ESE 545: Data Mining **Fall 2018**

Project 2 Writeup

Name: *Joseph Lauigan*

1. **Problem 1**

   The RCV1 dataset is stored in the Scipy Compressed Sparse Row (CSR) format. The preprocessing and iteration over the data requires coordination of the data, indices, indptr components of the dataset.

   (a) To construct the label vector, list comprehension was used to pick out the indices that correspond to a CCAT topic (33) in the rcvq.target.indices vector. Once this was determined, logic was needed to figure out which articles these indices corresponded to - this was done by determining whether the index of a given CCAT label lied between the offset of a given article number (i) and the next article offset (j). If so, the article (i) was given a 1 label, otherwise it is given a -1 label. This is an iterative approach, requiring one pass through the rcv1.target.indptr vector.

   (b) Please see Jupyter notebook - Prob. 1b section

2. **Problem 2**

   The PEGASOS algorithm was implemented on the training data. For a given iteration, a random subset of article indices were chosen using the np.random library to be used as the min batch. For each article in this subset, the relevant row data and column indices were determined and then the PEGASOS algorithm update (if needed) was done to the relevant indices of the w vector for a given row.

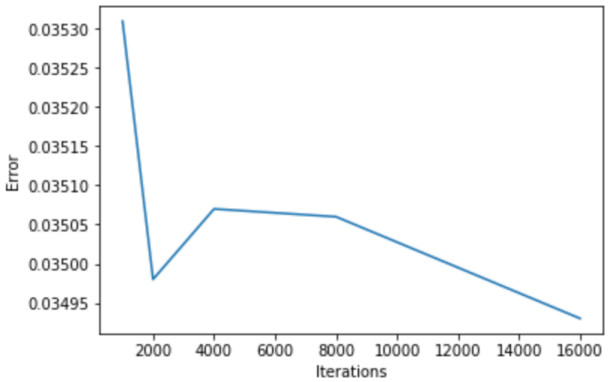
   The free parameters of the model are lambda value and the min batch size. To determine the optimal parameters, a sweep over different parameter configurations were done - where the training error was determined and compared. The iterations for this sweep was fixed to 1000 to allow for quick runtime, since we are more focused on relative performance between the parameter configurations. The results were as follows:

```
lambda:  1e-05 min_batch:  1e-05
lambda:  1e-05 min_batch:  0.0001
lambda:  1e-05 min_batch:  0.001
lambda:  1e-05 min_batch:  0.01
lambda:  1e-05 min_batch:  0.1
lambda:  0.0001 min_batch:  1e-05
lambda:  0.0001 min_batch:  0.0001
lambda:  0.0001 min_batch:  0.001
lambda:  0.0001 min_batch:  0.01
lambda:  0.0001 min_batch:  0.1
lambda:  0.001 min_batch:  1e-05
lambda:  0.001 min_batch:  0.0001
lambda:  0.001 min_batch:  0.001
lambda:  0.001 min_batch:  0.01
lambda:  0.001 min_batch:  0.1
lambda:  0.01 min_batch:  1e-05
lambda:  0.01 min_batch:  0.0001
lambda:  0.01 min_batch:  0.001
lambda:  0.01 min_batch:  0.01
lambda:  0.01 min_batch:  0.1
lambda:  0.1 min_batch:  1e-05
lambda:  0.1 min_batch:  0.0001
lambda:  0.1 min_batch:  0.001
lambda:  0.1 min_batch:  0.01
lambda:  0.1 min_batch:  0.1
```

```
error_matrix
#Lowest training error is with lambda  = 1e-5 and min_batch = 0.01
```

```
array([[0.26158, 0.09267, 0.04912, 0.03565, 0.03507],
       [0.17659, 0.07091, 0.0545 , 0.05307, 0.05293],
       [0.09437, 0.07948, 0.07872, 0.08   , 0.07947],
       [0.14134, 0.17044, 0.17883, 0.17907, 0.18072],
       [0.18916, 0.20904, 0.1991 , 0.19287, 0.19297]])
```

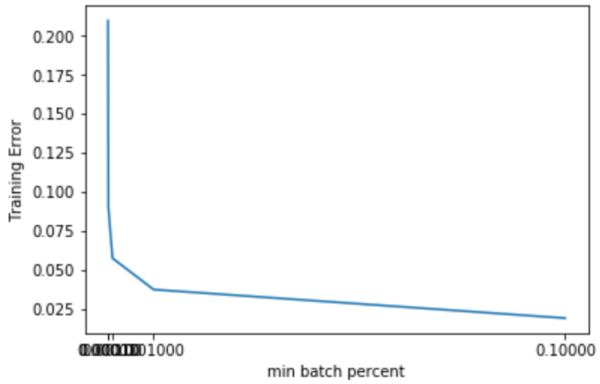
The lowest training error came from **lambda = 1e-5** and **min batch percentage of training data = 0.01 (1000 articles)**. Using these parameters, a sweep of different iterations were done, determining the training error for each of these. A plot of training error vs iteration is below:



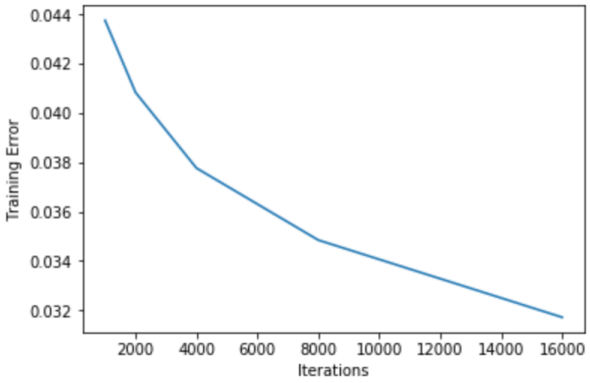
Through this plot, 16000 iterations gives the lowest training error - but it is only slightly better than 2000 iterations - so the runtime-error tradeoff for 2000 iterations is much better, and 2000 iterations was chosen for the test error use case.

3. **Problem 3**

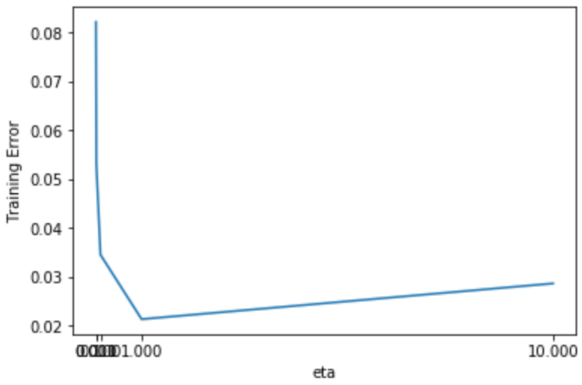
For Adagrad, the free parameter was the min batch size - as the eta is determined by the iterations - so a sweep of the the min batch size was done to compare the relative training errors. This was done with 4000 iterations:



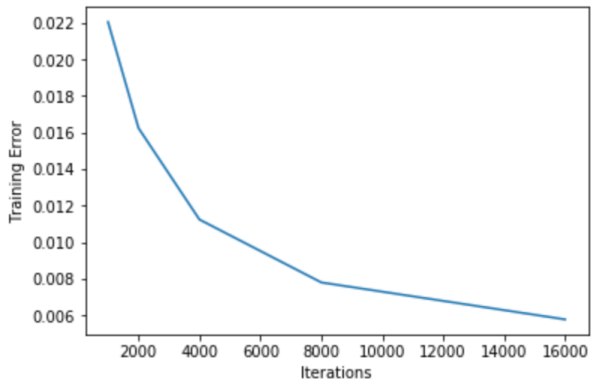
The minimum training error came with the min batch percentage of 0.1, but this is only marginally better than the min batch percentage of 0.01, so with runtime-error tradeoff considerations, the min batch percentage of 0.01 was used moving forward. The learning rate was set to be $\frac{1}{\sqrt{T}}$. The training error vs iterations plot is as follows:



As an additional analysis, the learning rate parameter was swept for different constants other than $\frac{1}{\sqrt{T}}$. The purpose of this is that the convergence of the algorithm would be much quicker for higher values of eta (e.g. 0.1, 1, 10, etc.). Using 1000 iterations, the following training error vs learning rate plot was constructed:

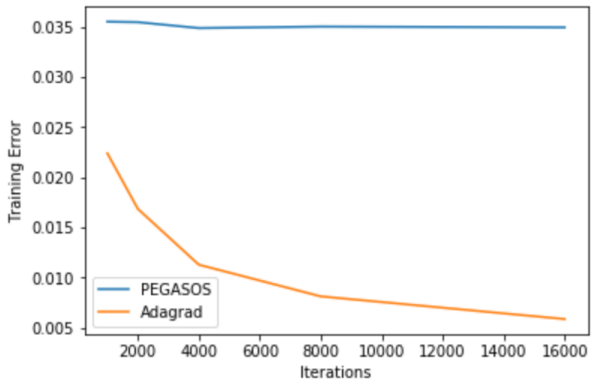


The learning rate that gave the lowest relative training error was 1 - so this was used in an analysis of training error vs iterations plot to see how the performance compared against setting the learning rate to $\frac{1}{\sqrt{T}}$. The plot is as follows:



Empirically, it looks like setting the learning rate to 1 has better performance than setting it to be $\frac{1}{\sqrt{T}}$. Therefore, the learning rate = 1 will be used for the test data.

The combined Adagrad and PEGASOS training error plot is as follows:

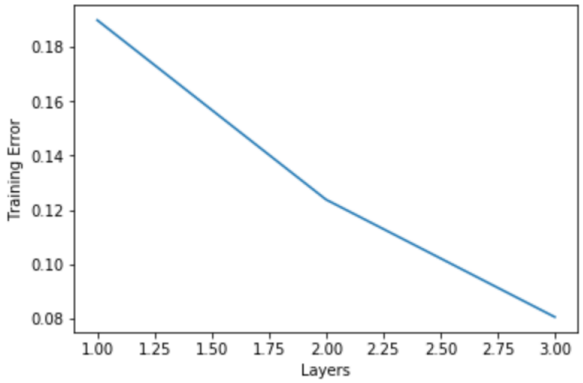


4. **Problem 4**

Neural networks were constructed using Keras - with each layer being a fully connected dense layer. Since the RCV1 data is in the scipy CSR format, there needs to be a conversion from the sparse representation of data to a dense representation to be able to be fed to the neural network for training and analysis. As the data is too big to have the whole dataset converted to a dense representation on my laptop, a sparse-to-dense generator is needed to do the conversion in batches and feed it to the neural network. The Keras layers and model classes allow for training and testing in batches using a generator, so a batch generator was constructed to feed the data in. The batch generator template was constructed by Chenlong Chen's comment in the following Kaggle discussion (https://www.kaggle.com/c/talkingdata-mobile-user-demographics/discussion/22567).

For the training of the neural network, the labels were changed from -1,1 to 0,1. Since this is a binary classification problem, the final layer is a logistic sigmoid function and the suitable loss function is the cross-entropy loss. Efficient optimization of this loss function over the network requires the labels to be 0,1 rather than -1,1. The batch size used for training is 100.

(a) Training error vs layers:



(b) Different configurations of the network architecture were analyzed - with units/layer, number of layers, loss function, and activation function combinations looked at. The results are summarized in the following table:

Training Error - Different Configurations								
MSE	**Binary Cross Entropy**							
3 Layers	*6 Layers*							
100 units	*100 units*				*200 units*			*300 units*
0.3385	*Sigmoid*	*ReLU*	*Tanh*	*ReLU/Tanh*	*ReLU*	*Tanh*	*ReLU/Tanh*	*ReLU*
	0.4697	0.0674	0.0578	0.0568	0.0644	0.0577	0.0581	0.0655

The combination that resulted in the lowest training error is the configuration with 6 layers, 100 units/layer and alternating ReLU/Tanh activation functions.

5. **Problem 5**

Using the parameters chosen in the training phase of the previous sections, the PEGASOS, Adagrad, and Neural Network algorithms were run on the test dataset and the test error is reported in the table below:

Test Error	
PEGASOS	0.5689
AdaGrad	0.5673
Neural Network	0.0645

As can be seen from the table above, the neural network does significantly better than both PEGASOS and Adagrad. I believe this is the case due to the sparseness of the data - in that the small training data set didn't allow for the PEGASOS algorithm to get a true sense of the underlying distribution of the data. Adagrad handles sparse data better, and it's training error is significantly lower than both the PEGASOS and Neural Network classifiers - but that would make sense since it was trained on the data that it was tested on for the training error. The expressiveness of the Adagrad classifier was limited in that it most likely overfit the training data and due to the sparseness of the data itself, wasn't able to generalize well on the test dataset. The neural network classifier was able to keep a rather consistent error on both the training and test datasets. This is due to the expressiveness and higher adaptability of the networks themselves, able to actually learn the features of the data better than a traditional SGD based method.