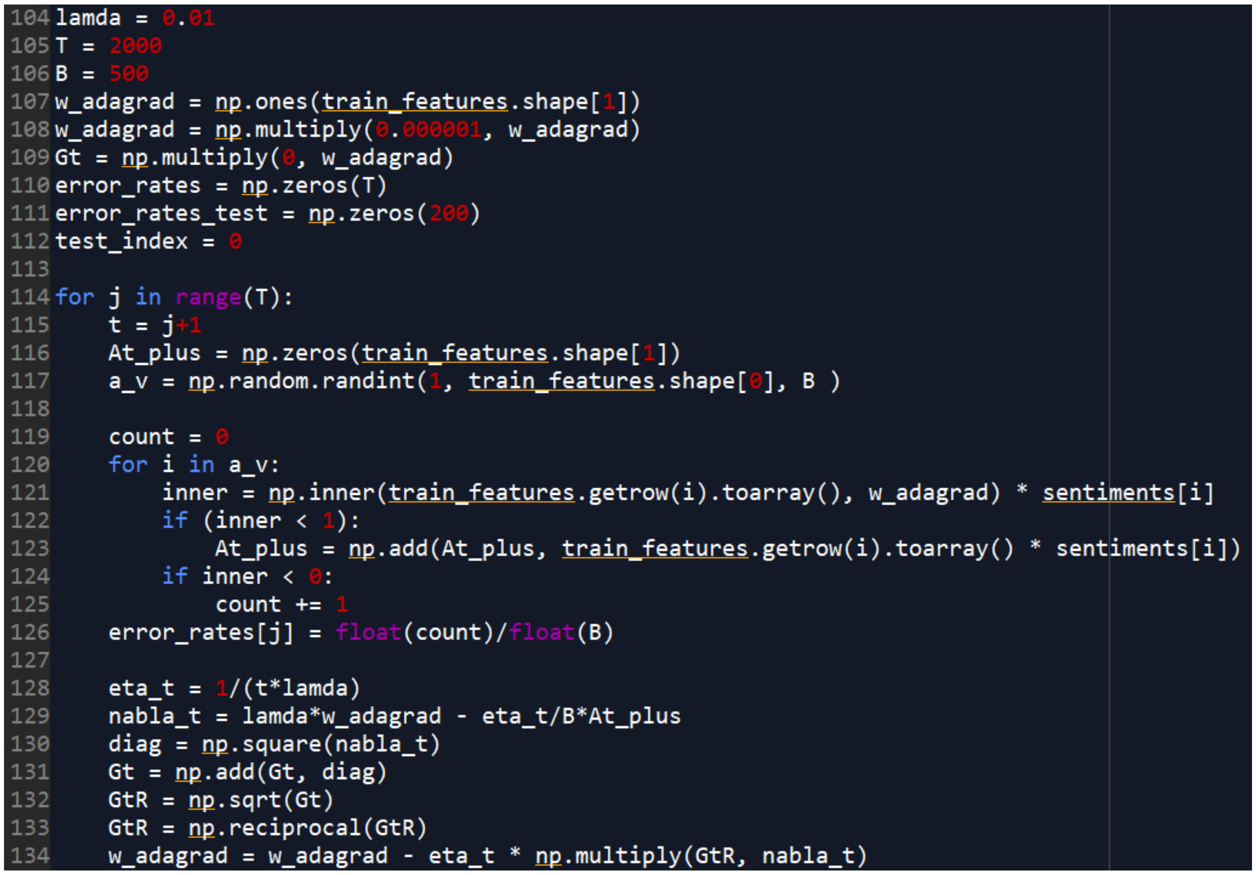
Question 1.5 - Use AdaGrad to train an SVM on the features extracted above. Make a plot of training error vs number of iterations.

As seen in class, we implemented the AdaGrad following very similar steps as in the PEGASOS algorithm.

The results were obtained using 500 iterations (T), each with a batch size of 500 tweets (B) and a value of lambda equal to 0.01.

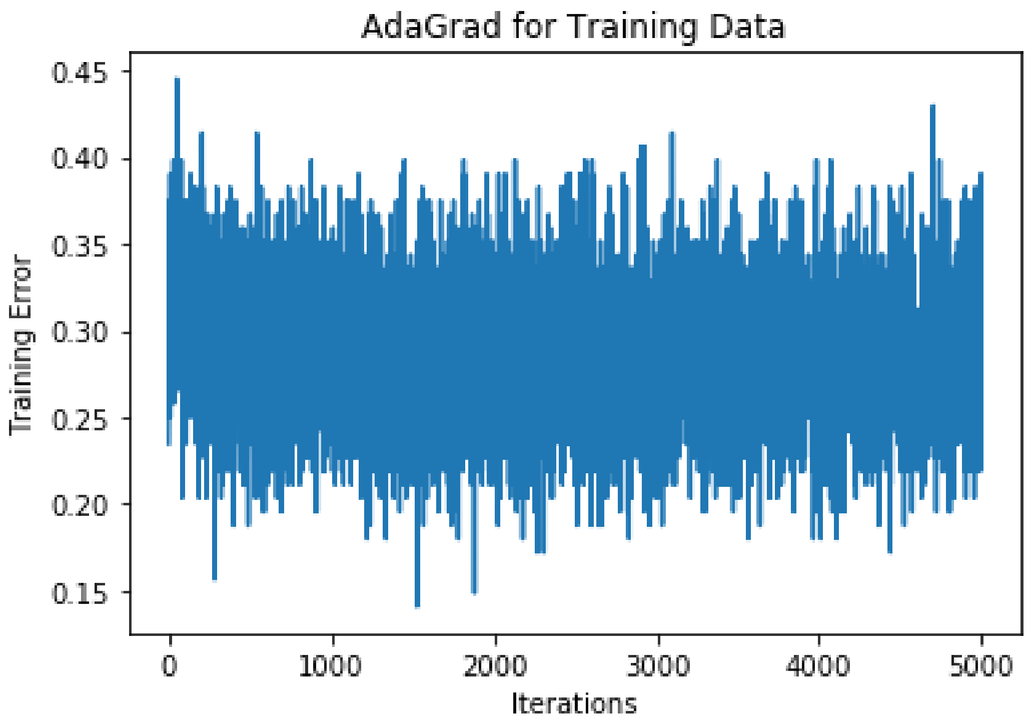
The difference in the implementation here is that for each iteration we are maintaining the sum of the calculation of the squares from each iteration gradient. This is G_t. Then, in each iteration we are calculating the square root of this G_t and then calculating the reciprocal, or $\frac{1}{G_t}$. Finally, we are incorporating this term into our update step.

The code for this is shown below.



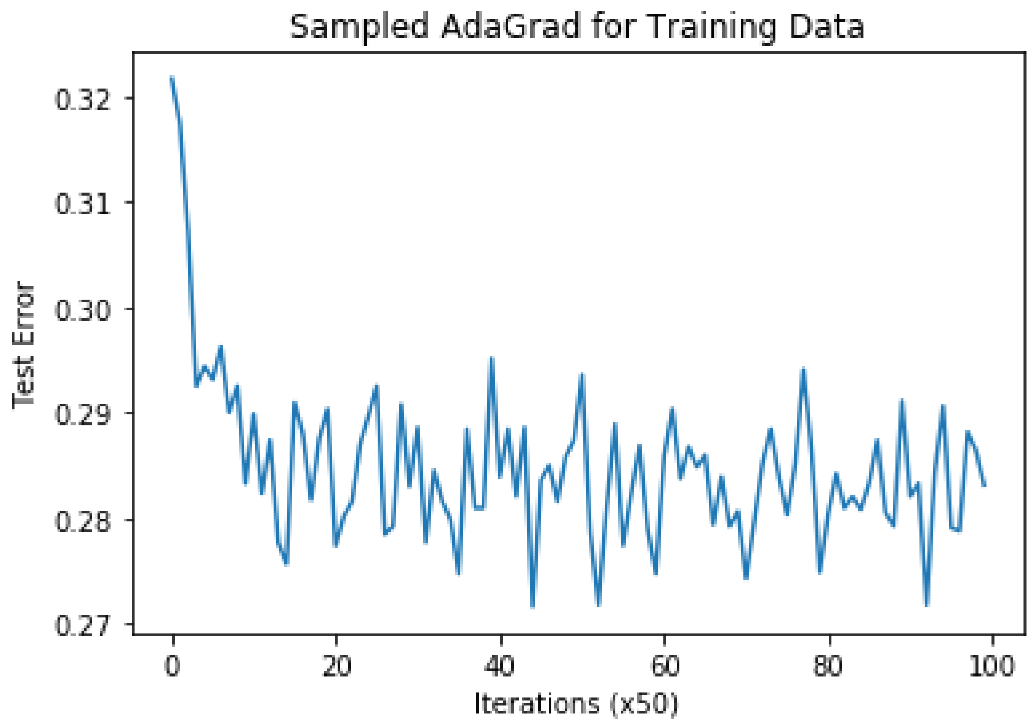
```python
lamda = 0.01
T = 2000
B = 500
w_adagrad = np.ones(train_features.shape[1])
w_adagrad = np.multiply(0.000001, w_adagrad)
Gt = np.multiply(0, w_adagrad)
error_rates = np.zeros(T)
error_rates_test = np.zeros(200)
test_index = 0

for j in range(T):
    t = j+1
    At_plus = np.zeros(train_features.shape[1])
    a_v = np.random.randint(1, train_features.shape[0], B )

    count = 0
    for i in a_v:
        inner = np.inner(train_features.getrow(i).toarray(), w_adagrad) * sentiments[i]
        if (inner < 1):
            At_plus = np.add(At_plus, train_features.getrow(i).toarray() * sentiments[i])
        if inner < 0:
            count += 1
    error_rates[j] = float(count)/float(B)

    eta_t = 1/(t*lamda)
    nabla_t = lamda*w_adagrad - eta_t/B*At_plus
    diag = np.square(nabla_t)
    Gt = np.add(Gt, diag)
    GtR = np.sqrt(Gt)
    GtR = np.reciprocal(GtR)
    w_adagrad = w_adagrad - eta_t * np.multiply(GtR, nabla_t)
```

The results of the trained SVM is shown in the plot shown below.

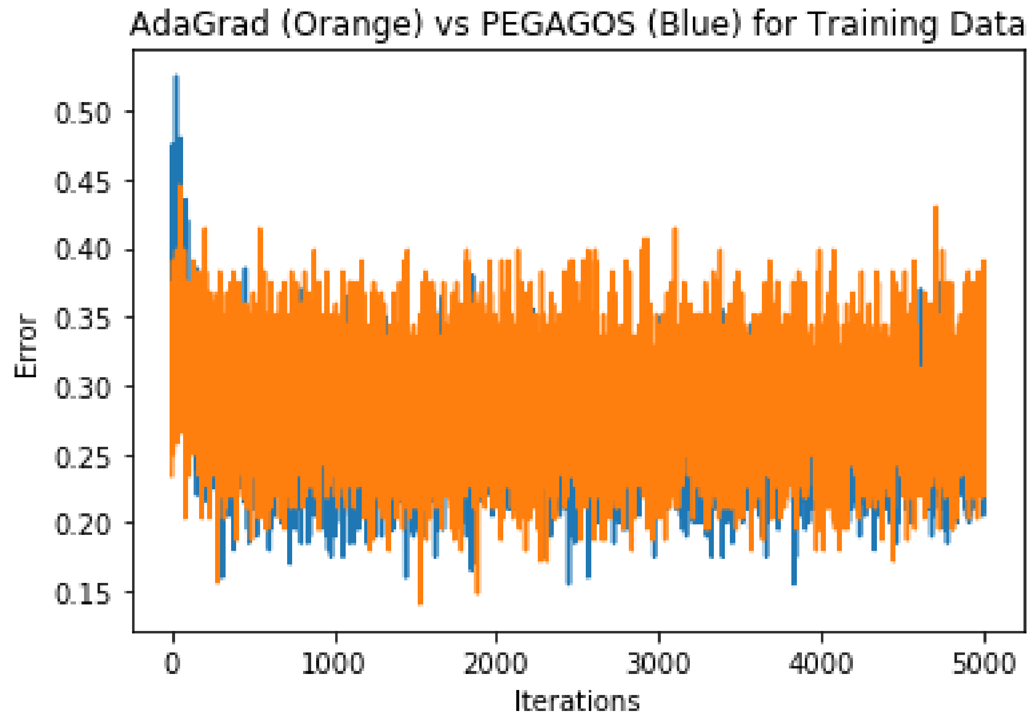


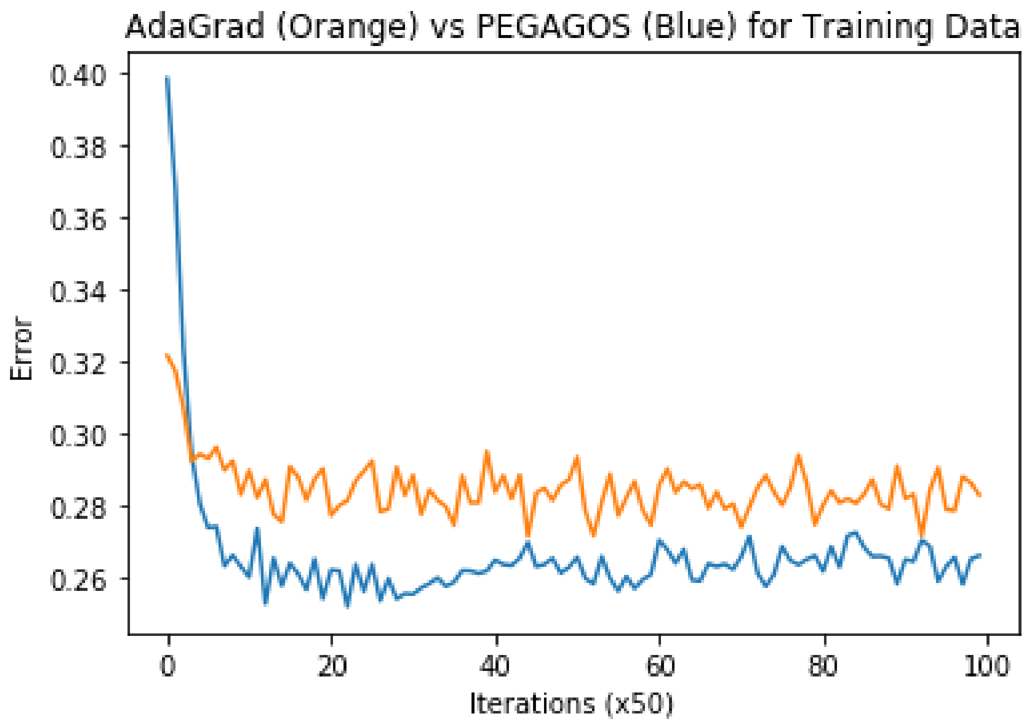
We can see from the plot that at the very start of the iterations, the values go to the randomness of 0.5. But then the plot starts to lower and converge to a point around 0.25 of error rate. It is worth noting that this error at each iteration was sampled from the same mini-batch for each iteration, and hence the amount of variation in the error and the noise in it.

If we take the average of the error every 50 iterations, we can obtain a clearer picture of the behavior of the algorithm.



If we compare this plot with the training of PEGASOS, we can see they both have a similar convergence.





Also, we can extract some useful information from the values of our '*error_rates*' variable.

Error for last batch: 0.2734375

Average error: 0.28493

Finally, we tested the SVM with the entire dataset of 1.6 million tweets. The obtained results are appropriate considering the time constraints and scope of the current project.

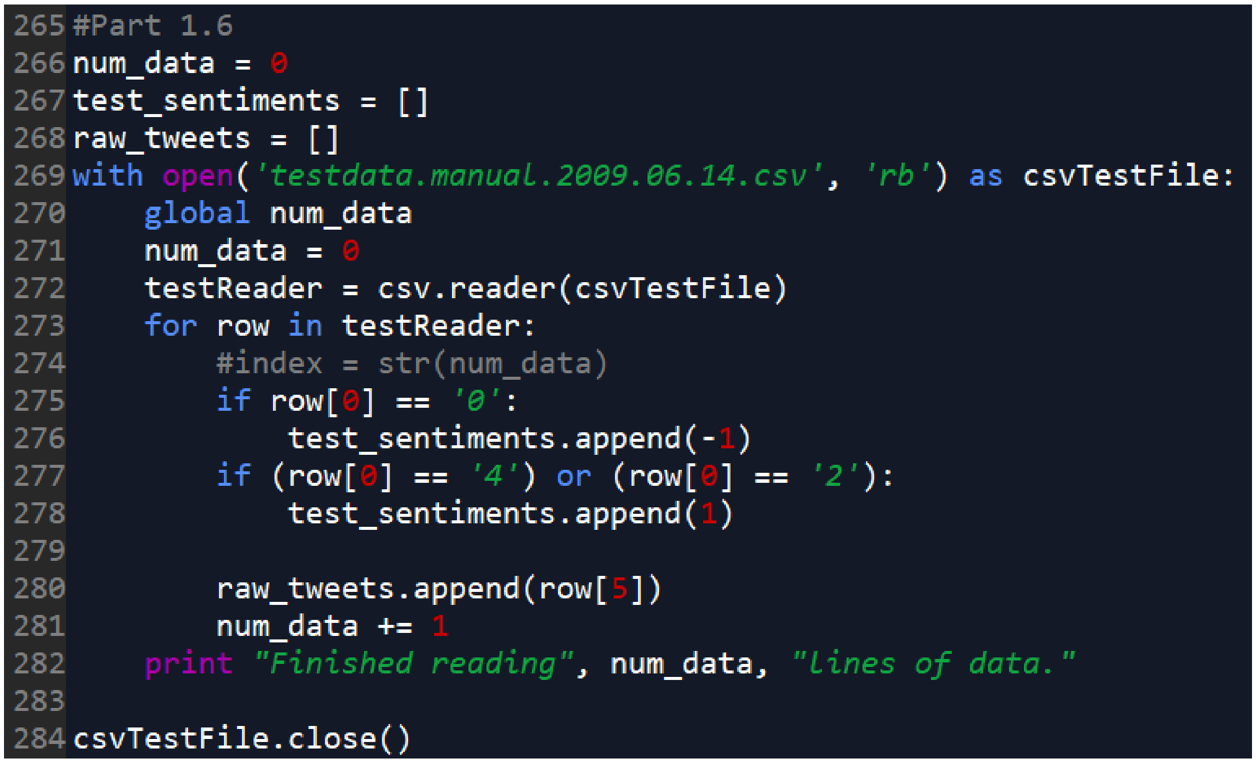
Error over the entire training dataset = 0.27821

It is important to note that in our code, the final trained SVM for this part is called "w_adagrad".

Question 1.6 – Verifying results with the test data

To verify the results of the trained classifiers, we started by reading the Test dataset similar to what we did in 1.1. The only change in this point is that when reading the sentiments, the values with sentiment 2 and 4 are being saved as '1', while the values with sentiment 0 are being saved as '-1'. The rest of the processing and cleaning up of the tweets is done in the same way. To obtain the matrix of classifiers, we are using the exact same bag of words that we used to train the PEGASOS and AdaGrad classifiers to maintain uniformity. Hence, we are just fitting the feature words found in the test data set to our large bag of words.

It is worth mentioning that we are storing our sentiments in a variable named '*test_sentiments*' and our features for the test tweets in a variable named '*test_features*'

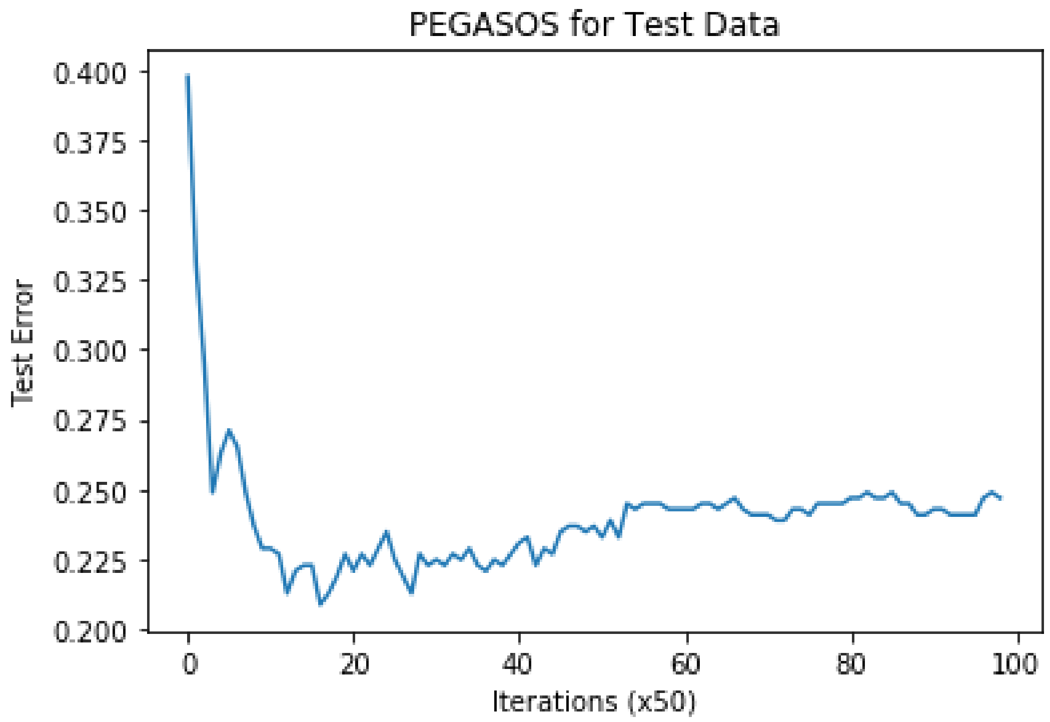


```python
#Part 1.6
num_data = 0
test_sentiments = []
raw_tweets = []
with open('testdata.manual.2009.06.14.csv', 'rb') as csvTestFile:
    global num_data
    num_data = 0
    testReader = csv.reader(csvTestFile)
    for row in testReader:
        #index = str(num_data)
        if row[0] == '0':
            test_sentiments.append(-1)
        if (row[0] == '4') or (row[0] == '2'):
            test_sentiments.append(1)

        raw_tweets.append(row[5])
        num_data += 1
    print "Finished reading", num_data, "lines of data."

csvTestFile.close()
```

To correctly report the training error per iteration for each of our classifiers, what we did is that while training the classifier, every 10 iterations we would do a pass over the entire test data with the current w_t. The results and the plot are shown below.

PEGASOS

The graph for the classifier obtained with the PEGASOS algorithm is shown below.

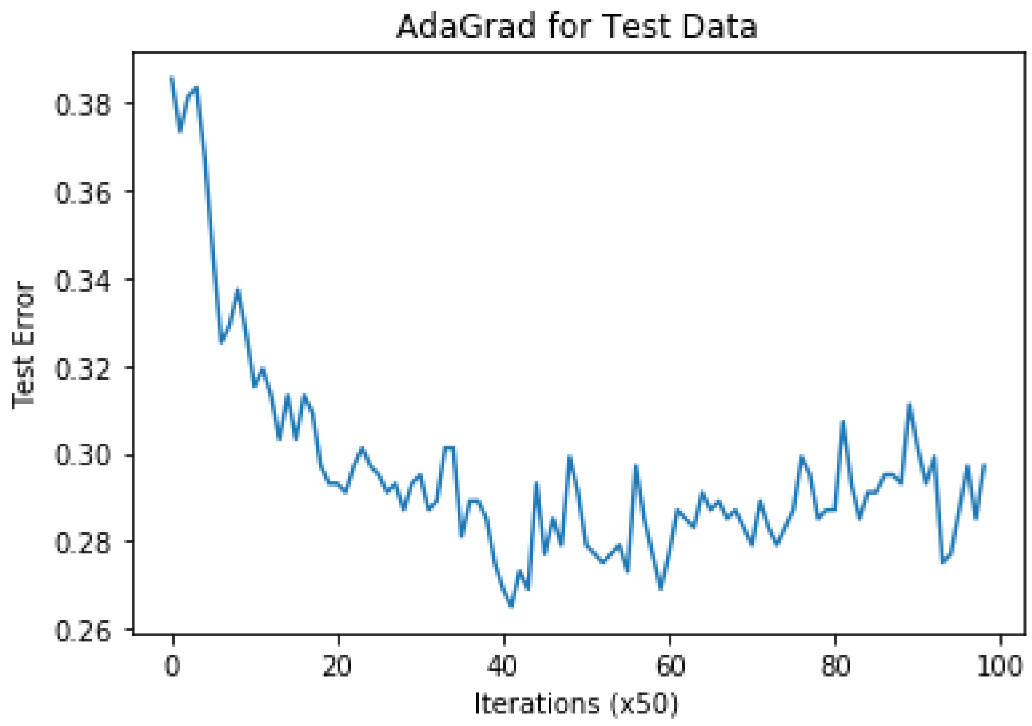


The results confirm that the classifier works very good.

Error for entire test dataset: 0.24899

AdaGrad

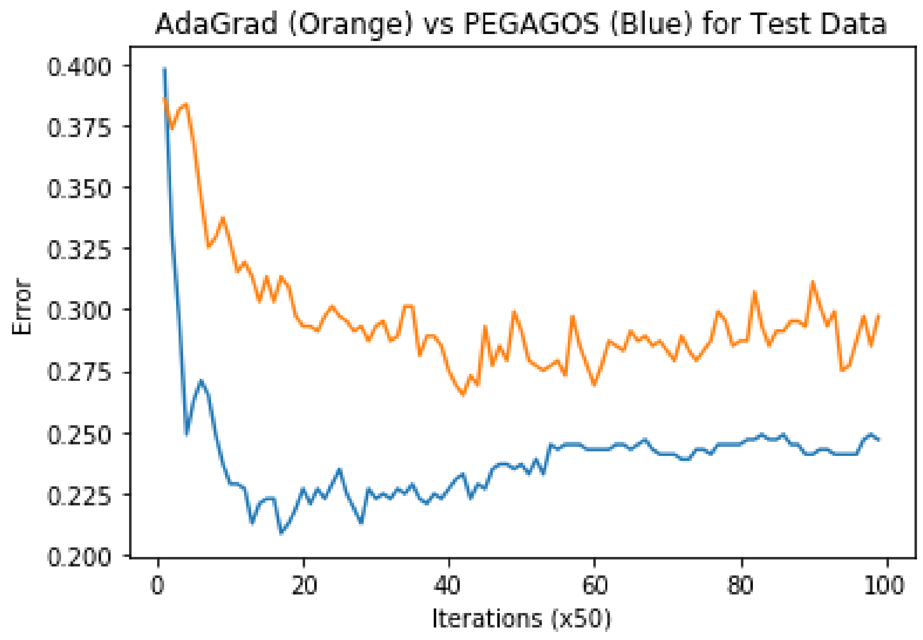
The graph for the classifier obtained with the AdaGrad algorithm is shown below.



The results confirm that the classifier works good and converges much faster than with the PEGASOS algorithm.

Error for entire test dataset: 0.28514

Together


AdaGrad (Orange) vs PEGAGOS (Blue) for Test Data
Error
Iterations (x50)