Ali Kozlu

Report

Format of the Report:

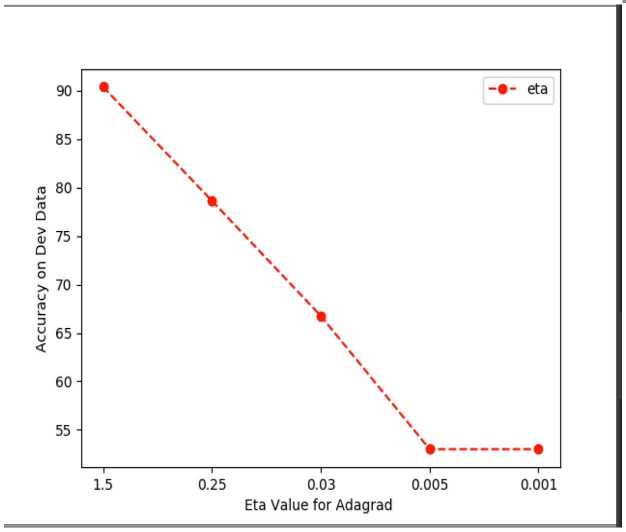
1. Parameter Tuning

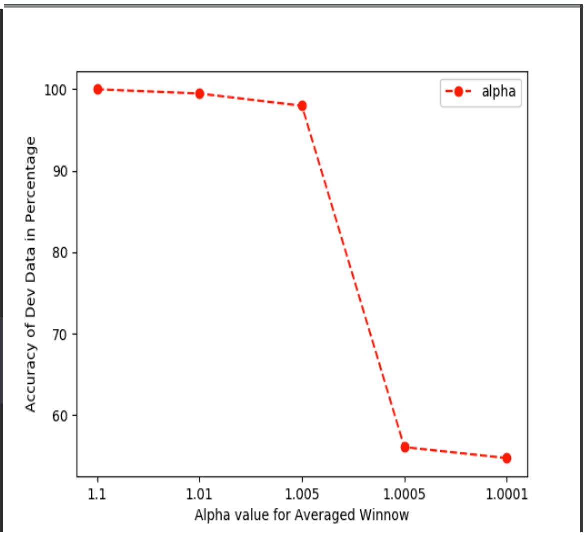
2.Learning Curve Plots and Observation of Results

 3. SVM and Average Perceptron Accuracy Results on four different data sets.

Parameter Tuning for Adagrad and Winnow:

Before testing Adagrad and Winnow on synthetic data, I plotted various alpha and learning rate parameters for normal versions of these algorithms. Below are the results. Afterwards the most accurate parameters on the development set were given to Classifier initialization as default values.

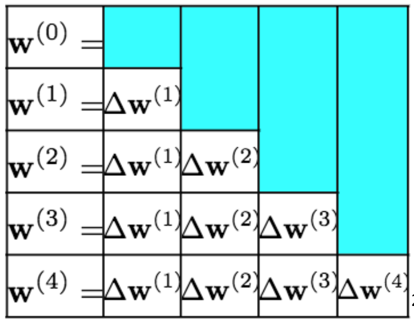




I implemented average versions of additive algorithms using Daume's method after encountering his 2006 PHD Thesis[1]. Daume's method works because when we write each w_i and c_i as sum of changes made to our original weight vector (explained in graph below), we get a telescoping series where each consistency term cancels out each other, except the last term, which accounts to u/c. I compared time complexity of Daume's method, where we only have to iterate over active features to a simpler cumulative method it was necessary to iterate through all features in order to keep the running sum. I was able to test that Daume's method allowed significant gains in sparse data set, where the vector dimension was 200, and in real data sets. Daume's method does not apply to multiplicative algorithms such as Winnow, so I implemented Averaged Winnow by keeping a cumulative weight vector self.u (including bias) and updating its contents with a weighted copy of current weight vector whenever a mistake was made. I also experimented with an implementation where we average the different weighted vectors over the number of examples we have seen. This implementation was fundamentally similar to keeping a cumulative weight vector. After implementing all of the algorithms and testing that they were working properly, I compared their development accuracy on synthetic data. After ten

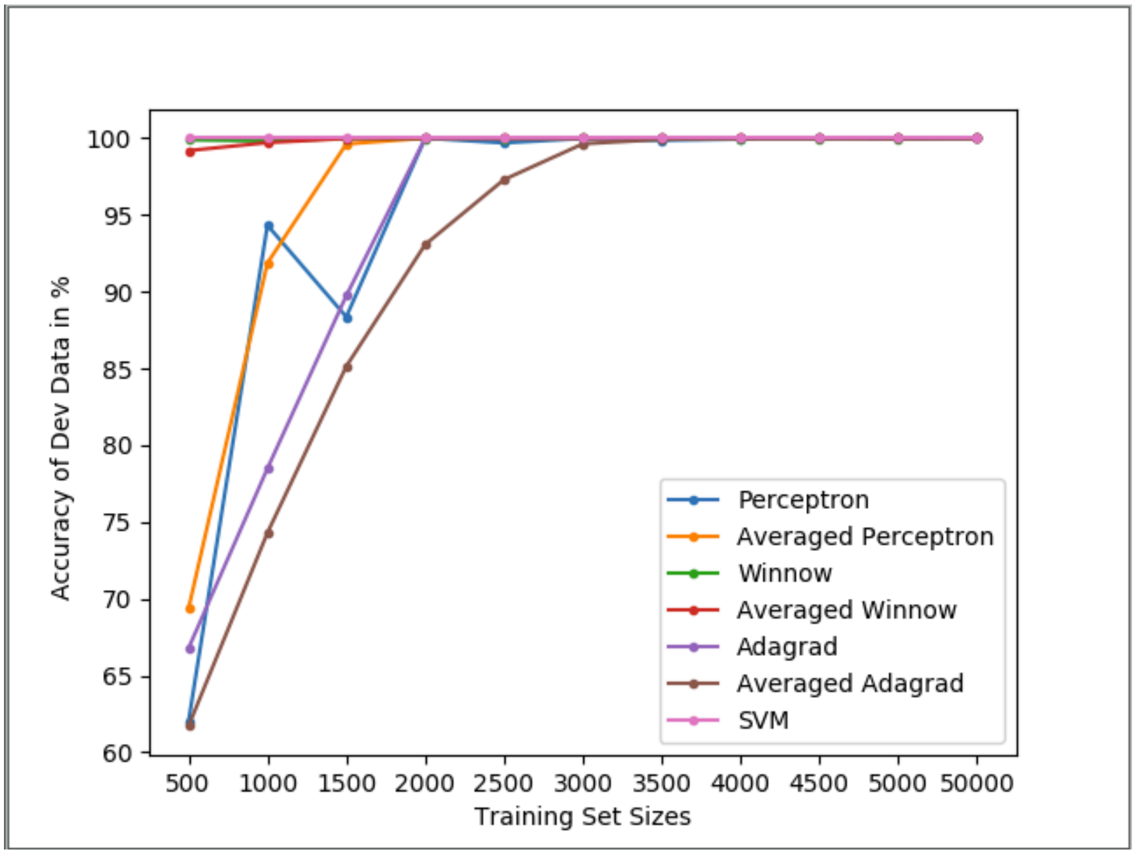
[1] Daume 2006 PHD Thesis: http://www.umiacs.umd.edu/~hal/docs/daume06thesis.pdf

iterations, all seven algorithms performed %100 on dense and sparse synthetic test data. Averaged Perceptron performed better than Perceptron after a single iteration over 50000 examples. However, averaged Adagrad had to "catch up" with its normal version. I explain my reasoning for this behavior when I talk about learning curve plots.


$\mathbf{w}^{(0)} =$
$\mathbf{w}^{(1)} = \Delta\mathbf{w}^{(1)}$
$\mathbf{w}^{(2)} = \Delta\mathbf{w}^{(1)} \Delta\mathbf{w}^{(2)}$
$\mathbf{w}^{(3)} = \Delta\mathbf{w}^{(1)} \Delta\mathbf{w}^{(2)} \Delta\mathbf{w}^{(3)}$
$\mathbf{w}^{(4)} = \Delta\mathbf{w}^{(1)} \Delta\mathbf{w}^{(2)} \Delta\mathbf{w}^{(3)} \Delta\mathbf{w}^{(4)}$
[2]

Learning Curve Plots:

Below is the learning curve plot for 11 different ***dense*** synthetic test sizes.

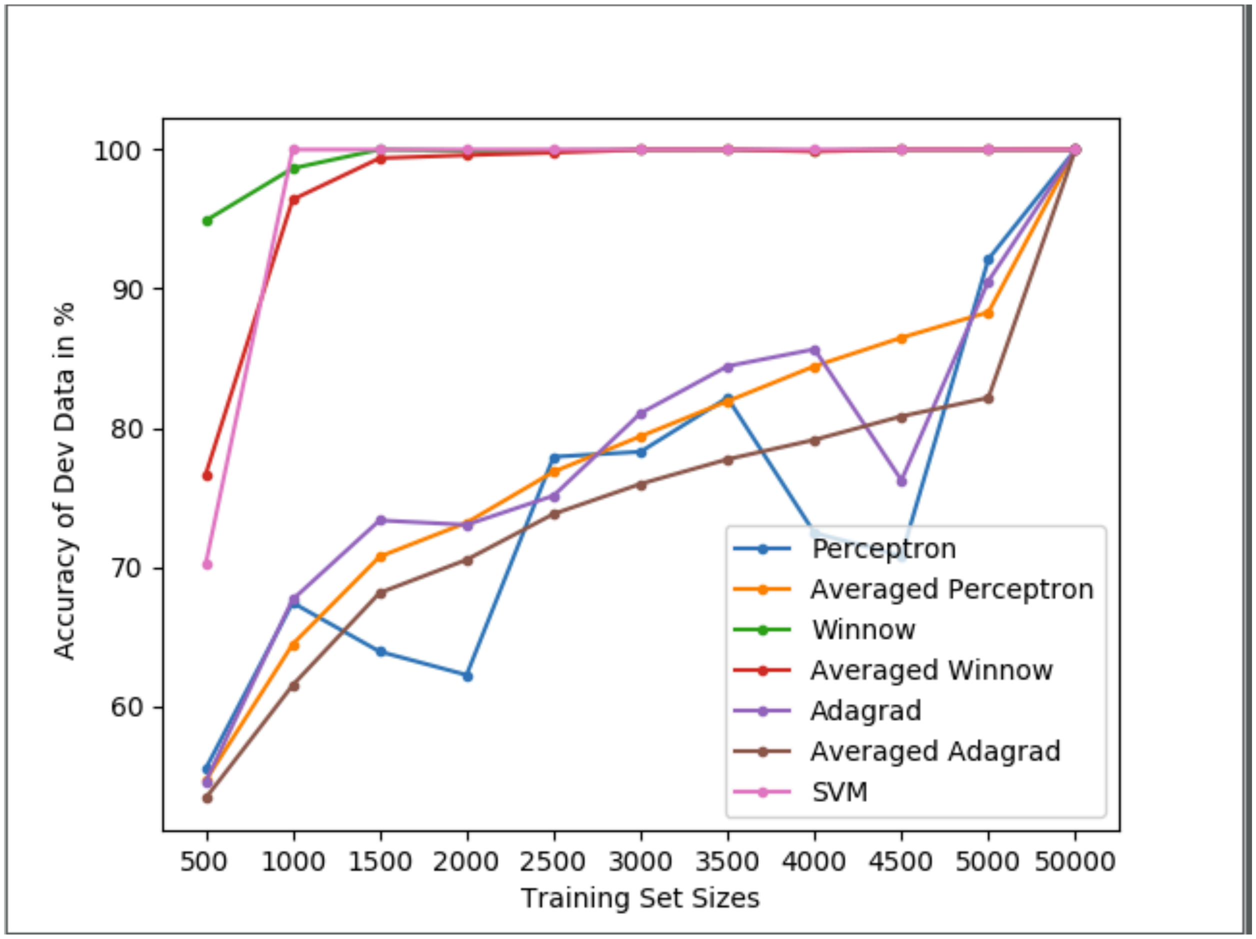


We see that Winnow, Averaged Winnow and SVM converge immediately after seeing 500 examples. This actually implies that Winnow makes the fewest mistakes when learning a k-disjunction over N

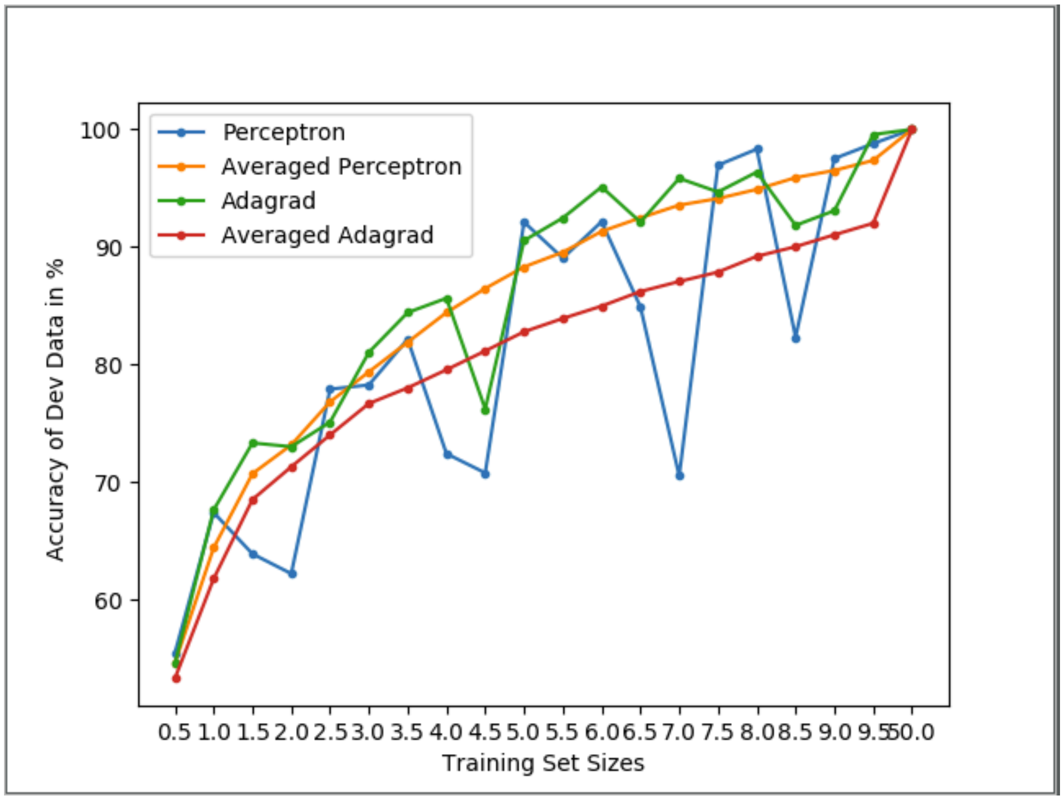
[2] Image taken from slides of Professor Liang Huang from Oregon State University. http://classes.engr.oregonstate.edu/eecs/fall2017/cs534/slides/1-perc.pdf

variables when compared to Perceptron and Adagrad. However, this conclusion is not easy to make in a dense data set as Perceptron, Adagrad and their average versions also converge after seeing at most 2500 examples. The reason that difference between mistake bounds of Winnow and Perceptron is not clear has to do our domain characteristics of dense data. Additive algorithms have the advantage (in terms of their mistake bound) when depending on "less features", i.e. when target vector is dense and the function space is sparse. Also I noticed that averaged versions of algorithms were consistent with more examples, something that is clearer in learning curve below.

Below is the learning curve plot for 11 different ***sparse*** synthetic test sizes.

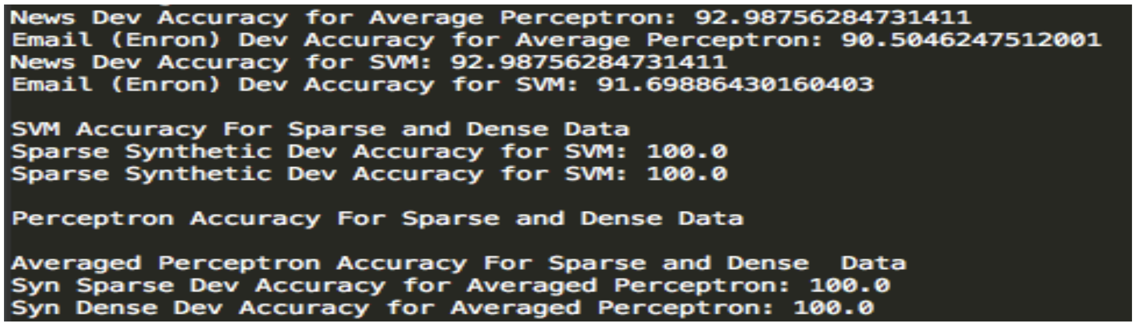


The difference between a multiplicative algorithm and additive algorithm is more clear when we are testing in sparse data. The number of examples needed to get 100% accuracy is between 500-1000 for Winnow, whereas Adagrad and Perceptron can reach only %90 after seeing 5000 examples. When a sparse target vector is depending on only small number of relevant features (k << N), winnow converges with much less examples, thus giving a better mistake bound. This shows that assuming N features with k relevant items and k<<N winnow makes logarithmic number of mistakes in terms of k and N, whereas additive algorithms make linear number of mistakes in terms of k and N. Simply put, Winnow is more robust to high-dimensional feature spaces. This result supports the conclusions Professor Roth reached in lecture. Now let us look another plot where I only show accuracies of Adagrad and Perceptron and averaged versions. I apologize for the way x-axis ticks were printed, they represent test sizes [500, 1000, 1500…., 9500, 10000, 50000]



We see that Perceptron and Adagrad can have some different accuracy when the training example difference is minuscule. For example, Perceptron with 6000 examples performs over %90, whereas perceptron with 7000 examples perform around %70. This result is consistent with the idea that we do not know when we will make the mistakes. Our Perceptron model made a mistake between 6000-7000 examples when a good performing model was thrown out. If we had only 7000 examples, Perceptron would not have given a global guarantee on performance. On the other hand average versions of Perceptron and Adagrad show consistency with the number of examples, yielding a global guarantee on unseen data through regularization. Also the way Adagrad outperforms Perceptron after having similar accuracies up to a certain test size might support the fact that it is supposed to update weights faster than Perceptron.

Accuracies of SVM and Average Perceptron (after a single iteration) on 4 datasets: Below are printed accuracies to Python console. You can run hw2.py to get the same results. Again, for real data sets, since the target vector is dense, Average Perceptron performs closer to SVM.



```
News Dev Accuracy for Average Perceptron: 92.98756284731411
Email (Enron) Dev Accuracy for Average Perceptron: 90.5046247512001
News Dev Accuracy for SVM: 92.98756284731411
Email (Enron) Dev Accuracy for SVM: 91.69886430160403

SVM Accuracy For Sparse and Dense Data
Sparse Synthetic Dev Accuracy for SVM: 100.0
Sparse Synthetic Dev Accuracy for SVM: 100.0

Perceptron Accuracy For Sparse and Dense Data

Averaged Perceptron Accuracy For Sparse and Dense  Data
Syn Sparse Dev Accuracy for Averaged Perceptron: 100.0
Syn Dense Dev Accuracy for Averaged Perceptron: 100.0
```

Conclusion:

The main lessons from this project were how different algorithms (additive vs multiplicative) performed when sparseness target function and function space changed. Also I was able to see that averaged versions of these algorithms were able to guarantee a globally optimal solution on unseen data. Also, extracting meaningful features from text data showed me another way of how to derive Boolean features from interesting feature types.