CPSC 340 Assignment 6 (due December 4)

Multi-Class Logistic, Label Propagation with Random Walks

- You can work in groups on the assignments. However, please hand in your own assignments and state the group members that you worked (as well as other sources of help like online material).
- Place your name and student number on the first page, and submit all answers as a single PDF file to handin.
- For questions that ask for code, you should include the relevant parts of the code in the appropriate place in the PDF file.
- Please organize your submission sequentially according to the sections used in this document.
- All Sections (1-5) are equally weighted.
- There may be updates/clarifications to the assignment after the first version is put online. Any modifications will be marked in red.

1 Multi-Class Logistic

The function *example_multiClass* loads a multi-class classification dataset and fits a 'one-vs-all' classification model using least squares, then reports the validation error and shows a plot of the data/classifier. The performance on the validation set is ok, but could be much better. For example, this classifier never even predicts that examples will be in classes 1 or 5).

1.1 Linear Log-Odds Model

A very common model for binary classification, where $y_i \in \{-1, 1\}$, is a *log-linear* model. The loss function used in these models is based on the 'odds ratio',

$$\frac{p(y_i = +1 | w^T x_i)}{p(y_i = -1 | w^T x_i)},$$

the probability of the model predicting the correct class divided by the probability of the model predicting the incorrect class (a value greater than one indicates that it makes the right prediction). Consider a model where we assume that the logarithm of the odds ratio is a linear function,

$$\log\left(\frac{p(y_i = +1 | w^T x_i)}{p(y_i = -1 | w^T x_i)}\right) = w^T x_i. \tag{1}$$

As discussed in class, when we have a probability p for the correct label y_i given our prediction, we can construct a loss function out of it and fit the model by minimizing the negative logarithm of the probability summed across all examples,

$$\underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{n} -\log(p(y_i | w^T x_i)), \tag{2}$$

where y_i is now the label given in the training set. Derive the form of $p(y_i = +1|w^T x_i)$ and $p(y_i = -1|w^T x_i)$ implied by the assumption (1), and use this to simplify the loss function (2) as much as possible.

Hint: First derive the form of $p(y_i = +1|w^T x_i)$ in (1) using that probabilities sum to one.

Answer:
Exponentiated both sides we get

$$\frac{p(y_i = +1|w^T x_i)}{p(y_i = -1|w^T x_i)} = \exp(w^T x_i),$$

or equivalently that

$$p(y_i = +1|w^T x_i) = \exp(w^T x_i) p(y_i = -1|w^T x_i).$$

Use that probabilities sum to one to get

$$p(y_i = +1|w^T x_i) = \exp(w^T x_i)(1 - p(y_i = +1|w^T x_i)).$$

Add $\exp(w^T x)i)p(y_i|w^T x_i)$,

$$(1 + \exp(w^T x_i))p(y_i = 1|w^T x_i) = \exp(w^T x),$$

or that

$$p(y_i = +1|w^T x_i) = \frac{\exp(w^T x_i)}{1 + \exp(w^T x_i)},$$

and multiply top and bottom by $\exp(-w^T x_i)$ to get

$$p(y_i = +1|w^T x_i) = \frac{1}{1 + \exp(-w^T x_i)},$$

which is the sigmoid function. Use that probabilities sum to one to get

$$\begin{aligned} p(y_i = -1|w^T x_i) &= 1 - p(y_i = +1|w^T x_i) \\ &= 1 - \frac{1}{1 + \exp(-w^T x_i)} \\ &= \frac{1 - 1 + \exp(-w^T x_i)}{1 + \exp(-w^T x_i)} \\ &= \frac{1}{1 + \exp(w^T x_i)}. \end{aligned}$$

Given the true label y_i, we can write both cases as

$$p(y_i|w^T x_i) = \frac{1}{1 + \exp(-y_i w^T x_i)}.$$

Taking the logarithm gives us

$$\log p(y_i|w^T x_i) = -\log(1 + \exp(-y_i w^T x_i)).$$

Plugging this into (2) gives us

$$\underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)),$$

which we recognize as the logistic regression model.

1.2 One-vs-all Logistic Regression

Using the squared error on this problem hurts performance because it has 'bad errors' (the model gets penalized if classifies examples 'too correctly'). Write a new function, *logLinearClassifier*, that replaces the squared loss in the one-vs-all model with the logistic loss. Hand in the code and report the validation error.

Answer:
The loop should should changed to something like this:

```
W = zeros(d,k); % Each column is a classifier
for c = 1:k
    yc = ones(n,1); % Treat class 'c' as (+1)
    yc(y ~= c) = -1; % Treat other classes as (-1)
    W(:,c) = findMin(@logisticLoss,W(:,c),500,1,X,yc);
end
```

The above uses *findMin* and *logisticLoss* as in previous assignments. This decreases the error from around 0.13 down to around 0.07.

1.3 Softmax Loss

Using a one-vs-all classifier hurts performance because the classifiers are fit independently, so there is no attempt to calibrate the columns of the matrix W. An alternative to this independent model is to use the softmax probability,

$$p(y_i|W,x_i) = \frac{\exp(w_{y_i}^T x_i)}{\sum_{c'=1}^k \exp(w_c'^T x_i)}.$$

Here c' is a possible label and $w_{c'}$ is column c' of W. Similarly, y_i is the training label, w_{y_i} is column y_i of W, and in this setting we are assuming a discrete label $y_i \in \{1,2,3,4,5\}$. Derive the loss function that results from using this probability (i.e., the negative logarithm of the probability), as well as the derivative of the loss with respect to a particular element W_{cj}. Try to simplify the derivative as much as possible.

Hint: for the gradient you can use $(x_i)_j$ to refer to element j and you can use an 'indicator' function, $I(y_i = c)$, which is 1 when $y_i = c$ and is 0 otherwise. Note that you can use the definition of the softmax probability to simplify the derivative.

Answer:
The loss is the negative logarithm of the probability,

$$-\log(p(y_i|W,x_i) = -w_{y_i}^T x_i + \log\left(\sum_{c'=1}^k \exp(w_{c'}^T x_i)\right).$$

The derivative with respect to a particular W_{ij} is given by

$$\frac{d}{dW_{cj}}[-\log(p(y_i|W,x_i)] = -I(y_i = c)(x_i)_j + \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^k \exp(w_{c'}^T x_i)}(x_i)_j.$$

This can be simplified even further by factoring out $(x_i)_j$ and noticing that the softmax probability appears in the second term,

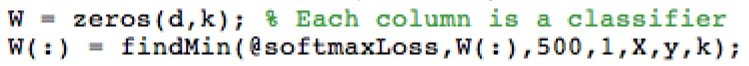
$$\frac{d}{dW_{cj}}[-\log(p(y_i|W,x_i)] = (x_i)_j[p(y_i|W,x_i) - I(y_i = c)].$$

1.4 Softmax Classifier

Make a new function, *softmaxClassifier*, which fits W using the softmax loss from the previous section instead of fitting k independent classifiers. Hand in the code and report the validation error.

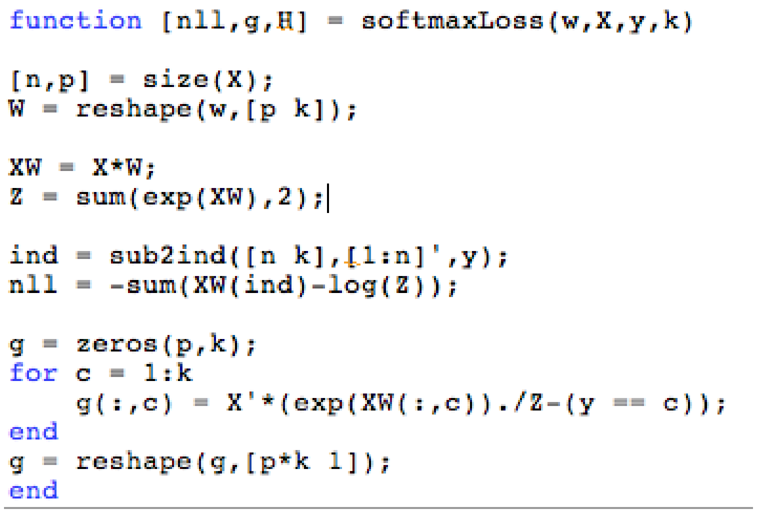
HInt: you may want to use the *autoGrad* function from A3 to check that your gradient code is correct.

Answer:
The loop should be replaced by something that looks like this:



```
W = zeros(d,k); % Each column is a classifier
W(:) = findMin(@softmaxLoss,W(:),500,1,X,y,k);
```

The *softmaxLoss* function should roughly look like this:



```
function [nll,g,H] = softmaxLoss(w,X,y,k)

[n,p] = size(X);
W = reshape(w,[p k]);

XW = X*W;
Z = sum(exp(XW),2);

ind = sub2ind([n k],[1:n]',y);
nll = -sum(XW(ind)-log(Z));

g = zeros(p,k);
for c = 1:k
    g(:,c) = X'*(exp(XW(:,c))./Z-(y == c));
end
g = reshape(g,[p*k 1]);
end
```

(Longer versions that use a 'for' loop instead of *sub2ind* are ok.) The validation error depends on how precisely you solve the optimization problem, but it decrease to something like 0.02.

1.5 Cost of Multinomial Logistic Regression

Assuming that we have

1. n training examples.
2. d features.
3. k classes.
4. t testing examples.
5. T iterations of gradient descent for training.

In $O()$ notation, what is the cost of training the softmax classifier? What is the cost of classifying the test examples?

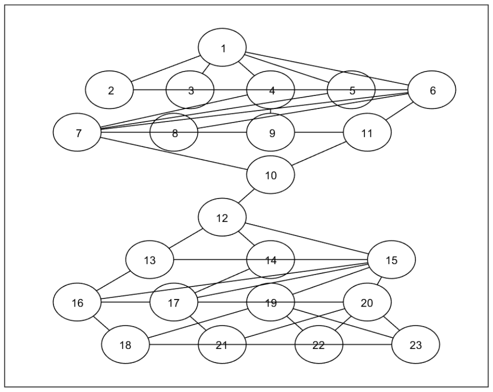
Answer:
Training the model involves T iterations of gradient descent. Each iteration involves computing the function value and gradient over all n examples. To evaluate the function for one example, the dominant cost is compute $w_{c'}^T x_i$ for all k values of c', each of which costs $O(d)$. Thus evaluating the function for one example costs $O(dk)$, and the gradient has the same cost. Putting everything together gives $O(ndkT)$.
At test time the largest cost is computing $\hat{X}W$. Taking into account the dimensions of these matrices gives $O(tdk)$.

(Technically, we could do some of these operations slightly faster using fast matrix multiplication methods.)

2 Label Propagation with Random Walks

In class we discussed a graph-based semi-supervised learning method. While its possible to fit this model using gradient descent, or by propagating probabilities through the graph, in this question we'll consider an interpretation of the model in terms of random walks on a graph.

The function *example_randomWalk* loads a dataset containing the adjacency matrix (A) for the following graph which was used as an example in class:



It also gives the *labelList*, a matrix where the first column contains node numbers and the second column contains class labels. In this case, we are told that node 2 has the label -1 and node 18 has the label $+1$. However, we do not have any features for any of these objects. Our job in this transductive learning setting is to label all the nodes in the graph. The labels are obtained by, starting from each node, running a random walk until we 'stop' at a label of $+1$ or -1 (the stopping criterion is described below).

By averaging over the result returned by multiple runs, we can get a probability that a random walk starting from the node will stop in $+1$ or -1. Subsequently, you can use the label with the highest probability to label the node (or perhaps leave them unlabeled if no label has a very high probability).

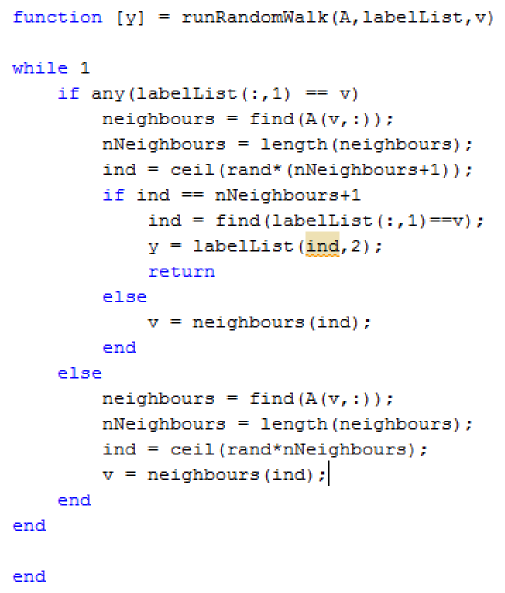
The random walk starts from the node we want to label, and calls this 'v'. Then, at each iteration we update the value 'v' to another node (or we decide to stop) as follows:

1. If v is *not* one of the labeled nodes, randomly set v to one of its neighbours in the graph (with each neighbour getting equal probability).
2. If v is one of the labeled nodes and has d_v neighbours, then return the label of the node with probability $1/(d_v + 1)$. Otherwise, randomly set v to one of its neighbours in the graph (with each neighbours again getting equal probability).

(The reason we don't immediately stop when we reach a labeled node is to account for cases where we've labeled nodes with a ton of neighbours. These high-degree 'hub' nodes are likely reachable from a variety of other nodes, so their label doesn't give strong evidence that its neighbours should receive a particular label.)

Write a function *runRandomWalk* that implements the above random walk, taking the adjacency matrix along with the labelList and a starting node as an input, and returning either $+1$ or -1 as described above. Hand in the function and report the probabilities of each class obtaining each label using this method.

Answer:
The code could roughly look like this:



```
function [y] = runRandomWalk(A,labelList,v)

while 1
    if any(labelList(:,1) == v)
        neighbours = find(A(v,:));
        nNeighbours = length(neighbours);
        ind = ceil(rand*(nNeighbours+1));
        if ind == nNeighbours+1
            ind = find(labelList(:,1)==v);
            y = labelList(ind,2);
            return
        else
            v = neighbours(ind);
        end
    else
        neighbours = find(A(v,:));
        nNeighbours = length(neighbours);
        ind = ceil(rand*nNeighbours);
        v = neighbours(ind);
    end
end

end
```