15-381 Fall '09, Homework 5

- Due: Wednesday, November 18th, beginning of the class
- You can work in a group of up to two people. This group does not need to be the same group as for the other homeworks. You only need to turn in one write-up per group, but you must include the andrew ID's of the group members.
- Late homework is due by 3:00 P.M. on the day they are due. Please bring late homeworks to Susan Hrishenko in GHC 7119 (slide it under the door if she is not there; write on the homework the date and time turned in). *Note that GHC 7th floor is now locked after 6pm. If you finish in the evening, you'll probably need to wait until the next morning to turn it in.*
- Email all questions to 15381-instructors@cs.cmu.edu

1 More Decision Trees- 10 pts

1. (5 pts) Describe how one might use search techniques (genetic algorithms, simulated annealing, etc.) to find a good decision tree. Make sure you address the representation problem.

   Answers were evaluated based on whether the approach was feasible, and whether the representation problem for the search techniques (states, neighbors, fitness function) were addressed. One possible solution was the following:

   Each *state* is a possible decision tree. A *neighbor* of some tree T is T with one split node pruned, or T with one of its leaves being a different split. (This state space will then look like a tree with the root being an empty decision tree). *Fitness function* is the tree evaluated on the training set. (With possibly an additional penalty term for large trees, to help avoid overfitting.)

   For GAs, one could generate decision trees, mutate by randomly choosing a neighbor, and crossing over by, say, picking a random point in one of the trees, pruning it there, and attaching the other tree to that point (although I have no idea if that would perform well). For hill climbing, one could start at random points in the state space and move around the state space as long as the fitness function kept increasing. One would want to do this multiple times, as this will probably get stuck at local optima very often. For simulated annealing, one could take the same approach as hill-climbing only with some probability $1/temp$ move to a random neighbor instead.

   Another solution we accepted uses a simulated annealing-based modification to ID3. States, neighbors, and fitness are as in ID3. Only at each split of the decision tree, with some probability $1/T$ choose a random attribute to split on, instead of the one with highest information gain.

2. (5 pts) What are pros and cons to doing that, compared with using the greedy information gain approach?

   Some pros:

   - The key advantage is that this will result in a much more thorough search of the solution space, and possibly result in a better tree.
   - If one uses a penalty term in the fitness function, it may avoid overfitting.

   Some cons:

   - This takes a lot longer to run.
   - You still may get caught at local optima, and most of your solutions will be pretty bad.
   - It only works for categorical data– it'd be much more difficult to implement for real-valued data (a huge state space).

2 Artificial Neural Networks- 20 pts

1. (5 pts) In class we discussed that a single perceptron cannot compute XOR. Construct a network of *multiple* perceptrons that will accomplish this (draw a picture, and include weights).

   Here's one possible solution. Assume threshold activation function at each node with threshold 0.

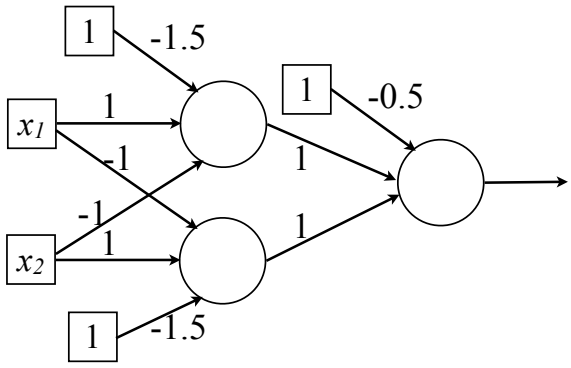

Figure 1: An XOR neural network

2. (5 pts) Now construct a network that will compute parity of four 0/1 inputs (that is, will output 1 if the sum is odd, or 0 if the sum is even).

   One solution is to use the XOR network from part 1 and "stack" them.

3. (5 pts) Suppose instead of using the notion of squared-error, as in lecture, we wanted to use $E = Err = |y - h_W(x)|$. Derive the perceptron learning rule. (Hint, you'll need to assume that error never becomes zero).

   To derive the update rule, we first take the gradient of the error function $E = |\text{Err}|$ with respect to the weights W. Here's the gradient with respect to one component, W_i. First,

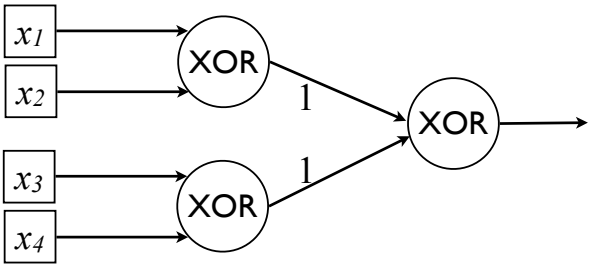

Figure 2: A neural network for parity

using a chain rule we can break it down to:

$$\begin{aligned}\frac{\partial E}{\partial W_i} &= \frac{\partial E}{\partial \text{Err}}\frac{\partial \text{Err}}{\partial W_i} \\ \frac{\partial E}{\partial W_i} &= \frac{\partial |\text{Err}|}{\partial \text{Err}}\frac{\partial \text{Err}}{\partial W_i}\end{aligned}$$

Assuming that Err is never 0, we have

$$\frac{\partial |\text{Err}|}{\partial \text{Err}} = \begin{cases} 1 \text{ if } \text{Err} > 0 \\ -1 \text{ if } \text{Err} < 0 \end{cases}$$

To avoid writing cases, we can rewrite the expression as

$$\frac{\partial |\text{Err}|}{\partial \text{Err}} = \frac{|\text{Err}|}{\text{Err}}.$$

So the overall gradient is:

$$\begin{aligned}\frac{\partial E}{\partial W_i} &= \frac{|\text{Err}|}{\text{Err}}\frac{\partial \text{Err}}{\partial W_i} \\ \frac{\partial E}{\partial W_i} &= -\frac{|\text{Err}|}{\text{Err}} g'(\text{in}) x_i\end{aligned}$$

where for the last step we just used the expression from the lecture slides for $\frac{\partial \text{Err}}{\partial W_i}$.

Given the gradient, we can update the weights by moving a little bit in the steepest decreasing direction, which is the opposite direction of the gradient. So the update rule is:

$$\begin{aligned} & W_i \leftarrow W_i - \alpha \frac{\partial E}{\partial W_i} \\ \Leftrightarrow \quad & W_i \leftarrow W_i + \alpha \frac{|\text{Err}|}{\text{Err}} g'(\text{in}) x_i\end{aligned}$$

for some $\alpha > 0$.

4. (5 pts) Why might we want to use a sigmoid function instead of a threshold function for activation?

   Sigmoid is differentiable, so we can use the update rule. It also might be more resilient to noise.

3 Linear Separators and SVMs- 10 pts

We will construct a support vector machine that computes the XOR function. It will be convenient to use values of 1 and -1 instead of 1 and 0 for the inputs/outputs. So a training example looks like ([-1, 1], 1) or ([-1, -1], -1).

1. (1 point) Draw the four examples using axes x_1, x_2, denoting class by +/-. Label them $(p_1, ..., p_4)$.

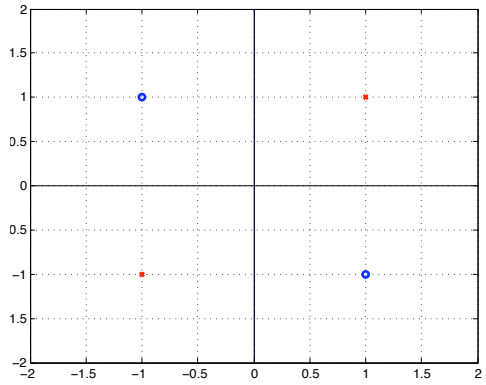

Figure 3: Plot of 4 points for xor where I forgot to label the axes: the horizontal axis is x_1 and the vertical is x_2.

2. (6 points) Propose a transformation that will make these points linearly separable (it can be done in 2D). Draw the four input points in this space, and the maximal margin separator. What is the margin?

   One transformation:

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= (x_1 - x_2)^2 \end{aligned}$$

   The margin is 2 (4 also accepted, since it wasn't clearly defined in lecture).

3. (3 points) Now draw the separating line back in the original Euclidean input space in part 1.

   For the transformation given in part 2, the max-margin separator is at $y_2 = 2$. So working backwards, we get:

$$\begin{aligned} & y_2 = 2 \\ \Leftrightarrow \quad & (x_1 - x_2)^2 = 2 \\ \Leftrightarrow \quad & x_1 - x_2 = \sqrt{2} \text{ or } x_1 - x_2 = -\sqrt{2} \end{aligned}$$

4 Pruning Decision Trees- 35 pts

For this question, if you can find a suitable implementation of the algorithms online, you are welcome to use them.

Because different implementations were used, answers varied. Plots were expected to have been generated from actual data, and not hand-drawn.

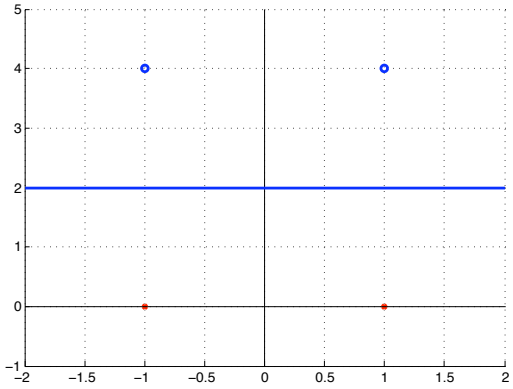

Figure 4: One transformation and the max-margin separator where I forgot to label the axes again: the horizontal axis is y_1 and the vertical is y_2.

1. (10 pts) Run ID3 to train a decision tree for the German Credit Approval data set, classifying whether someone is approved for credit. See data available on the webpage. As you are building the tree, record the error rates on both the training set and the test set (similar to Slide 18 in the second Decision Tree lecture). How many nodes are in the resulting tree?

   You would most likely get around 750 nodes.

2. (10 pts) Now run reduced-error pruning based on the validation set. Record the error rate for the validation set as the tree is being pruned, with the plot as in the previous part. How many nodes are in the resulting tree?

   You would most likely get around 50-100 nodes.

3. (10 pts) Now use the statistical significance testing method to prune the tree. Plot the accuracy for different thresholds of significance. How many nodes are in the resulting tree?

   For the "resulting" tree we accepted either the tree with the threshold with the best accuracy, or the smallest tree. You would also most likely get around 50-100 nodes.

4. (5 pts) Interpret your plots. How does pruning affect results on the training and test error? What pruning method works best, and why do you think that is? If we picked out a new test set to measure accuracy, which pruning method would have better results?

   Typically reduced-error pruning achieved up to 80% accuracy, while significance testing achieved between 65-80%. RE pruning probably did better because you're essentially *fitting on the test set*, while SS is ignorant of the test data and therefore a little more immune to overfitting on test sets. For this reason, the resulting tree from the SS data would perform better than the resulting tree from RE on the test set we used earlier. (Also accepted: if you assumed that we were fitting RE on the new test set, then saying that RE does better was fine.)

5 Clustering- 30 pts

For this question, implement the algorithms yourself and submit code in the AFS space.

Sometimes instead of treating data as clumps of points, we may want to assume they have a certain distribution. *Gaussian mixture models* are one example of these.

For example, suppose that for some reason you wanted to classify a population of students as male or female. We know that men are *generally* taller than women (with each population having approximately a Gaussian/normal distribution), but there's a lot of overlap. We also know that the male:female at CMU is 3:2, so if you have a data point of middling height, you're probably safer guessing that the student is male. Gaussian mixture models try to take this into account, by giving a probabilistic assignment to membership, rather than deterministic as in K-means.

The way one can go about solving this is to use what's called the *EM Algorithm*. For Gaussian mixture models, the EM algorithm is similar to K-means, only in addition to calculating means (centers) for each cluster, you also use"mixing" variables a_i to reflect the overall distribution of the classes (a 3:2 ratio would have $(a_0, a_1) = (.6, .4)$. Based on these parameters, each point will be assigned to a cluster with some *probability* of membership to each class. After initialization of μ_i and a_i for $i = (0, 1)$, you iterate through the following two steps until convergence:

First, in the expectation ("E") step, you estimate an expectation value for the "membership" m_{ij} of each point x_j for distribution Y_i.

$$m_{ij} = \frac{a_i f_{Y_i}(x_j; \mu_i, \sigma_i)}{\sum_{k=1}^{K} a_k f_{Y}{}_k(x_j; \mu_k, \sigma_k)}$$

where f is the formula for the normal distribution. Then, based on the membership values assigned, you re-adjust your estimates of the mean and variance of each Gaussian to maximize the likelihood ("M-step"), as well as the mixing coefficients:

Each mixing coefficient is the mean of the membership values for all points.

$$a_i = \frac{1}{N} \sum_{j=1}^{N} m_{ij}$$

The mean of each Gaussian is a *weighted* average of all points with its membership value.

$$\mu_i = \frac{\sum_j m_{ij} x_j}{\sum_j m_{ij}}$$

You can derive an update for variance, but for this problem we'll cheat and assume $\sigma = 1$.

1. (15 points) Implement EM as above for two 1-D Gaussian mixtures, and run on the GMM data set available on the webpage. Do several runs, each time randomizing your initial settings for μ and a (let's say pick some μ between (-10, 10)). Show the distribution of the end estimates for μ and a over all the initial settings. What is the average accuracy? What is the maximum accuracy attained by any single run?

   In general, EM can get stuck at local optima (which is why we initally intended to see the distributions of results). Most people's implementations did not run into this problem, however, if you did occasionally that's to be expected. The most common convergence point was around $(-0.007, 2.482)$, very close to the generating parameters of $(0, 2.5)$. Implementations that had no issue with local optima tended to have equivalent values for average and max accuracy, others varied.

   Accuracy is measured by comparing the resulting class assignments with the classes provided in the gmmclasses.txt file. You can measure accuracy in one of two ways: 1) for each point, if its "membership" of one class was higher than the other, automatically assign it to that class.

2) For each point, probabilistically assign membership (flip a biased coin, where the bias is the membership probability of one class). Since this is an unsupervised learning problem, accuracy is really only in *differentiating* clusters, not whether or not the class labels are correct. So, since sometimes the class labels get flipped (depending on which μ value was lower initially), you can occasionally get really bad accuracy even if you're doing it right. In the ideal implementation you could simply take $accuracy = max(accuracy, 1 - accuracy)$, since $1 - accuracy$ is what you'd get if you flipped the labels.

Note that it's impossible to get 100% accuracy on this data, because of the overlap. This was one source of missed points. Also, a 70% accuracy usually indicated that you were labeling all as one class (since 70% of the data came from one class). Points were also taken off if EM never converged to anything close to accurate, since that indicates a bug.

2. (10 points) Implement K-means (one dimensional, for K=2) and repeat, also using several runs with random initialization. Show the distribution for the end estimates of the centers over all initial settings. What is the average accuracy? What is the maximum accuracy attained by any single run?

   Usually the best that k-means did was around 88% accuracy. There was generallyeven less problem with local maxima, mainly because of the low dimensionality of the parameters. (EM has to fit both the as and the μs.) Generally the centers chosen were around $(-0.21, 2.37)$.

3. (5 points) How do your results compare? Why do you think one performs better than the other? In which cases would the other perform better?

   EM usually performed better. Reasons for this are because the data are distributed in such a way that EM for Gaussians is suited. The data classes aren't very even (70-30 split), and the data was generated using two different Gaussians. If the data were generated from a *uniform random* distribution, k-means might be better. If the data were linearly separable, k-means might perform better.

   Note that EM is not limited to Gaussians, or even to clustering– EM is an algorithm that can be used for a wide range of problems.