-CSE 242 Homework 1-

Vhal Purohit vpurohit@ucsc.edu,
Kejun Chen kchen158@ucsc.edu
October 2019

Problem 2)

Implement stochastic gradient descent for logistic regression (see equation4.91, but update for each example in turn rather than summing the gradients over all examples) and do the following.

1. 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?

   First, we need to clarify some definitions in the following solution. First, the probability expression, logistic sigmoid function, the log-likelihood and loss function. t_n is the label, y_n is the output also denoted $h\_w(x)$, is a non linear function, w is the weights parameter,x is the data feature:

$$h\_w(x) = y\_n = \frac{1}{1+\exp(-w \cdot x)} \tag{1}$$

$$P(0|x;w) = 1 - h\_w(x) \tag{2}$$

$$\log p(t|w) = -\sum \_n = 1^N \{t\_n \log y\_n + (1-t\_n)\log(1-y\_n)\} \tag{3}$$

$$E(w) = -\log(p(t|w) \tag{4}$$

   Using SGD, we need to know gradient of error function, but note that if we use stochastic gradient update, we don't need sum all the data points, we can just take one (x_i, t_i) each step, so we can also get the derivatives:

$$\frac{\partial E}{\partial w} = \sum \_n = 1^N (y\_n - t\_n) x\_n \tag{5}$$

$$\frac{\partial E}{\partial w\_j} = (y\_i - t\_i) x\_i, j \tag{6}$$

   Then we can get the updated weights:

$$w\_j = w\_j - \eta \frac{\partial E}{\partial w\_j} \tag{7}$$

$$= w\_j - \eta (y\_i - t\_i) x\_i, j \tag{8}$$

   The accuracy definition is obvious in logistic regression, when the output probability is bigger than 0.5, predict 1, else 0. Compared with the label, we can know whether it is right or wrong. The error rate is the number predicted right / the total number of data size. In practice, we compare the difference $abs(y\_n - t\_n)$, if the difference is bigger than 0.5, the prediction is right otherwise it is wrong.

   There are a lot of methods to judge whether the solution is converged. We choose the range of error function is small enough, to be specific, in every epoch, we learned an new updated weight and then calculate its average loss function (loss function shown (4) / the number of data set N), we check the difference between the new average loss function and last time's average loss function, until the difference is less than ϵ, we found using different ϵ, the iteration times are different, in the last, we use a suitable $\epsilon = 0.00001$.

   As for the learning rate η, one of the most important parameter in our test, which will greatly effect our iterations time, accuracy, so we have tried different learning rate and found the suitable one is 0.00001

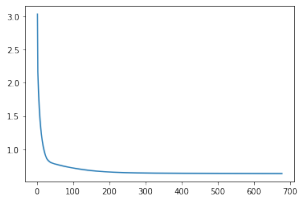
   The main equations have been shown before, here are some small tips in our codes. We consider bias so there are 8+1 features now. I normalize the data set first, and then add one column 1 in the data set to achieve the bias. Our

outside loop is each epoch, and it will ended until the converge condition is satisfied. The inner loop is update weights traversing the entire list, but we don't calculate the error for each update, we just keep updating new weights. After traversing the entire list each time, we get the final weights for this traverse, and then we calculate the average loss for the whole training set and record it for us to judge whether it is converged. Besides, we also record the accuracy for the whole training set. So during each epoch, we can get the accuracy and average loss, the loop will end when the difference of average loss is small enough. Last, we traverse the whole list from the first data point to the last data point, because when I use 'shuffle' or 'random' function to add randomness, the average loss curve will up ans down so I delete the randomness in my code.

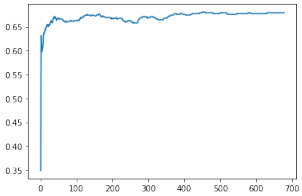
Answer: When the learning rate is large and ϵ is small, it will satisfy convergence requirement soon, but it is suitable in practice. We should set a suitable ϵ to make the accuracy and loss stable, in other words, varies little. When the learning rate is $0.01, 0.001$ it doesn't converge, while 0.000001 or less it takes longer step. I calculate the time cost for each epoch, the average is around 0.01626s, and it takes 678 epochs to converge. One possible weights (the first one is bias) result is: [[0.62971935] [0.12418549] [0.0088641] [-0.03182058] [-0.00321401] [0.00259271] [-0.01794743] [0.53151222] [-0.00941804]]

In this situation, in the training set, the log-likelihood is -390.9387 so the average log-likelihood is -0.6367 and the error rate is 0.3218. In the test set, the log-likelihood is -99.3182 so the average log-likelihood is -0.6449 and the error rate is 0.3701. Ignore the bias, the highest positive weights is Diabetes Pedigree Function and the largest negative weights is Blood Pressure. But I have to address that the weights varies a lot during different simulation, but the accuracy and loss value are more stable.

Average loss



Average accuracy



2. 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?

   Using Normalization function in python to normalize feature, but the label is still the same. Then we find we cannot use the same η as the un-normalized data because the normalized data should converge faster than un-normalized data. I calculate the time cost for each epoch, the average is around 0.0157s, and it takes 513 epochs to converge. In this case, we choose another parameters $\eta = 0.001$, then we get one possible weight: [[-0.03546145] [0.7471336] [0.64598564] [-2.49292033] [-0.54588267] [0.0256517] [-0.20091524] [0.24859705] [0.1227684]]

   In this situation, in the training set, the log-likelihood is -379.3645 so the average log-likelihood is -0.6178 and the error rate is 0.3502. In the test set, the log-likelihood is -95.1544 so the average log-likelihood is -0.6179 and the error rate is 0.3636. The highest positive weights is Pregnancies and the largest negative weights is Blood Pressure. But I have to address that the weights varies a lot during different simulation, but the accuracy and loss value are more stable.

   Compared with the un-normalized model, it is hard to say the weights are similar or so, but according to the accuracy and loss, it is quite similar. So it is reasonable to say that logistic regression relies less on the normalized or un-normalize. But is obvious that it matters the convergence a lot, to be specific normalized data contributes to convergence soon.

3. 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 data set.

There are a lot common transformation methods including normalization we have shown before. I have tried Sqrt Root, MinMaxScaler transformation, but I found none of them improved the accuracy, in fact, the results make sense as I explained before, these transformation won't affect Logistic regression accuracy much considering linear threshold. Then I tried Principal component analysis to reduce the dimension of the raw data set first, and then applied logistic regression. The reason why I use PCA is because the raw data feature can be linear relative, but I found the accuracy is still similar. So I calculate the co-variance matrix and VIF in Matlab and found the results makes senses because the VIF is almost equal to 1, which means the data isn't show multicollinearity. It seems different model has different suitable η, so I have tried different η in each model, and only list the most suitable η in the form. For PCA method, I also tried different principal components and found 8 is the most suitable one. The error results of different transformation with un-normalized data:

method	train	test	time per epoch	epoch	η
sqrt	0.34202	0.3571	0.0158	978	0.00001
MinMaxScaler	0.3306	0.3766	0.0159	2063	0.0001
PCA	0.2475	0.25324	0.01559	747	0.000001

Problem 3)

First apply a decision tree classier (we recommend the one with scikit learn) on the training set. Experiment with dierent 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 dierent numbers of trees (perhaps 5, 20, and 100). Report the training and test accuracies of your forests.

To implement Decision Trees and Random Forests, we use scikit learn's inbuilt modules: sklearn.tree.DecisionTreeClassifier and sklearn.ensemble. respectively. We train Decision Trees on values of maximum tree depth ranging from 1 to 14, and find that the best results are obtained around max_depth=3. The quality of the split was determined using Gini impurity. The results can be seen in the table below

max_depth	train_acc	test_acc	time_taken
1	0.749	0.682	0.011
2	0.749	0.688	0.0035
3	0.789	0.740	0.0038
4	0.816	0.740	0.0038
5	0.847	0.7	0.0044
6	0.879	0.746	0.0047
7	0.92	0.72	0.0049
8	0.96	0.746	0.0047
9	0.98	0.714	0.005
10	0.992	0.740	0.005
11	0.993	0.733	0.0058
12	0.996	0.740	0.0056
13	0.998	0.714	0.0053
14	1.0	0.734	0.0050

For Random Forests, we tried maximum tree depths of 2, 4, 8 and 14; each having 5, 20 and 100 trees for their forests respectively. These are our findings:

num_trees	max_depth	train_acc	test_acc
5	2	0.74	0.69
5	4	0.80	0.72
5	8	0.91	0.727
5	14	0.96	0.694
20	2	0.75	0.72
20	4	0.83	0.74
20	8	0.94	0.74
20	14	0.99	0.73
100	2	0.77	0.71
100	4	0.84	0.75
100	8	0.96	0.72
100	14	1.0	0.74

Problem 4)

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 for 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).

We use LBGFS solver, using an alpha(regularization parameter for the network) value of 0.0001. We next train the training set using these hyper parameters and get the following results:

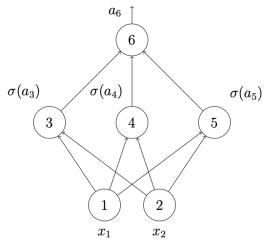
the first column shows the time taken, the second the training accuracy and the third test accuracy. I did 4 * 10 simulations and the best one is achieved by 2 layer, each layer is 50, the accuracy is 0.753:

size	time	training accuracy	test accuracy
[10]	0.083705	0.786645	0.733766
[20]	0.067669	0.788274	0.740260
[30]	0.095760	0.789902	0.740260
[40]	0.010804	0.703583	0.668831
[50]	0.079415	0.789902	0.740260
[60]	0.115588	0.786645	0.753247
[70]	0.154232	0.794788	0.753247
[80]	0.091084	0.781759	0.733766
[90]	0.045992	0.781759	0.733766
[100]	0.149636	0.793160	0.740260
[10, 10]	0.042440	0.791531	0.733766
[20, 20]	0.014097	0.703583	0.707792
[30, 30]	0.024330	0.688925	0.688312
[40, 40]	0.015375	0.649837	0.649351
[50, 50]	0.242269	0.786645	0.753247
[60, 60]	0.025031	0.664495	0.662338
[70, 70]	0.039829	0.705212	0.681818
[80, 80]	0.155806	0.775244	0.727273
[90, 90]	0.362785	0.781759	0.740260

[100, 100]	0.049315	0.679153	0.662338
[10, 10, 10]	0.132013	0.793160	0.740260
[20, 20, 20]	0.165162	0.773616	0.727273
[30, 30, 30]	0.017820	0.651466	0.649351
[40, 40, 40]	0.249210	0.785016	0.740260
[50, 50, 50]	0.037060	0.700326	0.668831
[60, 60, 60]	0.359446	0.789902	0.727273
[70, 70, 70]	0.443694	0.775244	0.733766
[80, 80, 80]	0.498756	0.771987	0.720779
[90, 90, 90]	0.611042	0.788274	0.733766
[100, 100, 100]	0.120086	0.706840	0.675325
[10, 10, 10, 10]	0.164896	0.796417	0.733766
[20, 20, 20, 20]	0.209865	0.780130	0.746753
[30, 30, 30, 30]	0.280150	0.789902	0.740260
[40, 40, 40, 40]	0.062025	0.697068	0.681818
[50, 50, 50, 50]	0.072706	0.679153	0.668831
[60, 60, 60, 60]	0.205725	0.719870	0.688312
[70, 70, 70, 70]	0.100941	0.708469	0.681818
[80, 80, 80, 80]	0.634621	0.768730	0.733766
[90, 90, 90, 90]	0.095828	0.651466	0.649351
[100, 100, 100, 100]	0.913843	0.781759	0.740260

Problem 5)

Hand-simulate backprop. Consider the following artificial neural network.



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

$$(w_{31}, w_{32}) = (1, 1)$$
$$(w_{41}, w_{42}) = (1, -1)$$
$$(w_{51}, w_{52}) = (-1, -1)$$
$$(w_{63}, w_{64}, w_{65}) = (1, 1, 1)$$

(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 back propagation 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 back prop update. See the back propagation handout

for the procedure to use.

$$x_1 = 1 \qquad x_2 = 2$$
$$a_3 = w_{3,1} * x_1 + w_{3,2} * x_2 = 3 \qquad z_3 = max(0, a_3) = 3$$
$$a_4 = w_{4,1} * x_1 + w_{4,2} * x_2 = -1 \qquad z_4 = max(0, a_4) = 0$$
$$a_5 = w_{5,1} * x_1 + w_{5,2} * x_2 = -3 \qquad z_5 = max(0, a_5) = 0$$
$$a_6 = w_{6,3} * z_3 + w_{6,4} * x_4 + w_{6,5} * x_5 = 3 \qquad z_6 = a_6 = 3$$

We don't have the bias, so we can ignore b. now we can know the error is 0.5. Then we need calculate the δ_j values for each non-input nodes.

If j is the output node, $j = 6$:

$$\begin{aligned}\delta_j &= \frac{\partial error}{\partial a_j} \\ &= \frac{1}{2} * 2 * 1 * (3 - 2) \\ &= 1\end{aligned}$$

If j is not the output node, the f is Relu in this problem, and k is the nodes who use z_j:

$$\begin{aligned}\delta_j &= \frac{\partial error}{\partial a_j} \\ &= f^{'}(a_j) \sum_k \delta_k w_{k,j}\end{aligned}$$

Then we can get:

$$\delta_3 = w_{6,3} * \delta_6 * 1 = 1$$
$$\delta_4 = w_{6,4} * \delta_6 * 0 = 0$$
$$\delta_5 = w_{6,5} * \delta_6 * 0 = 0$$

Last, we can update w based on gradient:

$$\begin{aligned}w_{j,i} &= w_{j,i} - \eta * \frac{\partial error}{\partial w_{j,i}} \\ &= w_{j,i} - \eta * \delta_j * z_i\end{aligned}$$

We can replace the results we have got into the updated weights:

$$w_{6,3} = 1 - 0.1 * 1 * 3 = 0.7$$
$$w_{6,4} = 1 - 0.1 * 1 * 0 = 1$$
$$w_{6,5} = 1 - 0.1 * 1 * 0 = 1$$
$$w_{5,1} = -1 - 0.1 * 0 * 1 = -1$$
$$w_{5,2} = -1 - 0.1 * 0 * 2 = -1$$
$$w_{4,1} = 1 - 0.1 * 0 * 1 = 1$$
$$w_{4,2} = -1 - 0.1 * 0 * 2 = -1$$
$$w_{3,1} = 1 - 0.1 * 1 * 1 = 0.9$$
$$w_{3,2} = 1 - 0.1 * 1 * 2 = 0.8$$

Then we can continue to calculate the z, δ and error using the new weights, it should be less than 0.5.