Gil Landau
11/5/18
ESE 545

<u>Project 2 Report</u>

Part 1:

You can find the label vector in the part of the code labelled “Question 1.” I set the labels to 1 if it has been classified as CCAT and to -1 if it has not been classified as CCAT. I split the labels and the features into two groups: train_vector_labels and train_vector_features are the first 100,000 articles and their respective features and CCAT labels. test_vector_labels and test_vector_features are the last 700,000 or so articles and their respective features and CCAT labels.

Part 2:

You can find my implementation of PEGASOS in the part of the code labelled “Question 2.” PEGASOS has three input variables (not including the training vectors): λ (a regularization parameter), T (the number of iterations), and K (the size of a mini-batch). Below you can see graphs of the PEGASOS algorithm over 10,000 iterations. All information comes from the original paper here: http://ttic.uchicago.edu/~nati/Publications/PegasosMPB.pdf

λ (regularization parameter):

A regularization parameter is needed to scale the weights by reasonable amount, to allow soft-margin in the SVM, and to prevent overfitting. Essentially, a higher regularization parameter decreases the variance of the SVM weights but increases the bias. If λ is set too high, it can completely overwhelm the actual calculations being done. If it is set too low, it can result in overfitting.

Here, if the parameter was too low it overfit and if it was too high, it performed badly. .001 was a good middle ground

T (the number of iterations):

More than one iteration is usually required because PEGASOS is based on stochastic gradient descent to find the minimum of the loss function. A key attribute of stochastic gradient descent is iteratively updating the weights based upon the change in the loss function (the "gradient"). As the gradient becomes smaller, the algorithm comes closer and closer to finding a minimum to the loss function (represented as a convex function). *Stochastic* gradient descent tries to avoid some pitfalls of standard gradient descent, such as performance costs and local minimums. The *stochastic* aspect means that the current iteration compares the its gradient with that of a random sample (can be one) in the set.

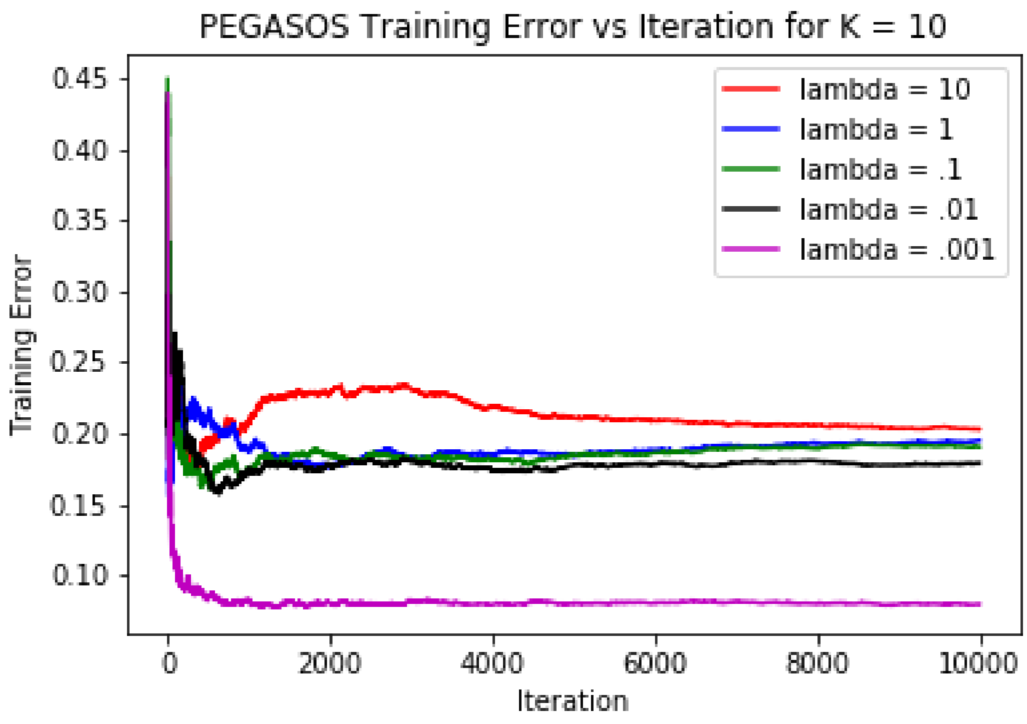
The number of iterations and the regularization parameter also influence the learning rate of the PEGASOS. As the algorithm goes on, the learning rate gets smaller and smaller. This is so gradient descent avoids overshooting the minimum.

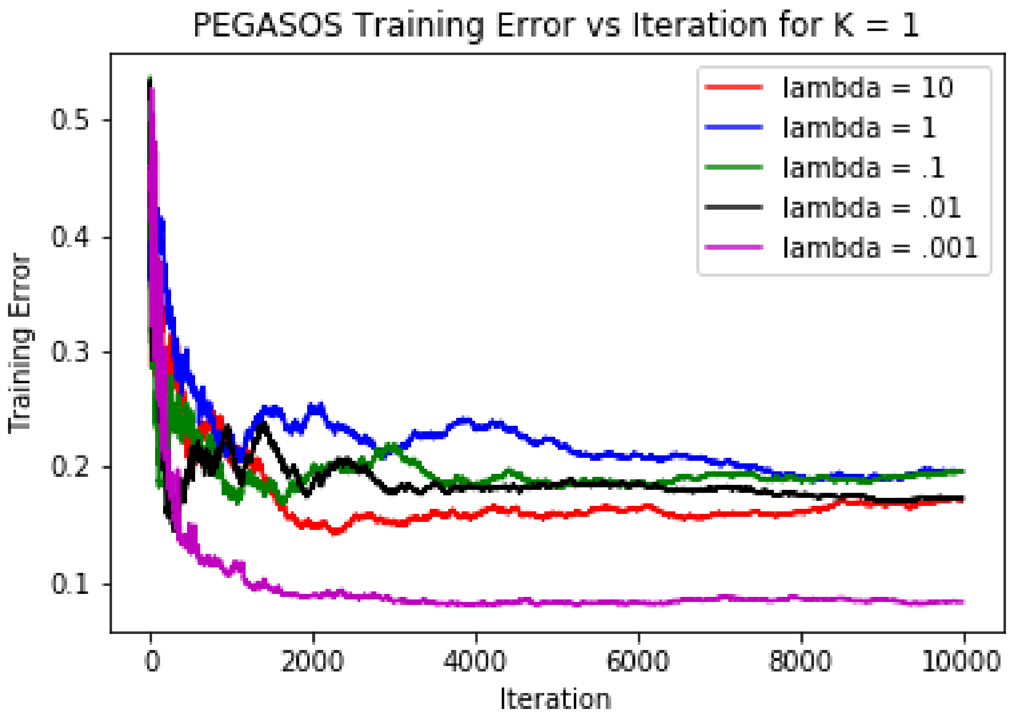
If T was too high, it took too much time to run. After a certain point, the model converges at won't perform much better. 10,000 iterations were plenty of time for it to converge.

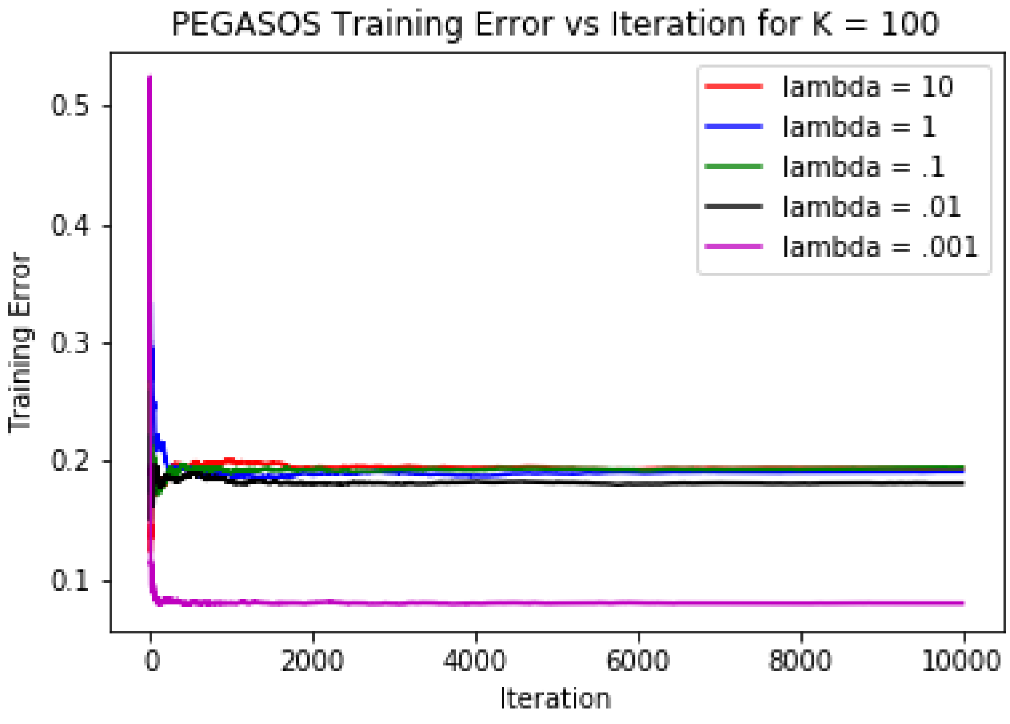
K (the size of a mini-batch):

The K is not necessarily a feature of PEGASOS, but rather a feature of a mini-batch system (PEGASOS exists in "true stochastic" K=1 and mini-batch variants). Mini-batches can be used to optimize performance at the slight expense of accuracy. When K is equal to the size of the set, each iteration is solving the original objective function. Mini-batches act to solve an approximate objective function, by selecting a random subset of the inputs at each iteration of PEGASOS for which the weight vector w yielded a non-zero loss. PEGASOS then uses those inputs to calculate the sub-gradient at that iteration.

K = 100 was a good middle ground and performed best in my tests.




PEGASOS Training Error vs Iteration for K = 1
Training Error
Iteration
0.1
0.2
0.3
0.4
0.5
0
2000
4000
6000
8000
10000
lambda = 10
lambda = 1
lambda = .1
lambda = .01
lambda = .001


PEGASOS Training Error vs Iteration for K = 100
Training Error
Iteration
0.1
0.2
0.3
0.4
0.5
0
2000
4000
6000
8000
10000
lambda = 10
lambda = 1
lambda = .1
lambda = .01
lambda = .001

<u>Part 3:</u>

Note: I know I was supposed to graph the training error vs the number of iterations in the same plot as above, but I decided to separate them out because I wanted to show the effect of λ and K on the results. The charts were already very crowded.

You can find my implementation of ADAGRAD in the part of the code labelled "Question 3." ADAGRAD has four input variables (not including the training vectors): λ (a regularization parameter), T (the number of iterations), K (the size of a mini-batch), and e (an incredibly small number). Below you can see graphs of the ADAGRAD algorithm over 5,000 iterations. All information comes from the original paper here:

http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

λ (regularization parameter):

The regularization parameter plays a similar role in ADAGRAD as it did in PEGASOS, particularly in setting the learning rate.

Here, if the parameter was too low it overfit and if it was too high, it performed badly. .001 was a good middle ground

T (the number of iterations):

Again, like PEGASOS, ADAGRAD, as a stochastic gradient descent-based model, needs to run over several iterations to be effective.

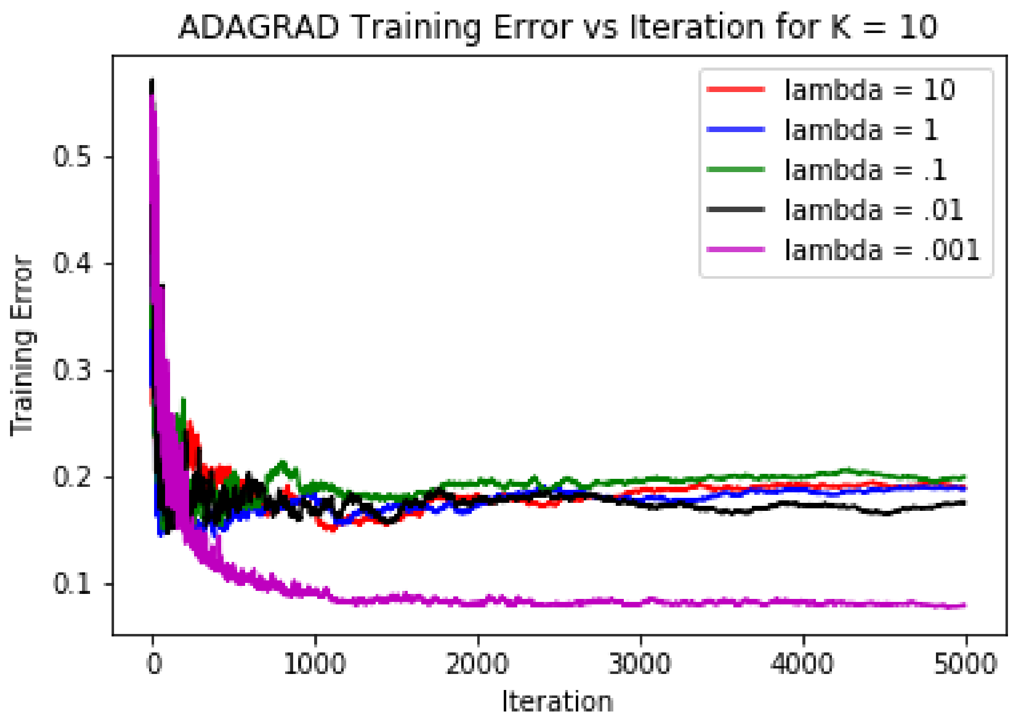
If T was too high, it took too much time to run. After a certain point, the model converges at won't perform much better. 5,000 iterations were plenty of time for it to converge.

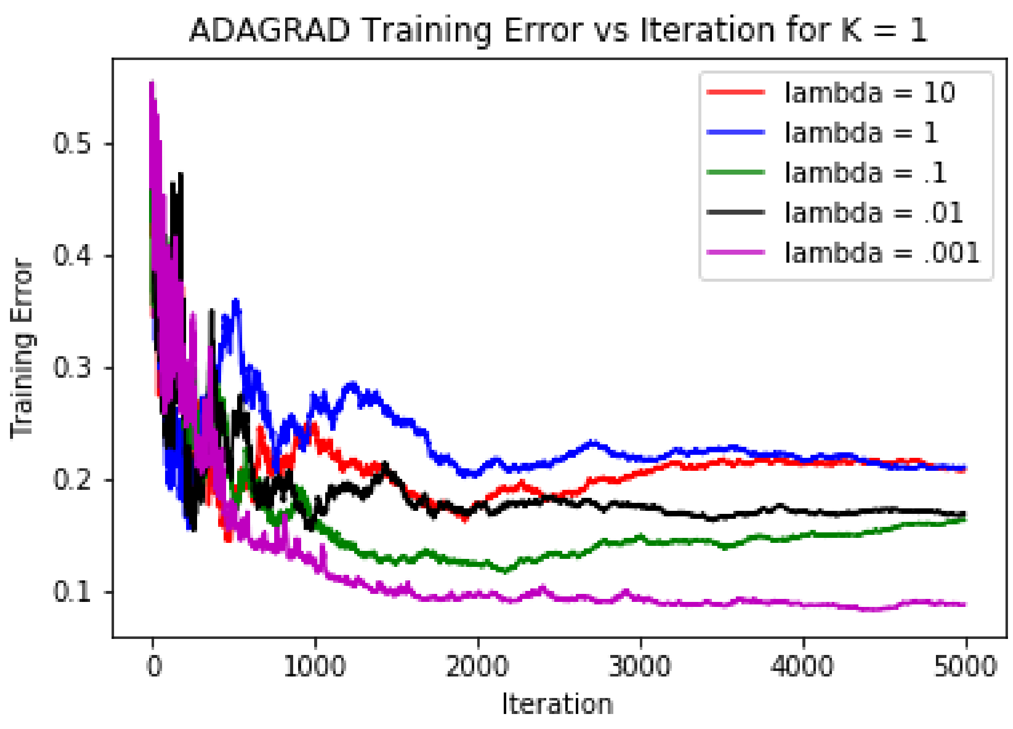
K (the size of a mini-batch):

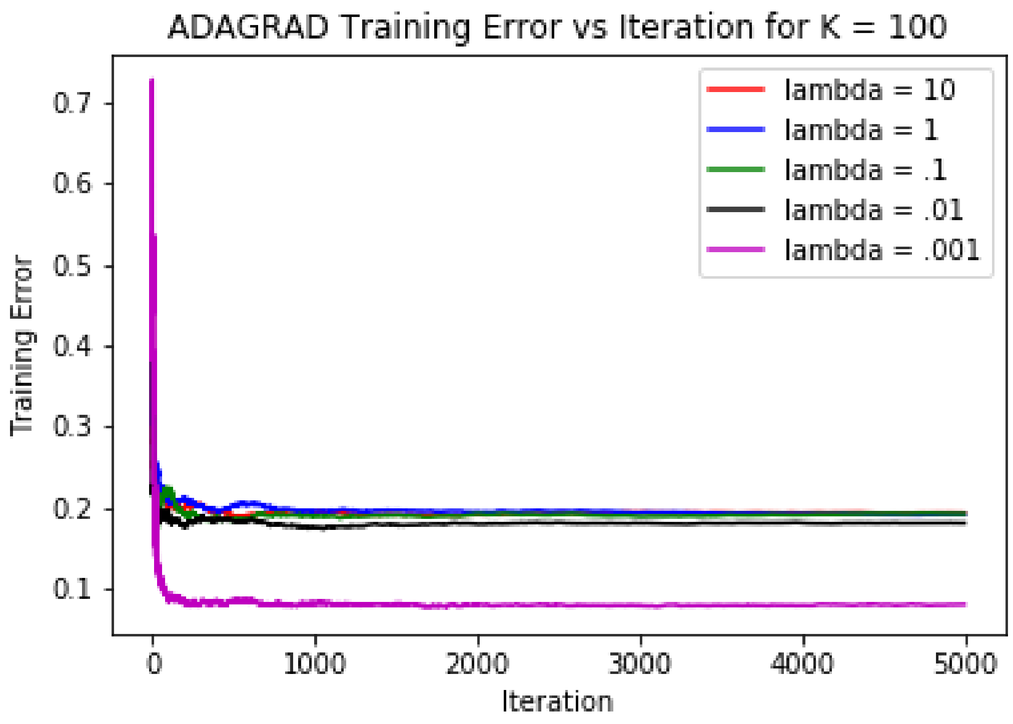
Like PEGASOS, ADAGRAD can use a mini-batch system as well as a “true stochastic” system. K = 10 surprisingly performed best in my tests.

e (an incredibly small number):

Unlike PEGASOS, ADAGRAD gives each feature its own gradient. This theoretically allows for more adaptive algorithm and allows the weights of features to be more fine-tuned. However, practically speaking, this means scaling the learning rate by the previous gradients (in some form or another) and at the start of the algorithm, those values are initialized to 0. Therefore, you need some small value to avoid dividing by 0. It can really be anything, as long as it is small. I set *e* to .00000001.



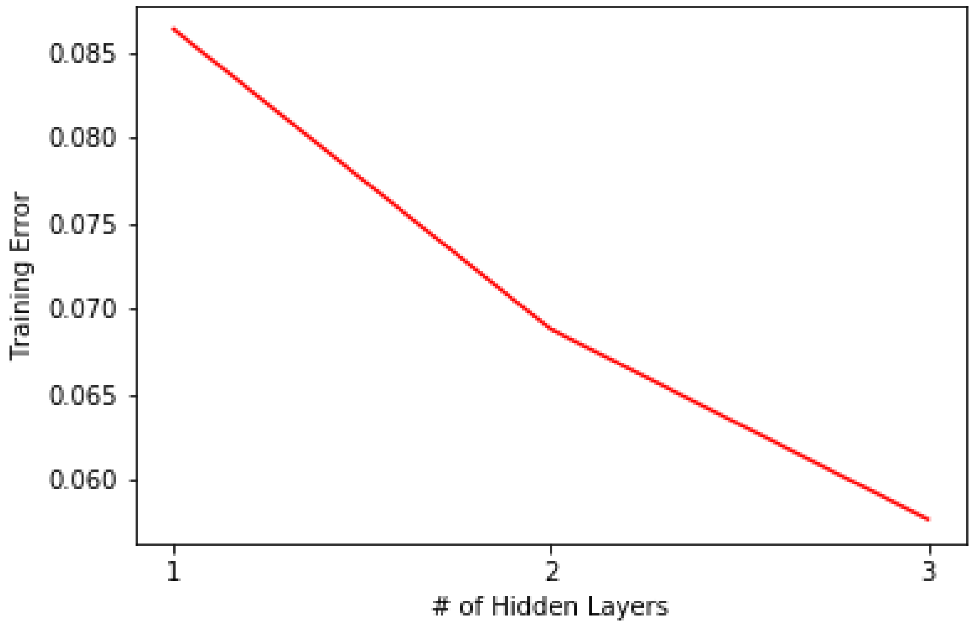

ADAGRAD Training Error vs Iteration for K = 1
Training Error
Iteration
0.1
0.2
0.3
0.4
0.5
0
1000
2000
3000
4000
5000
lambda = 10
lambda = 1
lambda = .1
lambda = .01
lambda = .001

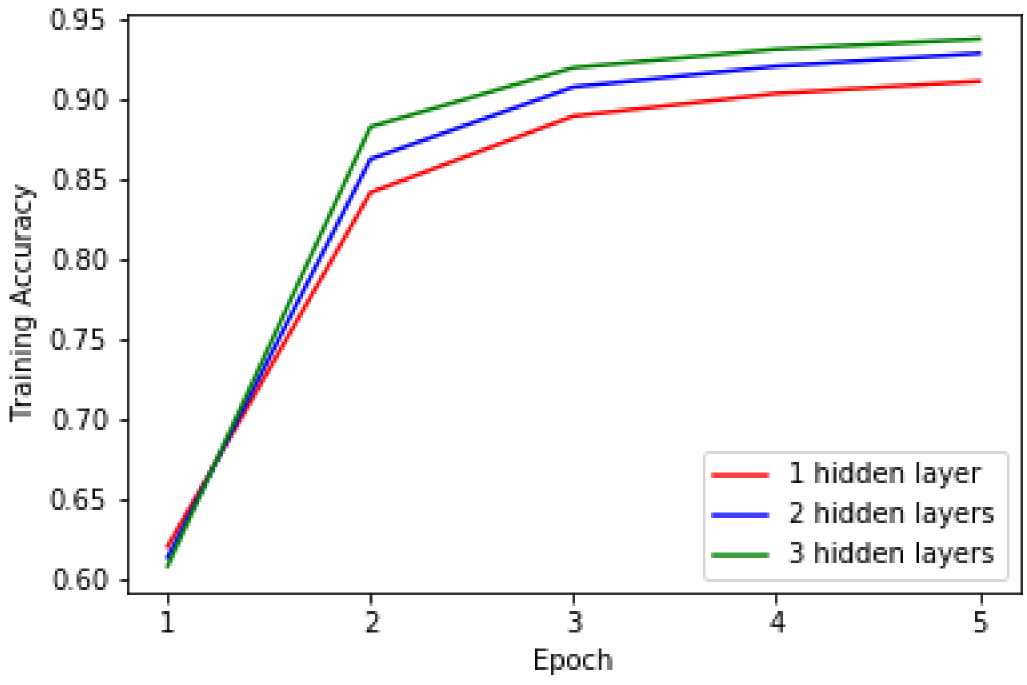

ADAGRAD Training Error vs Iteration for K = 100
Training Error
Iteration
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
1000
2000
3000
4000
5000
lambda = 10
lambda = 1
lambda = .1
lambda = .01
lambda = .001

Part 4A:

You can find my code in the section labelled "Question 4." Below you can see my charts for both the training error over the number of layers and the training error over the number of epochs. As far as parameters are concerned, I chose a basic sequential model (since we want to create a simple linear model with several simple hidden layers).

I chose to use ReLUs as my activation function, because they can zero out irrelevant features. It has a function max(0,x) for an input x. It has the benefits of being efficient, has sparse activation (which works especially well on a sparse matrix) and is clearly one-sided (as opposed to something like a sigmoid). My loss function is the bog-standard mean squared error. I tried to use hinge (similar to an SVM's margin function) which gave me disastrous results (60% accuracy). I believe that might be because it ends up removing many of the key features in conjunction with the RELU. I chose to use a batch size of 100 as a middle ground to avoid long run times.





Part 4B:

You can see graphs below based on the number of units and the number of layers I stuck with ReLUs and MSE for reasons mentioned above. Increasing the number of units per layer drastically increases the epoch compute time, so I tried only three different numbers of units: 50, 100, 200. Regarding the number of layers, I found that there were diminishing returns on the number of layers I added. In general, I found that both the size of the batches and the number of epochs increased accuracy but took longer to evaluate. It was a matter of balancing those two variables as well. As a result, I focused on being more efficient than accurate.

NN Model	**Training Error**
1 hidden layer, 50 units, 100 batch	91.436%
1 hidden layer, 200 units, 100 batch	91.45%
2 hidden layers, 50 units, 100 batch	92.995%
2 hidden layers, 200 units, 100 batch	91.543%
6 hidden layers, 50 units, 100 batch	94.217%
6 hidden layers, 100 units, 100 batch	94.724%
5 hidden layers, 50 units, 50 batch	94.685%

Part 5:

Model	**Training Accuracy**	**Testing Accuracy**
PEGASOS(λ = .001, T= 10000, K = 100)	**92.02%**	**91.99%**
PEGASOS(λ = .0000001, T= 10000, K = 100)	97.565%	92.83%
ADAGRAD(λ = .001, T= 5000, K = 10)	**91.67%**	**91.62%**
NN-6 hidden layers, 100 units, 100 batch	**94.724%**	**93.49%**

My neural network model performed best, followed by my PEGASOS and then my ADAGRAD. I think the neural network did best because it managed to zero out some of the unnecessary features and build up new features as the number of layers increased. There was a clear pattern of more accurate results as more layers were added, so simply generating more useful features was a vital advantage.