CS446: Machine Learning	**Fall 2014**
Problem Set 3	
Handed Out: September 29^{th}, 2014	*Due: October* 10^{th}, 2014

- 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 manuscript as a single pdf file.
- A large portion of this assignment deals with programming 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 several online learning algorithms, including *Winnow*, *Perceptron* and *AdaGrad* (both with and without margin), and experiment with them by comparing their performance on synthetic datasets. We supply Matlab code that can help you generate the data you will be using in this problem set; however, you *do not need to* code your algorithms in Matlab.

In the course of this problem set you will get to see the impact of parameters on the performance of learning algorithms and, most importantly, the difference in the behavior of learning algorithms when the target function is sparse and dense. During experimentation, you will generate examples that are labeled according to a simple *threshold function*, an l-of-m-of-n function. That is, the function is defined on the n-dimensional Boolean cube $\{0,1\}^n$, 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 function that your your algorithms attempt to learn.

To make sure you understand the function, try to write it for yourself as a linear threshold function. This way, you will make sure you understand that Winnow, Perceptron and AdaGrad can represent this function. Also, notice that this concept 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 (that is, it knows n, of course, but it does not know m or l).

The goal is to evaluate various online algorithms under several conditions, derive some conclusions on their relative advantages and disadvantages, and get some experience in running machine learning experiments.

The instance space is $\{0,1\}^n$ (that is, there are n boolean features in the domain). You will run experiments for several values of l, m, and n.

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. Besides the fixed initialization, we will also give you several options for the parameter(s) of each algorithm, and you will run a tuning procedure on part of each given training set, to determine the best set of parameters for this training set. The **Parameter Tuning** section below gives information on how to do that.

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

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

, where (w, θ) are the learned parameters of the target function.

1. **Perceptron**: The simplest, pure version of the Perceptron Algorithm. With this algorithm, the current weights will be updated following the observation of a training example (x, y) such that $y(w^\intercal x + \theta) \leq 0$.

   Note that the Perceptron algorithm needs to learn both the 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:

   $$\begin{aligned} w_{t+1} &\leftarrow w_t + \eta y x \\ \theta_{t+1} &\leftarrow \theta_t + \eta y \end{aligned}$$

   where η is the learning rate.

   Note that if we assume that the order of the examples presented to the algorithm is fixed, and we initialize $\begin{bmatrix} w & \theta \end{bmatrix}$ to be zero and learn w and θ using the update rules above, then the learning rate η, in fact, does not have any effect. Under this assumption, if w_1 and θ_1 is the output of the Perceptron algorithm with learning rate η_1, it is easy to see that 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).

   Parameters: Given the discussion above, we can fix the learning rate to $\eta = 1$, and there are no parameters to tune.

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

2. **Perceptron with margin**: This algorithm is similar to Perceptron; but in this algorithm, a weight update will be performed after observing the example (x, y) if $y(w^\intercal x + \theta) \leq \gamma$, where γ is an additional **positive** parameter of the algorithm. 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 $\gamma = 1$.

   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. Since we have fixed the value of γ, we only need to tune η.

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

   Parameter Candidate Set:

   $\eta \in \{1.5, 0.25, 0.03, 0.005, 0.001\}$

3. **Winnow**: The simplest, pure 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:

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

   where α is the promotion/demotion parameter and $w_{t,i}$ is the i-th component of the weight vector after the ts mistake.

   Parameters: Promotion/demotion parameter α

   Initialization: $w^\intercal = \{1, 1, ..., 1\}, \theta = -n$ (θ is fixed here, we don't update it.)

   Parameter Candidate Set: $\alpha \in \{1.1, 1.01, 1.005, 1.0005, 1.0001\}$

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 $y(w^\intercal x + \theta) \leq \gamma$, where γ is an additional **positive** parameter specified by the user. Note that, as in the case of Perceptron with margin, the 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: $w^\intercal = \{1, 1, ..., 1\}, \theta = -n$ (θ is fixed here, we don't update it.)

   Parameter Candidate Set:

   $\alpha \in \{1.1, 1.01, 1.005, 1.0005, 1.0001\}$

   $\gamma \in \{2.0, 0.3, 0.04, 0.006, 0.001\}$

5. **AdaGrad**: AdaGrad alters the learning rate to adapt 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 $j = 1, ..., n$ 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),

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

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:

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

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

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

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

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

Parameter Candidate Set:

$\eta \in \{1.5, 0.25, 0.03, 0.005, 0.001\}$

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: Some 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 count the number of 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 is increased by 1. So, the number of mistakes an algorithm makes does not necessarily 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 Matlab file **gen.m** 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 $\{0, 1\}$ variable *noise*. When *noise* is set 0, it will produce clean data. Otherwise it will produce noisy data. (Make sure you place **addnoise.m** with **gen.m** in the same workspace. See Problem 3 for more details.) Given values of l, m, n and *noise* set 0, the following call generates a **clean** data set of 50,000 labeled examples of dimensionality n in the following way.

```
[y,x] = gen(l,m,n,50000,0);
```

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 to understand 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. Let's 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:

1. 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 D_2. Record the accuracy of your model on D_2.

2. 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

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

   First you will evaluate the five online learning algorithms with data generated according to two target function 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	Data Set	
		$n = 500$	$n = 1000$
Perceptron w/margin	η		
Winnow	α		
Winnow w/margin	α		
	γ		
AdaGrad	η		

   (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)[1].

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

For each of the two data sets ($n = 500$ and $n = 1000$) plot the curves of all five algorithms in one graph. That is, you should have two graphs with five curves on each. Be sure to label your graphs clearly!

Comment: If you are getting results that seem to be unexpected despite using the best parameters (which you tuned earlier), 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. **[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. 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. Noth that you have five different training sets, so you need to tune each algorithm for each of these separately. Like before, record the chosen parameters in the following table:

Algorithm	Parameters	Data Set				
		$n = 40$	$n = 80$	$n = 120$	$n = 160$	$n = 200$
Perceptron w/margin	η					
Winnow	α					
Winnow w/margin	α					
	γ					
AdaGrad	η					

   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[2]. Run one as many examples as needed, by cycling through the training data you generated multiple times. If you are running into convergence problems (e.g., 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 about the various learning curves you see as part of your report.

[2]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.

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

In the third experiment you will evaluate all five algorithms in a more realistic setting. To do this, you will follow the following steps:

(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. You should use the function in the following way:

```
[train_y,train_x] = gen(l,m,n,50000,1);
[test_y,test_x] = gen(l,m,n,10000,0);
```

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 on test data 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, 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 testing data for all the five learning algorithms. You may want to use the **save** and **load** commands to save the datasets you have created to disk and load them back.

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

Algorithm	Parm & Accy	Data Set		
		$m = 100$	$m = 500$	$m = 1000$
Perceptron	Accy %			
Perceptron w/margin	η			
	Accy%			
Winnow	α			
	Accy%			
Winnow w/margin	α			
	γ			
	Accy%			
AdaGrad	η			
	Accy%			

Write down what you have observed. For example, which algorithm is the best algorithm in this experiment? Is this the same one you found to be the best in the second experiment?

4. [**10 points**] **Bonus:** In our experiments so far, we have been working with balanced data sets, i.e. an equal number of positive and negative examples are provided. Consider the situation where we have a relatively unbalanced **clean** data set, where the number of negative examples is 90% of the data set, and the positive ones make up the remaining 10%. Under such an unbalanced data set, the baseline result to beat is no longer 50% accuracy, but rather 90%, obtained when we naively output -1 for every example sent to the classifier. Modify the basic ***Perceptron*** algorithm so that it can handle such unbalanced data sets and show that it can outperform the baseline result empirically. Report two parameter sets and two accuracy values, i.e. Perceptron before modification and after modification on this unbalanced data set. Do this in the batch setting of Problem 3.

   To generate an unbalanced data set, use the procedure **unba_gen.m** we provided in the following ways (0.1 is the percentage of positive instances, and 50000 is the total number of instances):

```
[train_y,train_x] = unba_gen(l,m,n,50000,0.1);
[test_y,test_x] = unba_gen(l,m,n,10000,0.1);
```

   Recall that, once you generate the data, you first need to tune the algorithm's parameters (unless you keep the basic Perceptron without tunable parameters), train it, and then test it.

What to submit

- A detailed yet concise report. Describe what you did. Comment on your design decisions in your implementation. 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. Try to make your report interesting.
- 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.
- One table for each of the first two experiments; three tables for the third experiment, one for 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.

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