CS 5350/6350: Machine Learning Spring 2015

Homework 2

Anirudh Narasimhamurthy(u0941400)

February 18, 2015

1 Boolean Functions

In this problem, we are asked to write Boolean functions and linear threshold functions based on given labeled data.

1. In this question we are asked to come up with Boolean functions which will produce the label y when the input provided are x_1, x_2, x_3, x_4. The label y and the inputs are shown below:

y	x_1	x_2	x_3	x_4
0	1	0	0	0
0	1	1	0	0
1	1	0	1	1

Table 1: Original Table

From the three inputs and the labels for y we can clearly see that the label has value 1 if either $x3$ or $x4$ is 1 or it could be the case of both x_3 and x_4 must be 1 for the label to be 1. Based on this, the following are some of the Boolean functions which satisifies those constraints:

BF 1. $(x_1 \vee x_2) \wedge x_3 \wedge x_4$
BF 2. $(x_1 \vee x_2 \vee x_3) \wedge x_4$
BF 3. $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$

2. After expanding the table1 with more entries to get Table 2 as shown below, the following would be the number of errors that each of the functions described in [1] would make:

BF 1 — 0 errors
Reason: As it considers both x_3 and x_4 to be equal to 1 for the label to be 1.

BF 2 — 0 errors
Reason: Though we might not know the correct function, the examples provided are

in such a way that if x_4 is not equal to 1, the label for y is always zero. Hence **BF 2** produces zero errors.

BF 3 —2 errors.
Reason: For two examples which have x_3 alone as 1 and x_4 being zero, the function BF 3 would produce a label 1. But the actual labels for them are 0.

y	x_1	x_2	x_3	x_4
0	1	0	0	0
0	1	1	0	0
1	1	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	1	1	0
1	0	1	1	1

Table 2: Expanded Table

3. Based on the data in table 2 one possible linear threshold function could be the one provided below:

   $x_3 + x_4 = 2$

   This is based on the fact the label y will have a value 1 only when both x_3 and x_4 have a value of 1 and is independent of the other two variables x_1 and x_2.

   Again this holds true only for the given training examples and the LTU might even change to $x_4 = 1$ if the other unseen training examples proves so.

   LTU classifies an example x using the following classification rule:

   Output=sgn(w^Tx +b) =sgn(b +$\sum w_i x_i$) If the sign is negative , label is 0 else label is 1. The above LTU can be effectively described as w3 and w4=1 and w1 and w2=0 and a constant bias term of -2 or like in the notes assume it to be w0 which will always be 2.

y	x_1	x_2	x_3	x_4	$x_3 + x_4 - 2$	y'
0	1	0	0	0	-2	0
0	1	1	0	0	-2	0
1	1	0	1	1	0	1
0	0	1	0	0	-2	0
0	0	1	1	0	-1	0
0	1	1	1	0	-1	0
1	0	1	1	1	0	1

Table 3: Expanded table containing LTU and y'

Another possible and slightly more involved LTU for the given table would be : $2x_3 + 2x_4 \geq x_1 + x_2 + 2$

This has been written considering the fact that both x_3 and x_4 should be 1 at the same time for the label to be 1. Also from the examples seen in the expanded table, we know that $x_3 = 1$ alone cannot make our final label of y=1. Hence the bias term of -2 would take care of that.

In other words the above LTU can be described as x_1 and x_2 having negative weights of 1. And x_3 and x_4 having positive weights of 2 and a bias term of -2 This would ensure all examples gets the correct label as per the given data.

y	x_1	x_2	x_3	x_4	$2x_3 + 2x_4 - (x_1 + x_2 + 2)$	y'
0	1	0	0	0	-3	0
0	1	1	0	0	-4	0
1	1	0	1	1	1	1
0	0	1	0	0	-3	0
0	0	1	1	0	-1	0
0	1	1	1	0	-2	0
1	0	1	1	1	1	1

Table 4: Expanded table containing LTU and y'

2 Mistake Bound Model of Learning

1. **Size of the concept class**

   We are given an instance space consisting of integer points on the 2D plane. Each function in the concept class f_r is defined by radius r and $(1 \leq r \leq 128)$.

   By definition we know the concept class is the set of functions from which the true target function is drawn. And we see that each function is defined by the limit on r. Hence the size of concept class, $|\mathcal{C}|$ is **128**

2. **Expression to check whether a given expression has made a mistake**

   We know that if $x_1^2 + x_2^2 \leq r^2$ the label should be +1 , else it should be -1. We can consider a mistake will be made if the following expression is not satisfied:

   $$\sum_{i=0}^{t} [f_r(x_1^t, x_2^t) == y^t]! = t$$

   This effectively implies that if the function f_r correctly classifies the example with the label then we move to the next function and if all of them had classified correctly up till now then summation would be equal to t going by our expression. If it doesn't add up then it is an indication thatbone of the functions in the concept class has classified the label incorrectly. This works for both positive and negative examples.

3. **Update r**

   One way of thinking on how to update r would be to think when a mistake would be made on example. If the positive example has been incorrectly classified with a label

'-' then we add +r and if a negative example had been classified incorrectly as '+' then we subtract '-r'. Since we know r is bound between 1 and 128 this update will help us in finding out the correct target function when r either reaches 1 or 128.

Another way or school of thought as per the CON algorithm given in the slides would be to check for those values of 'r' for which $f_r(x_1, x_2)! = y$ and then remove those r's from our list. So this would effectively imply we would be left with one value of r lesser and would have made progress.

4. **Pseudocode for Mistake-driven Learning Algorithm**

Let C be concept class

```
t=0
while C_t != 1:
 t=t +1
 Choose a random function f_r in C
 for i=0 to t:
  if  f_r(x1_i,x2_i) != y_i
   error=yes
   remove r from C
   break
  if error==yes
   C_t+1=C_t  - delete(r)
```

Idea here is everytime there is an error we try to remove that function from the concept space and work with the remaining functions which are consistent on the diffeerent examples. Our concept class size reduces by 1. This implies we have made progress.

Maximum number of mistakes that the algorithm makes on any dataset would be given by $|\mathcal{C}|$ -1. For our concept space this would be 127.

5. **Halving Algorithm**

   (a) **Defining the set of hypothesis by storing only two integers**

   We know that the concept class size is 128. At any stage set of hypothesis consistent with all examples seen so far can be defined in terms of either the number 1 or 128 and the value of r.

   For instance if the r value is decreasing then having [1,r] can be used to represent all the examples which have been consistent so far. If r was increasing then [r,128] would help describe.

   As and when you do a mistake the r value is updated making sure it is consistent with the set of examples seen so far.

   (b) In halving algorithm we consider the majority of the values to classify an example with a label. So the below expression will help is check if there is an error for an example (x_1^t, x_2^t) that has the label y^t.

$$\sum_{r=0}^{128} f_r(x_1^t, x_2^t) \leq 0? -1 : +1)! = y^t$$

Basically we take the input and run it through the function. Based on the value returned by the function, if it is lesser than or equal to zero then we give the label -, else we give the label '+'. And if y^t value does not match with our predicted label, then its an error.

(c) **Halving Algorithm**

```
Let C be finite concept class.
Initialize C_o=C
while  C_i != 1:
Pick an example
For each function  in concept class, pick a function f_i
apply it to the training examples:
If value_calculated  > 0:
 label= +
 funclist=f_i
Else:
 label=-1
 funcnegative_list=f_i
If label != y':
 if y' != f_r(x_i,y_i):
 Remove those functions from the concept space
  C_i+1=C_i -len{funclist}
```

The logic used here we pick a single example and we send it to all the functions in concept class. The functions are applied on this example and each of them returns their own label. The majority vote is used to determine what the final label would be. If the label voted by majority is incorrect then all the functions which voted for are removed or thrown away. This ensures that at every stage atmost half of the nodes are thrown away. We wait until we have $C_i = 1$ which implies we have found our target function.

(d) **Mistake bound of Halving Algorithm**

Mistake bound for the halving algorithm is given by $log_2 N$.That is the halving algorithm makes atmost logN mistakes. For our concept space the mistake bound would be $log_2(128)$ which gives us a value of **7**

3 The Perceptron Algorithm and Its Variants

3.3 Experiments

1. **Sanity Check**

The simple perceptron algorithm was run on Table 2 for one pass. I tried experimenting with both random weights and random bias as well as having the bias and weights

initialized to zero. I have described the results below:

When weight vectors and bias are randomly generated

The weight vector which was generated for the given training data is :

[**-0.21128718 0.43170958 0.21082405 0.02697772**]

The bias that was generated was : **0.104427398404**

After the perceptron updates, the weight vector that the algorithm returned after 1 pass was :

[**-0.21128718 0.43170958 0.21082405 0.02697772**]

The updated bias was : **0.104427398404**

And the **Number of mistakes made :5**

Incorrect prediction for : [1 0 0 0]
Incorrect prediction for : [1 1 0 0]
Incorrect prediction for : [0 1 0 0]
Incorrect prediction for : [0 1 1 0]
Incorrect prediction for : [1 1 1 0]

We can see that the weight vector or the bias hasn't been updated in this case. This is because w= w+ r.x.y will always be zero for an incorrect prediction on an example with label 0. Similarly the bias b= b+ r.y will also be zero. The five mistakes which the algorithm made were all on examples which had label 0. Hence for the given table data the updated weight vector is same as the initial weight vector.

However if the algorithm makes a mistake on an example with label 1, the weight vectors and bias gets updated like in this case shown below:

The weight vector generated for the given training data is :
[**-0.3064184092667379, -0.6362682286577468,**
-0.8382228996031185, 0.7388938678944432]

The updated Weight vector at the end of one pass is :
[**-0.30641841 -0.53626823 -0.7382229 0.83889387**]

In this case the number of mistakes made was 6 and the mistakes were :

Incorrect prediction for : [1 0 0 0]
Incorrect prediction for : [1 1 0 0]
Incorrect prediction for : [0 1 0 0]
Incorrect prediction for : [0 1 1 0]
Incorrect prediction for : [1 1 1 0]

Incorrect prediction for : [0 1 1 1]

As you can see for the example [0 1 1 1] which has label 1 is incorrectly predicted and so the weight vector updates.

One way of getting around this problem is to probably have the label as -1 instead of 0 in which case the algorithm would be able to learn faster.

But if we ignore this fact and try to focus on what the question intended to ask the Average number of mistakes which the simple perceptron makes over the Table 2 on single pass is 5. To be even more precise it makes 5.4 mistakes on an average over 10 separate runs. Weight vector gets updated when there is a mistake on the example with label 1.

2. **Online settings**

In this part of the assignment we are asked to run the perceptron and the margin perceptron algorithm on the Adult data set for one pass.

To apply the perceptron algorithm on the Adult data set, I did some data munging to convert the data given in the LIBSVM format to a form which would be comfortable for us to handle the algorithm. I stored the labels present at the start of each line of the test or training data in a separate list in python. And using ':' as delimiter and with a bit of cleaning, i was able to form the feature vector 'x' out of the given data.

To determine the length of the feature vector, I performed a check over the last feature vector specified in each line and I found that for training data, the feature vector length was 119. So I initialized my feature vector to all zeros and then updated those indexes as 1, which had been mentioned in the sparse representation in the LIBSVM file format.

The weight vector was also assigned the same size as that of the feature vector. This was done because that would make the dot product much easier. Also the values of the weight vector was assigned with random reals between -1 and 1.

I then simply followed the updates in the algorithm and have implemented it. A mistake is said to be made if

$$y(w^T x + b) \leq 0$$

. Whenever a mistake is done, we update the weight vector w and the bias term b with the new weights and bias. The update is given by :

$$\mathbf{w}_{new} \leftarrow \mathbf{w}_{old} + ry\mathbf{x}$$

$$b_{new} \leftarrow b_{old} + ry.$$

Simple perceptron algorithm

The number of updates made and the accuracy on the test and training set varies based on the hyper parameter 'r'. The general trend which I observed was when the value of r was smaller say 0.01, the mistakes were more and accuracy was less. But as I changed/increased the r values to 0.1 to 0.5 and to 1, I could see the improvement in accuracy.

In a way it definitely makes sense, as r being the learning rate makes lesser mistakes when the rate is high. It indicates the fact that the algorithm learns from its mistakes and is making progress by making lesser and lesser mistakes. The expected values are provided below:

Case 1: When r=1

- No of updates/mistakes made: **368**
- Accuracy of the final weight vector on training data: **82.74%** [277 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **81.984%** [5577 mistakes/30956]

Case 2: When r=0.1

- No of updates/mistakes made: **390**
- Accuracy of the final weight vector on training data: **79.624%** [327 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **78.495%** [6657 mistakes/30956]

Case 3: When r=0.01

- No of updates/mistakes made: **453**
- Accuracy of the final weight vector on training data: **76.386%** [379 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **77.732%** [6893 mistakes/30956 examples]

So its clear that tweaking the hyper paramter 'learning rate' changes the accuracy and the number of mistakes/updates made. For r=1 we have higher accuracy and lesser updates then when compared to r=0.01. However since our assignments are random there might also be situations where I experienced both r=0.1 and r=1 having nearly the same value for accuracy. In general this parameter is good way to check the mistake /learning rate.

Margin perceptron algorithm

In the margin perceptron algorithm we will perform an update on an example (x,y) if $y(w^T X + b) \leq \mu$ where μ is a positive hyper parameter. For our case, we are usually setting value of μ between 0 and 5 as suggested in the assignment.

An important point to be noted is this μ will only be used during the update. After we obtain the updated weight vector, while checking the accuracy on test or training data we will only be checking if $w^T x + b \leq 0$. If it is less than or equal to zero, we assign a label of -1 to it else we assign +1 to it. We then compare our prediction with the actual label. Accuracy is then computed by the number of mistakes we make in the test and also in this case the training examples.

For margin perceptron in addition to the learning rate hyperparameter, we also have the paramter μ which we need to experiment with.

The values obtained are reported below:

Case 1: When r=0.01 and mu=5

- No of updates/mistakes made: **865**
- Accuracy of the final weight vector on training data: **75.887%** [387 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **76.392%** [7308 mistakes/30956]

Case 2: When r=0.1 and mu=5

- No of updates/mistakes made: **701**
- Accuracy of the final weight vector on training data: **83.177%** [270mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **83.442%** [5125 mistakes/30956]

Case 3: When r=1 and mu=5

- No of updates/mistakes made: **479**
- Accuracy of the final weight vector on training data: **83.86%** [259 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **83.0722%** [5240 mistakes/30956 examples]

These values again show that as the learning rate parameter goes higher and higher the number of mistakes goes down.

I had also experimented with learning rate r=1 and changing μ values from 0.1 to 2 to 1.5 to 5.0. But the observation which I made was once the learning rate was at its highest, irrespective of the other value of the hyperparameter mu, the accuracy and the number of mistakes more or less remained the same.

This again is not suprising as we use r in the perceptron and margin perceptron updates and that updated weight vector determines the accuracy. We use μ to determine if a mistake has been made. It could be thought in terms of a mistake threshold.

3. **Using online algorithms in batch settings:**

In this part of the experiment we would be experimenting with an additional hyper parameter **Number of epochs**. For the given question we are asked to experiment with values of 3-epochs and 5-epochs. In our earlier questions we performed just one pass over the training data. In batch settings we would be doing multiple passes determined by the epoch

Another additional change that we make to the code is that we shuffle the training data before starting each epoch. The weight vectors, bias are set with random values and we set μ and learning rate with fixed values.

Simple Perceptron - 3 Epochs

- No of updates/mistakes made: **1058**
- Updates made in each iteration : **[385,338,335]**
- Accuracy of the final weight vector on training data: **81.6822%**[294 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **79.936%.** [6211 mistakes/30956 examples.]

Bias: 0.062684913632

r= 0.1

Margin Perceptron - 3 Epochs

- No of updates/mistakes made: **1605**
- Updates made in each iteration : **[608,524,473]**
- Accuracy of the final weight vector on training data: **84.299%**[252 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **83.1116%.** [5228 mistakes/30956 examples.]

Simple Perceptron - 5 Epochs

Bias: 0.216625037303

r= 0.1

- No of updates/mistakes made: **1736**
- Updates made in each iteration : **[368,347,356,323,342]**
- Accuracy of the final weight vector on training data: **80.0623%**[320 mistakes/1605 examples]

- Accuracy of the final weight vector on test data: **79.9263%.** [6214 mistakes/30956 examples.]

Margin Perceptron - 5 Epochs

mu=2, r=0.1

- No of updates/mistakes made: **2651**
- Updates made in each iteration : **[602,505,535,497,512]**
- Accuracy of the final weight vector on training data: **83.3644%**[267 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **82.0648%.** [5552 mistakes/30956 examples]

Explanation of Results:

The details provided above are for one particular value of 'r' and 'mu' with the weights and bias being randomly generated. Based on the values which we see from the table, we find that the accuracy for the 3 epochs is slightly greater than when we use 5 epochs. But then again, there is a lot of randomness involved with the weight vectors and bias. The fact that the training data is also shuffled at the beginning of every epoch will make the algorithm train the data well and have a good accuracy in test examples.

One immediate effect which we can see from the values of simple perceptron for online and batch learning is that the accuarcy on the test data increases by a little for the batch learning when we have 3 or 5 epochs. This again comes from the fact that we would be using the updated weight vector on the training data in the following iteration. In a rough sense it is equivalent to finding the weight vector and then applying it back on the training examples to get the accuracy. That particular scenario as I have observed always shows an improved accuracy.

Also the no of updates/mistakes in every epoch decreases most of the time. This is understandable as the perceptron corrects or learns by making the updates to the weights and bias and it gets to apply it on the same training data albeit in a different order, as training data is shuffled at the start. This could be seen in values mentioned in the table and this makes intuitive sense. in the cases of no of updates increasing, it could be attributed to the shuffling of the training data as well as the ohter random factors.

4. **Aggressive perceptron with Margin:**

In this part of the question we are asked to implement an extension of the margin perceptron which performs aggressive update as follows:

If $y(\mathbf{w}^T\mathbf{x} + b) < \mu$, then update

(a) $\mathbf{w}_{new} \leftarrow \mathbf{w}_{old} + \eta y x$

(b) $b_{new} \leftarrow b + \eta y$,

Unlike the standard Perceptron algorithm, here the learning rate η is given by

$$\eta = \frac{\mu - y(\mathbf{w}^T\mathbf{x} + b)}{\mathbf{x}^T\mathbf{x} + 1}$$

This question as I understood was to to perform the batch setting for just the aggressive perceptron instead of doing it for both margin and normal perceptron. The results are shown below:

Aggressive Perceprton - 5 Epochs -Without Shuffle

- No of updates/mistakes made: **3307**
- Updates made in each iteration : [**708, 668, 653, 645, 633**]
- Accuracy of the final weight vector on training data: **83.177%**[270 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **82.9952%.** [5264 mistakes/30956 examples]

Aggressive Perceprton - 5 Epochs -With Shuffle

- No of updates/mistakes made: **3325**
- Updates made in each iteration : [**703, 645, 639, 627, 611**]
- Accuracy of the final weight vector on training data: **74.953%**[402 mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **72.784%.** [8246 mistakes/30956 examples]

Aggressive Perceprton - 3 Epochs -Without Shuffle

- No of updates/mistakes made: **2030**
- Updates made in each iteration : [**710, 667, 653**]
- Accuracy of the final weight vector on training data: **83.052%**[272mistakes/1605 examples]
- Accuracy of the final weight vector on test data: **82.988%.** [5266 mistakes/30956 examples]

Aggressive Perceprton - 3 Epochs -With Shuffle

- No of updates/mistakes made: **2042**
- Updates made in each iteration : [**731, 662, 649**]
- Accuracy of the final weight vector on training data: **77.0716%**[368 mistakes/1605 examples]

- Accuracy of the final weight vector on test data: **77.7652%.** [6883 mistakes/30956 examples]

We can clearly observe that the number of mistakes made in the aggressive perceptron is comparitively high with all other methods. However the learning/update is so good that it can almost classify the example which it had classified incorrectly to classifying it correctly right after it updates its weight vector. This is clearly observed in accuracy rates on the training data.

The batch experiments also show that shuffling the data causes the accuracy to go down as the updated weight vector encounters a different order of examples at the start of every epoch and so is not able to do really well on accuracy in such short epochs. But if we were to repeator make 100 passes it would probably give a good estimate.

Thus the perceptron and the different variants of perceptron were successfully implmented and the results have been docuemnted. However the results which have been documented alone would not be a true reflection of the work that I had put in and hence if the numbers are off by a bit, I hope my code makes it clear to you with regards to the implementation and that i have reported values which I got on the initial run. Due to the randomness associated, there could/could not be a difference but I hope looking at my code wil probably give you a better picture of the overall efforts put into this project.