CS446: Machine Learning	**Spring 2017**
Problem Set 3	
Handed Out: February 15^{th}, 2017	*Due: February* 27^{th}, 2017

- Feel free to talk to other members of the class in doing the homework. I am more concerned that you learn how to solve the problem than that you demonstrate that you solved it entirely on your own. You should, however, write down your solution yourself. Please try to keep the solution brief and clear.
- Please use Piazza first if you have questions about the homework. Also feel free to send us e-mails and come to office hours.
- Please, no handwritten solutions. You will submit your solution report as a single pdf file. Follow the instructions in the "What to submit" section for more details.
- A large portion of this assignment deals with programming the online learning algorithms as well as running experiments to see them in action. While we do provide some pieces of code, you are required to try and test several online learning algorithms by writing your own code. While we encourage discussion within and outside of the class, cheating and code copying is strictly prohibited. Copied code will result in the entire assignment being discarded from grading at the very least.
- The homework is due at 11:59 PM on the due date. We will be using Compass2g for collecting the homework assignments. Please submit an electronic copy via Compass (`http://compass2g.illinois.edu`). Please do NOT hand in a hard copy of your write-up. Contact the TAs if you face technical difficulties in submitting the assignment.

Online Algorithm Comparison - 100 points

In this problem set, you will implement five online learning algorithms – *Perceptron* (with and without margin), *Winnow* (with and without margin), and *AdaGrad*; and experiment with them by comparing their performance on synthetic datasets. We provide Python code that generates the synthetic dataset, but you *do not need to* code your algorithms in Python.

In this problem set, you will get to see the impact of parameters on the performance of learning algorithms and, more importantly, the difference in the behavior of learning algorithms when the target function is sparse and dense. You will also assess the effect of noisy training data on the learning algorithms.

First, you will generate examples that are labeled according to a simple l-of-m-of-n boolean function. That is, the function is defined on instance space $\{0,1\}^n$, and there is a set of m attributes such that an example is positive iff at least l of these m are active in the example. l, m and n define the hidden concept that your algorithms will attempt to learn. The instance space is $\{0,1\}^n$ (that is, there are n boolean features in the domain). You will run experiments on several values of l, m, and n.

To make sure you understand the above function, try to write it as a linear threshold function. This way, you will make sure that Winnow and Perceptron can represent this function (recall these algorithms learn linear separators). Also, notice that the l-of-m-of-n concept class is a generalization of monotone conjunctions (when $l = m$) and of monotone disjunctions (when $l = 1$).

Your algorithm does not know the target function and does not know which are the relevant attributes or how many are relevant. The goal is to evaluate these algorithms under

several conditions and derive some conclusions on their relative advantages and disadvantages.

Algorithms

You are going to implement five online learning algorithms. Notice that they are essentially the same algorithm, only that their update rules are different.

We will prescribe the weight initialization and options for the parameter(s) for each algorithm, and you should determine the best parameters for each algorithm via tuning on a subset of training data. More details on this in the **Parameter Tuning** section.

In all the algorithms described below, labeled examples are denoted by (x, y) where $x \in \{0, 1\}^n$ and $y \in \{-1, 1\}$. In all cases, your prediction is given by

$$y = sign(w^\intercal x + \theta)$$

1. **Perceptron**: The simplest version of the Perceptron Algorithm. In this version, an update will be performed on the example (x, y) if $y(w^\intercal x + \theta) \leq 0$.

   There are two things about the Perceptron algorithm that should be noted.

   First, the Perceptron algorithm needs to learn both the bias term θ and the weight vector w. When the Perceptron algorithm makes a mistake on the example (x, y), both w and θ will be updated in the following way:

   $$\begin{aligned} w_{\text{new}} &\leftarrow w + \eta y x \\ \text{and} \quad \theta_{\text{new}} &\leftarrow \theta + \eta y \end{aligned}$$

   where η is the learning rate. See the lecture notes for more information.

   Second (and more suprisingly), if we assume that the order of the examples presented to the algorithm is fixed, and we initialize $\begin{bmatrix} w & \theta \end{bmatrix}$ with a zero vector and learn w and θ together, then the learning rate η, in fact, does not have any effect[1].

   Initialization: $w^\intercal = \{0, 0, \cdots, 0\}, \theta = 0$

   Parameters: Given the second fact above, we can fix $\eta = 1$. So there are no parameters to tune.

2. **Perceptron with margin**: This algorithm is similar to Perceptron; but in this algorithm, an update will be performed on the example (x, y) if $y(w^\intercal x + \theta) < \gamma$, where γ is an additional **positive** parameter specified by the user. Note that this algorithm sometimes updates the weight vector even when the weight vector does not make a mistake on the current example.

   Parameters: learning rate η (to tune), fixed margin parameter γ=1.

[1] In fact you can show that, if w_1 and θ_1 is the output of the Perceptron algorithm with learning rate η_1, then w_1/η_1 and θ_1/η_1 will be the result of the Perceptron with learning rate 1 (note that these two hyperplanes give identical predictions).

Given that $\gamma > 0$, using a different learning rate η will produce a different weight vector. The best value of γ and the best value of η are closely related given that you can scale γ and η.

Initialization: $w^\intercal = \{0, 0, \cdots, 0\}, \theta = 0$

Parameter Recommendation: Choose $\eta \in \{1.5, 0.25, 0.03, 0.005, 0.001\}$.

3. **Winnow**: The simplest version of Winnow. Notice that all the target functions we deal with are monotone functions, so we are simplifying here and using the simplest version of Winnow.

   When the Winnow algorithm makes a mistake on the example (x, y), w will be updated in the following way:

   $$w_{t+1,i} \leftarrow w_{t,i}\alpha^{yx_i}$$

   where α is promotion/demotion parameter and $w_{t,i}$ is the ith component of the weight vector after t mistakes.

   Parameters: Promotion/demotion parameter α

   Initialization: $w^\intercal = \{1, 1, \cdots, 1\}, \theta = -n$ (θ is fixed here, we **do not** update it)

   Parameter Recommendation: Choose $\alpha \in \{1.1, 1.01, 1.005, 1.0005, 1.0001\}$.

4. **Winnow with margin**: This algorithm is similar to Winnow; but in this algorithm, an update will be performed on the example (x, y) if $y(w^\intercal x + \theta) < \gamma$, where γ is an additional **positive** parameter specified by the user. Note that, just like perceptron with margin, this algorithm sometimes updates the weight vector even when the weight vector does not make a mistake on the current example.

   Parameters: Promotion/demotion parameter α, margin parameter γ.

   Initialization: $w^\intercal = \{1, 1, \cdots, 1\}, \theta = -n$ (θ is fixed here, we **do not** update it)

   Parameter Recommendation: Choose $\alpha \in \{1.1, 1.01, 1.005, 1.0005, 1.0001\}$ and $\gamma \in \{2.0, 0.3, 0.04, 0.006, 0.001\}$.

5. **AdaGrad**: AdaGrad adapts the learning rate based on historical information, so that frequently changing features get smaller learning rates and stable features get higher ones. Note that here we have different learning rates for different features. We will use the hinge loss:

   $$Q((x, y), w) = \max(0, 1 - y(w^\intercal x + \theta)).$$

   Since we update both w and θ, we use g_t to denote the gradient vector of Q on the $(n + 1)$ dimensional vector (w, θ) at iteration t.

   The per-feature notation at iteration t is: $g_{t,j}$ denotes the jth component of g_t (with respect to w) for $j = 1, \cdots, n$ and $g_{t,n+1}$ denotes the gradient with respect to θ.

   In order to write down the update rule we first take the gradient of Q with respect to the weight vector (w_t, θ_t),

$$g_t = \begin{cases} 0 & \text{if } y(w_t^\intercal x + \theta) > 1 \\ -y(x, 1) & otherwise \end{cases}$$

That is, for the first n features, that gradient is $-yx$, and for θ, it is always $-y$.

Then, for each feature j ($j = 1, ..., n+1$) we keep the sum of the gradients' squares:

$$G_{t,j} = \sum_{k=1}^{t} g_{k,j}^2$$

and the update rule is

$$w_{t+1,j} \leftarrow w_{t,j} - \eta g_{t,j} / (G_{t,j})^{1/2}$$

By substituting g_t into the update rule above, we get the final update rule:

$$w_{t+1,j} = \begin{cases} w_{t,j} & \text{if } y(w_t^\intercal x + \theta) > 1 \\ w_{t,j} + \eta y x_j / (G_{t,j})^{\frac{1}{2}} & otherwise \end{cases}$$

where for all t we have $x_{n+1} = 1$.

We can see that AdaGrad with hinge loss updates the weight vector only when $y(w^\intercal x + \theta) \leq 1$. The learning rate, though, is changing over time, since $G_{t,j}$ is time-varying.

You may wonder why there is no AdaGrad with Margin: note that AdaGrad updates w and θ only when $y(w^\intercal x + \theta) \leq 1$ which is already a version of the Perceptron with Margin 1.

Parameters: η

Initialization: $w^\intercal = \{0, 0, \cdots, 0\}, \theta = 0$

Parameter Recommendation: Choose $\eta \in \{1.5, 0.25, 0.03, 0.005, 0.001\}$.

Warning: If you implement *AdaGrad* in MATLAB, make sure that your denominator is non-zero. MATLAB may not give special warning on this.

Important: Note that some of the above algorithms update the weight vector even when the weight vector does not make a mistake on the current example. In some of the following experiments, you are asked to calculate how many mistakes an online algorithm makes during learning. In this problem set, the definition of the number of mistakes is as follows: for every new example (x, y), if $y(w^\intercal x + \theta) \leq 0$, the number of mistakes will be increased by 1. So, the number of mistakes an algorithm makes does not necessary equal the number of updates it makes.

Data generation

This section explains how the data to be used is generated. We recommend that you use the python file **gen.py** to generate each of the data sets you will need. The input parameters of this data generator include l,m,n, the number of instances, and a $\{True, False\}$ variable *noise*. When *noise* is set $False$, it will produce **clean** data. Otherwise it will produce noisy data. (Make sure you place **add_noise.py** with **gen.py** in the same workspace. See Problem 3 for more details.) Given values of l, m, n and *noise* set $False$, the following call generates a clean data set of 50,000 labeled examples of dimensionality n in the following way (y and x are Numpy arrays)

```
from gen import gen
(y, x) = gen(l, m, n, 50000, False)
```

For each set of examples generated, half will be positive and half will be negative. Without loss of generality, we can assume that the first m attributes are the relevant attributes, and generate the data this way. (Important: Your learning algorithm does NOT know that.) Each example is generated as follows:

- For each positive example, pick randomly and uniformly l attributes from among $x_1, \ldots, x_m$ and set them to 1. Set the other $m - l$ attributes to 0. Set the rest of the $n - m$ attributes to 1 uniformly with a probability of 0.5.
- For each negative example, pick randomly and uniformly $l - 2$ attributes from among $x_1, \ldots, x_m$ and set them to 1. Set the other $m - l + 2$ attributes to 0. Set the rest of the $n - m$ attributes to 1 uniformly with a probability of 0.5.

Of course, you should not incorporate this knowledge of the target function into your learning algorithms. Note that in this experiment, all of the positive examples have l active attributes among the first m attributes and all of the negative examples have $l - 2$ active attributes among the first m attributes.

Parameter Tuning

One of the goals of this this homework is understanding the importance of parameters in the success of a machine learning algorithm. We will ask you to tune and report the best parameter set you chose for each setting. Lets assume you have the training data set and the algorithm. We now describe the procedure you will run to tune the parameters. As will be clear below, you will run this procedure and tune the parameters for each training set you will use in the experiments below.

Parameter Tuning Procedure

- Generate two distinct subsamples of the training data, each consisting of 10% of the data set; denote these data sets D_1 and D_2 respectively. For each set of parameter values that we provided along with the algorithm, train your algorithm on D_1 by running the algorithm 20 times over the data. Then, evaluate the resulting model on D_2 and record the accuracy.

- Choose the set of parameters that results in the highest accuracy on D_2.

Note that if you have two parameters, a and b, each with 5 options for values, you have $5 \times 5 = 25$ sets of parameters to experiment with.

Experiments

Note: For the following experiments, you will generate data sets for multiple configurations of l, m, and n parameters. For each configuration, make sure to use the **same** training data and the **same** testing data across all learning algorithms so that the results can be compared across algorithms.

1. **[20 points] Number of examples versus number of mistakes**

   First you will evaluate the online learning algorithms with two concept parameter configurations: (a) $l = 10$, $m = 100$, $n = 500$, and (b) $l = 10$, $m = 100$, $n = 1000$.

   Your experiments should consist of the following steps:

   (a) You should generate a **clean** data set of 50,000 examples for each of the two given l, m, and n configuration.

   (b) In each case run the tuning procedure described above and record your optimal parameters.

Algorithm	Parameters	Dataset n=500	Dataset n=1000
Perceptron			
Perceptron w/margin			
Winnow			
Winnow w/margin			
AdaGrad			

   (c) For each of the five algorithms, run it with the best parameter setting over each training set once. Keep track of the number of mistakes (W) the algorithm makes.

   (d) Plot the cumulative number of mistakes made (W) on N examples ($\leq 50,000$) as a function of N (i.e. x-axis is N and y-axis is W)[2]

   For each of the two datasets (n=500 and n=1000), plot the curves of all five algorithms in one graph. Therefore, you should have two graphs (one for each dataset) with five curves on each. Be sure to label your graphs clearly!

   Comment: If you are getting results that seem to be unexpected after tweaking the algorithm parameters, try increasing the number of examples. If you choose to do so, don't forget to document the attempt as well. It is alright to have an additional graph or two as a part of the documentation.

[2]If you are running out of memory, you may consider plotting the cumulative error at every 100 examples seen instead.

2. **[35 points] Learning curves of online learning algorithms**

   The second experiment is a learning curve experiment for all the algorithms. Fix $l = 10$, $m = 20$. You will vary n, the number of variables, from $n = 40$ to $n = 200$ in increments of 40. Notice that by increasing the value of n, you are making the function sparse. For each of the 5 different functions, you first generate a dataset with $50,000$ examples. Tune the parameters of each algorithm following the instructions in the previous section. Note that you have five different training sets (for different values of n), so you need to tune each algorithm for each of these separately. Like before, record the chosen parameters in the following table:

Algorithm	Parameters	n=40	n=80	n=120	n=160	n=200
Perceptron						
Perceptron w/margin						
Winnow						
Winnow w/margin						
AdaGrad						

   Then, run each algorithm in the following fashion:

   (a) Present an example to the learning algorithm.

   (b) Update the hypothesis if needed; keep track of the number W of mistakes the algorithm makes.

   Keep doing this, until you get a sequence of R examples on which the algorithm makes no mistakes. Record W at this point and stop.

   For each algorithm, plot a curve of W (the number of mistakes you have made before the algorithm stops) as a function of n on one graph. Try this with convergence criterion $R = 1000$. It is possible that it will take many examples to converge[3] ; you can do it by cycling through the training data you generated multiple times. If you are running into convergence problems (e.g., no convergence after cycling through the data more than 10 times), try to reduce R, but also think analytically about the choice of parameters and initialization for the algorithms. Comment on the various learning curves you see as part of your report.

3. **[45 points] Use online learning algorithms as batch learning algorithms**

   The third experiment involves running all learning algorithms in the batch setting with noisy training data. In the batch setting, you will still make weight updates per mistake, but will loop over the entire training data for multiple training cycles. The steps of this experiment are as follows:

[3] If you are running into memory problems, make sure you are not storing extraneous information, like a cumulative count of errors for each example seen.

(a) [**Data Generation**] For each configuration (l, m, and n), generate a **noisy** training data set with 50,000 examples and a **clean** test data set with 10,000 examples.

```
(train_y,train_x) = gen(l,m,n,50000,True);
(test_y,test_x) = gen(l,m,n,10000,False);
```

In the noisy data set, the label y is flipped with probability 0.05 and each attribute is flipped with probability 0.001. The parameters 0.05 and 0.001 are fixed in this experiment. We **do not** add any noise in the test data.

You will run the experiment with the following three different configurations.

- P1: l = 10, m = 100, n = 1000.
- P2: l = 10, m = 500, n = 1000.
- P3: l = 10, m = 1000, n = 1000.

(b) [**Parameter Tuning**] Use the Tuning Procedure defined earlier on the noisy training data generated. Record the best parameters (three sets of parameters for each algorithm, one for each training set).

Since we are running the online algorithms in a batch process, there should be, in principle, another tunable parameter: the *number of training cycles* over the data that an algorithm needs to reach a certain performance. We will not do it here, and just assume that in all the experiments below you will cycle through the data **20** times.

(c) [**Training**] For each algorithm, train the model using 100% of the noisy training data with the best parameters selected in the previous step. As suggested above, run through the training data 20 times. (If you think that this number of cycles is not enough to learn a good model you can cycle through the data more times; in this case, record the number of cycles you used and report it.)

(d) [**Evaluation**] Evaluate your model on the test data. Report the accuracy produced by all the online learning algorithms. (That is, report the number of mistakes made on the test data, divided by 10,000).

Recall that, for each configuration (l, m, n), you have generated a training set (with noise) and a corresponding the test set (without noise), that you will use to evaluate the performance of the learned model. Note also that, for each configuration, you should use the **same** training data and the **same** test data for all the five learning algorithms. You may want to use the **numpy.save** and **numpy.load** commands to save the datasets you have created to disk and load them back.

Use the table below to report your tuned parameters and resulting accuracy.

Algorithm	m=100		m=500		m=1000	
	acc.	params.	acc.	params.	acc.	params.
Perceptron						
Perceptron w/margin						
Winnow						
Winnow w/margin						
AdaGrad						

Write down your observations about the resulting performance of the algorithms. Be sure to discuss how the results vary from the previous experiments?

4. [**10 points**] **Bonus:** In our experiments so far, we have been plotting the misclassification error (0-1 loss) for each of the algorithms. Consider the **AdaGrad** algorithm, where we learn a linear separator by minimizing a *surrogate* loss function – Hinge loss instead of directly minimizing the 0-1 loss. In this problem, we explore the relationship between the 0-1 loss and the *surrogate* loss function.

   We will use the **AdaGrad** update rule which was derived using **Hinge Loss** as our loss function. Run the algorithm for 50 training rounds using the batch setting of Problem 3. At the end of each round, record the misclassification error and the Hinge loss over the dataset for that round. Generate two plots: misclassification error as a function of the number of training rounds, Hinge loss (over the dataset) as a function of the number of training rounds.

   We will use a **noisy** dataset consisting of 10000 instances. Use the configuration where $l = 10$, $m = 20$ and $n = 40$ (from Problem 2). Use the procedure **gen.py** to generate the training data as follows:

```
(data_y,data_x) = gen(l,m,n,10000,True);
```

   Recall that, once you generate the data, run the training procedure for 50 rounds and obtain the required plots for misclassification error against the number of training rounds and risk (loss over the dataset) against the number of training rounds. You can re-use the parameters obtained in Problem 2. Write down your observations about the two plots obtained. Feel free to experiment with other values of n and plot them in the same graphs.

What to submit

- A detailed report. Discuss differences you see in the performance of the algorithms across target functions and try to explain why. Make sure you discuss each of the plots, your observations, and conclusions.
- Three graphs in total, two from the first experiment (2 different concept parameter configurations) and one from the second experiment (changing the number of variables n, but keeping l and m fixed), each clearly labeled. Include the graphs in the report. If you attempt the bonus problem, you should have two additional graphs to submit.
- One table for each of the first two experiments; three tables for the third experiment, from P1 through P3. Include the tables in the report.
- Your source code. This should include the algorithm implementation and the code that runs the experiments. You **must** include a README, documenting how someone should run your code.

Comment: You are highly encouraged to start on this early because the experiments and graphs may take some time to generate.