CMPS 242 Fourth Homework, Fall 2019

5 Problems. 17 pts, due 11:50pm Monday November 25

This homework is to be done in groups of 2. Each group members should completely understand the group's solutions and *must* acknowledge all sources of inspiration, techniques, and/or helpful ideas (web, people, books, etc.) other than the instructor, TA, and class text. Each group should sign up on Canvas and electronically submit a single set of solutions for their group. See the Piazza post **New method for submitting homework** for more details.

Your solutions should be clearly numbered and in order. Although there are no explicit points for "neatness", the TA may deduct points for illegible or poorly organized solutions.

I *strongly* recommend that each student work on all of the problems individually before integrating their solutions into the group consensus (although the programming problems might be done as "pair programming"). Dividing up the problems so each student does just a subset of them is the wrong approach.

Do this first: Download the Pima Indian Diabetes dataset. It is available from Kaggle: `https://www.kaggle.com/uciml/pima-indians-diabetes-database`. Load the data into your system (python recommended, as you will want to use scikit-learn for some of the problems below) representing the examples with numpy vectors. Do a random stratified partition of the examples into a training set (80%) and test set (20%). The stratified partition keeps the class proportions as close as possible in the two sets (i.e. put 20% of the "1" labeled examples in the test set, and 20% of the "0" labeled examples in the test set). Create a second version of the dataset where the features are normalized (shifted and rescaled) to lie in the interval [0,1]. Read and understand the python's time library so that you can time experiments (using wall-clock time).

1. (1 pt) Was this homework done in a group of 2? (Hint: this is to encourage group formation)

2. (7 pts total) Implement stochastic gradient descent for logistic regression (see equation 4.91, but update for each example in turn rather than summing the gradients over all examples) and do the following.

   (a) (3 pts) Apply your SGD algorithm to the un-normalized training set. Keep track of the log-likelihood of the training set, and run until the log-likelihood seems to be converging. Experiment a little bit with the step-size. What step sizes seem to lead to faster convergence? How long does the convergence take (measured in wall-clock time and epochs). What is your error rates and *average* log-likelihood on the training data and on the test data? (The average log-likelihood is the log-likelihood divided by the number of examples.) What features are given the highest positive weights? Which features are given the largest negative weights?

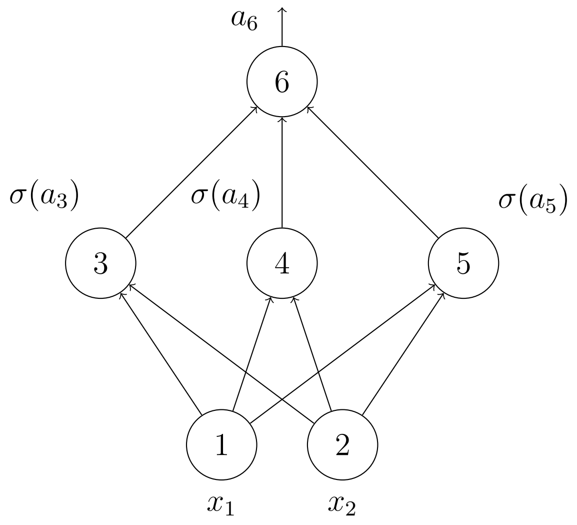
   (b) (3 pts) Apply your SGD algorithm to the normalized training set. You can use a good step-size from the previous part. Report the log-likelihoods and error rates on the training and test sets. Are the learned weights similar? Are the learned hypotheses similar after taking into account the rescaling?

(c) (1 pt, conceptual) Logistic regression creates linear threshold hypotheses. What kind of data transformations could be used to increase the power/flexibility of the hypotheses produced by logistic regression? *Although not required*, interested students might want to experiment with the effects of applying the transformations they suggest on the diabetes dataset.

3. (3 points) First apply a decision tree classifier (we recommend the one with scikit learn) on the training set. Experiment with different maximum depths, and report their error rates on the training and test data. Also report the training times required. Should the training or test set accuracies be the same on the unnormalized data as the normalized data? Why or why not? Next, apply a random forrest learner (like `sklearn.ensemble.RandomForrestClassifier`) to the training data. Try a few different numbers of trees (perhaps 5, 20, and 100). Report the training and test accuracies of your forests.

4. (4 points, open ended) Scikit learn has a `neural_network.MLPClassifier` module, use that or something similar to train up a neural network on your normalized training set. Experiment a bit with the number of hidden layers (say 1-4) and number of nodes on each layer (say 10 to 100). Report the training time and accuracies on the training set and test set. Neural network packages tend to have many tunable parameters. Explore the effects fo them on the running time and goodness of the produced hypothesis. Some of the more interesting candidates for exploration might be momentum, solver, and alpha (the L2 penalty parameter).

5. (2 points) Hand-simulate backprop. Consider the following artificial neural network.

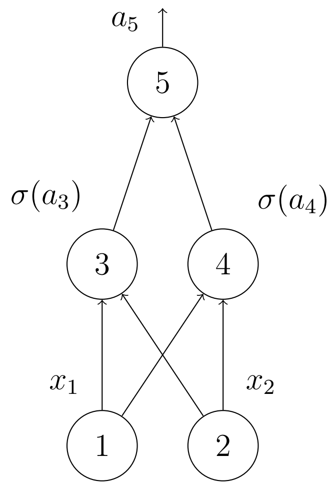


Let the $\sigma()$ function at nodes 3,4, and 5 be the ReLU (rectified linear unit), $\sigma(a) = \max(0, a)$. Assume that the the nodes do *not* have bias terms, and the initial weights are:

$$
\begin{aligned}
(w_{3,1}, w_{3,2}) &= (1, 1)\\
(w_{4,1}, w_{4,2}) &= (1, -1)\\
(w_{5,1}, w_{5,2}) &= (-1, -1)\\
(w_{6,3}, w_{6,4}, w_{6,5}) &= (1, 1, 1)
\end{aligned}
$$

(recall that weights are conventionally indexed with the "to" node first and the "from" node second). The error on the output is the squared error, $\frac{1}{2}(a_6 - t)^2$, so $z_6 = a_6$ and there is no non-linearity at the output node. Under these assumptions, perform one step of backpropagation with step size $\eta = 0.1$ on the training example $x_1 = 1$, $x_2 = 2$, $t = 2$. Show the a_i, z_i and δ_i values for each non-input node, and the new weights after the backprop update. See the backpropagation handout for the procedure to use.

Bonus problem(not to be turned in): Hand-simulate backprop on the following artificial neural network with sigmoid activations.



Let the $\sigma()$ function be the logistic sigmoid, $\sigma(a) = 1/(1 + e^{-a})$. Assume that the the nodes do *not* have bias terms, and the initial weights are all 0's, the error on the output is the squared error, $\frac{1}{2}(a_5 - t)^2$, so $z_5 = a_5$ and there is no sigma-function at the output node. Under these assumptions, perform one step of backpropagation with step size $\eta = 0.1$ on the training example $x_1 = 1$, $x_2 = 2$, $t = 1$. Show the a_i, z_i and δ_i values for each non-input node, and the new weights after the backprop update. See the backpropagation handout for the procedure to use.

Bonus problem (not to be turned in): Do an MNIST deep learning tutorial, like:
`https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-`

54c35b77a38d
or
https://www.digitalocean.com/community/tutorials/how-to-build-a-neural-network-to-recognize-handwritten-digits-with-tensorflow
or
https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627
or one or more of the many others (google "deep learning tutorial MNIST"). Most of these are for one of the primary neural network packages, like Keras, pytorch, or tensorflow.

Bonus problem (not to be turned in): Perceptron algorithm with noise experiment. Implement the Perceptron algorithm presented in class, where $\mathbf{w}_{\text{new}} := \mathbf{w}_{\text{old}} + t\mathbf{x}$ when the algorithm makes a mistake for examples with 15 features. You may use any appropriate programming language/system you are familiar with. Create a set of 300 instances where the instances are selected uniformly from the boolean hyper-cube $\{\pm 1\}^{15}$ (i.e. each feature is $+1$ with probability 1/2 and -1 with probability 1/2).

1. Label your 300 instances with the first component of the instances, so $t_i = x_{i,1}$ to create a labeled sample. Report how many epochs (iterations through the training set) and prediction errors occur before producing a hypothesis that correctly labels the training set.

2. Re-label the 300 instances training set so that $t_i = +1$ if at least half the features in $\mathbf{x}_i$ are $+1$ (and $t_1 = -1$ if at least half the features are -1. Run the perceptron algorithm on the modified training set and report how many epochs and prediction errors occur before producing a hypothesis that correctly labels the training set. (Would you expect this to take more or fewer epochs than the first part? Why?)

3. Create a third training set with label noise from the 300 instances. Set each label t_i to $\text{sign}(r + \sum_{j=1}^{11} x_{i,j})$ where r is a random number uniformly distributed between -4 and 4 (note that the sum ranges over only the first 11 features). Run three epochs through your training set, shuffling the order of the training set between epochs and keeping track of the 900 $\mathbf{w}$ vectors produced by the algorithm, call these vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{900}$ (note that if $\mathbf{w}_j$ will be the same as $\mathbf{w}_{j+1}$ if $\mathbf{w}_j$ correctly predicts the jth instance's label). Consider creating several hypotheses created from the three-epoch run. Many of these are intended to reduce the effect of the noisy labels.

   (a) The *last hypothesis* predicts with $\text{sign}(\mathbf{w}_{900} \cdot \mathbf{x})$, using the last weight vector from the run.

   (b) The *average hypothesis* computes $\bar{\mathbf{w}} = \sum_{j=1}^{900} \mathbf{w}_j / 900$ and predicts with $\text{sign}(\bar{\mathbf{w}} \cdot \mathbf{x})$.

   (c) The *voted hypothesis* votes all 900 weight vectors, predicting with (if the vote is tied, treat it as an incorrect prediction).

   (d) The *last epoch average* computes $\bar{\mathbf{w}}_\ell = \sum_{j=601}^{900} \mathbf{w}_j / 300$ and predicts with $\text{sign}(\bar{\mathbf{w}}_\ell \cdot \mathbf{x})$.

   (e) The *last epoch vote* predicts with the vote $\text{sign}\left(\sum_{i=601}^{900} \text{sign}(\mathbf{w}_i \cdot \mathbf{x})\right)$

   Which of these do you think will be more accurate on new unseen data? Create a separate test set of 500 examples (with the random noise) and evaluate the accuracy of these five hypotheses on it. Ambitious students may run this experiment multiple times (say 10) and report the average accuracies.