CS446: Machine Learning **Fall 2014**

Problem Set Problem 1

Nikhil Unni *Handed In: September 16, 2014*

1. Learning Disjunctions

   a. My algorithm for learning the disjunction is actally to use only the negative examples. Because a single "truth" value in the disjunction will make the output positive, we can iterate through all the negative examples and take out all the "truths." For positive examples, a value of 1 or 0 has very little meaning if there are other features with 1

```
1. Get m training examples of n long Boolean vectors
2. Initialize a set of all 2n possible members of the target function : x_1, ¬x_1, x_2, ¬x_2, ...
3. For each training example, x:
4.     If x is labeled −1:
5.         For i = 1 to n:
6.             Remove the appropriate member, x_i, of the set, i.e. if x_2 is 1, remove x_2,
               or if x_10 is 0, remove ¬x_10
7. If the set is empty:
8.     There exists no consistent hypothesis for the training data
9. Else:
10.    For all remaining members of the set, join them as a disjunction
```

   b. The idea behind the algorithm is that it only takes a single 1 in a disjunction of booleans to make the output positive. So because of this, for each negative example, x, we take out all potential members of the final hypothesis, h, that would make $h(x)$ positive. By this principle, our final h is for sure consistent for all negative examples (unless the data are inconsistent).

   Also, because this is a disjunction, the "right" values will always remain in the final hypothesis. All incorrect additions do not affect the accuracy of the training data (although increase likelihood of false positives when testing it on outside data). For example, if the target function is $x_1 \vee \neg x_4$, the proposed hypothesis has to be of the form : $x_1 \vee \ldots \vee \neg x_4 \vee \ldots$. Just the fact that x_1 and $\neg x_4$ are in the proposed hypothesis, none of the positive examples can be misclassified. This is merely a property of the OR function - If we say the target function is f, and the remaining additions of our hypothesis are h', because $\vee$ is associative, this can be represented as $f \vee h'$. Because f is the target function, if the data are consistent, it will always evaluate to 1. This makes our hypothesis always $1 \vee h'$, which is 1, regardless of what h' is, for all positive examples in the training data. This same reasoning can be used to show that the final hypothesis is consistent

with negative examples, and that there's no way that, in the process of iterating through the training data, the current hypothesis is inconsistent with past indeces of the iteration.

It's also worth mentioning that even in the case that there are no negative examples, our hyper-overfitted hypothesis, $x_1 \vee \neg x_1 \vee x_2 \vee \neg x_2 \vee \ldots$ will simplify to 1, which will be true for all of the training data, merely because there are no negative examples.

c. Initializing the set is an $O(n)$ operation, just adding $2n$ elements to a set.
Then, we double iterate through n and m, and at each stage we remove an element of a set. If we model our "set" as an array of size $2n$, where index 0 is x_1, index 1 is $\neg x_1$, etc. we can instantly access these values (1 if present in the final function, 0 if not) because we have the current index. This makes "removal" from the set $O(1)$. So far that's $O(n) + O(n)O(m)$.
After that, we iterate through the remaining set for the final hypothesis, which is just $O(n)$. Putting that all together, and taking only the significant terms, the algorithm is $O(n^2m)$.

d. I kind of alluded to some cases in part A. But, in general, I'd say that the algorithm increases in robustness with more negative examples. The extreme case with 0 negative examples produces the function $h(x) = 1$, which will match all of the positive test examples, but if the new, unlabed datum is a negative example it will for sure be wrong.
An interesting property of the algorithm (and probably boolean functions in general?) is that it cannot produce a function that gives a false negative value. Again, this is just because the target function is always embedded in our hypothesis. However all of the "extra" function, or h', to go back to my last example, contributes to false positives when trying out on outside data. The more negative data we have in our training set out of the set of all possible negative examples, the more we minimize h', and get closer to the target function f, which, in turn, increases our accuracy for outside data.

2. Linear Algebra Review

a. Pick a random point on the hyperplane, x_1. The vector between x_1 and our point, x_0 is given by $x_0 - x_1$. Next, we know that w is perpendicular to the hyperplane. Because we want this perpendicular distance (the shortest path from the plane to our point), we know that the distance to our point has to be $\|c\vec{w}\|$, for some scalar c.
We can find this scalar by taking the projection of the vector from x_0 to our arbitrary point x_1 onto the perpendicular to the plane, w. This becomes:

$$\begin{aligned} d &= \|proj_w(x_0 - x_1)\| \\ &= \frac{(x_0 - x_1) \cdot w}{\|w\|} \end{aligned}$$

$$= \frac{x_0 \cdot w - x_1 \cdot w}{\|w\|}$$

Because the plane is defined as the constraint of all points x such that $w^T x + \theta = 0$, and x_1 is a point on the plane, $x_1 \cdot w = -\theta$. So now we have:

$$\frac{w^T x_0 + \theta}{\|w\|}$$

But to be strictly correct, the distance should always be positive, giving us:

$$\frac{|w^T x_0 + \theta|}{\|w\|}$$

And that can be evaluated given any plane and point.

b. Similarly, we pick two random points on the planes, x_1 on the first plane (the one with θ_1) and x_2 on the second. And again, it'll be on some projection onto w since the distance between the planes is some scalar multiplied by w. And what we project is the vector $x_1 - x_2$. This leaves us with:

$$d = \|proj_w(x_1 - x_2)\|$$

$$= \frac{(x_1 - x_2) \cdot w}{\|w\|}$$

$$= \frac{x_1 \cdot w - x_2 \cdot w}{\|w\|}$$

Because the two planes are constraints, $w^T x + \theta_1 = 0$, and $w^T x + \theta_2 = 0$, and our points are on those respective planes, both the above dot products can be simplified in terms of θ_i. This leaves us with:

$$\frac{\theta_2 - \theta_1}{\|w\|}$$

And again, because the distance should always be positive, we get:

$$\frac{|\theta_2 - \theta_1|}{\|w\|}$$

3. Finding a Linear Discriminant Function via Linear Programming

a.1.

a.2.

a.3.

b.1. To get to the canonical form:

$$z(t) = c^T t$$

$$s.t. At \geq b$$

If we write out the equations we get $y_i(w^T x_i + \theta) \geq 1 - \delta$.
For positive examples, it becomes $w^T x_i + \theta + \delta \geq 1$ and for negative examples it becomes $-w^T x_1 - \theta + \delta \geq 1$.
If I set $t = [w, \theta, \delta]^T$, an example row in A for a positive example would be $[x_{11}, x_{12}, \ldots, 1, 1]$ and for a negative example it would be $[-x_{21}, -x_{22}, \ldots, -1, 1]$. Finally, the last row of A should include the constraint that $\delta \geq 0$. And so the last row of A is $[0, 0, \ldots, 0, 1]$.
With this same scheme, $b = [1, 1, 1, \ldots, 1, 0]^T$ where the 0 is from the $\delta \geq 0$ constraint.
Similarly, for c we only want to minimize δ so $c = [0, 0, \ldots, 0, 1]^T$. I represent this below in findLinearDiscriminant, in the Code Snippets section at the end of the document.

b.2. [FINISH ME INCLUDE hw1sample2d.txt DATASET PICTURE] For hw1conjunctions.txt, my returned values were (with some loss of precision so they could fit on the screen):

$$w = [2.910, -2.049, 0.177, 190.519, 0.139, -3.101, -2.953, -193.277, 1.167, -8.894]^T$$

$$\theta = -90.2115$$

$$\delta = -2.4158e - 13 \approx 0$$

From this data, it's easy to guess what the target conjunction really is, since w_4 has such a huge weight, and w_8 has such a huge negative weight, the function must be $x_4 \wedge \neg x_8$, while the remaining values of w can just be treated as negligible noise. Upon a cursory glance through hwconjunction1.txt, my prediction for the target conjunction looks correct. Geometrically speaking, this could be thought of in terms of the normal vector of a hyperplane. The large components will "pull" the normal of the plane in such a way that the hyperplane is almost perpendicular to the axis of that component, creating more separation on that component. The threshold will affect [FINISH ME].

b.3. [FINISH ME].

b.4.

4. Code Snippets

b.1.
```
function [w,theta,delta] = findLinearDiscriminant(data)
[m, np1] = size(data);
n = np1-1;

for i=1:m,
    if data(i,n+1) == -1
        data(i,:) = horzcat(data(i,1:n)*-1, data(i,n+1));
    end
end
```

```
A = vertcat( horzcat(data,ones(m,1)) , zeros(1,n+2) ); A(m+1 ,n+2)=1;
b = ones(m+1,1); b(m+1) = 0;
c = zeros(n+2,1); c(n+2) = 1;

[t, z] = linprog(c, -A, -b);

w = t(1:n);
theta = t(n+1);
delta = t(n+2);

end
```

b.2.
```
function plot2dSeparator(w, theta)
    x = linspace(-2,2,100);
    y = (-theta-w(1)*x)/w(2);
    plot(x,y);
end
```

b.3.
```
function y = computeLabel(x, w, theta)
    thresh = dot(x,w)+theta;
    if(thresh >= 0)
        y=1;
    else
        y=-1;
    end
end
```

b.4.
```
function [theta,delta] = findLinearThreshold(data,w)
[m, np1] = size(data);
n = np1-1;

A = zeros(m,2);
b = zeros(m,1);
c = [0,1];

for i=1:m,
    if data(i,n+1) == -1
        A(i,:) = [-1,1];
        b(i,:) = 1 + dot(w,data(i,1:n));
    elseif data(i,n+1) == 1
        A(i,:) = [1,1];
        b(i,:) = 1 - dot(w,data(i,1:n));
    end
```

```
end

A = vertcat(A, [0,1]);
b = vertcat(b, 0);

[t, z] = linprog(c, -A, -b);

theta = t(1);
delta = t(2);

end
```