Machine Learning CW3 Report
Yizheng Sun

3.1Explain briefly the knowledge supported your implementation and your design step by step:
My implementation works both for single-output case and multi-output case. It depends on training labels y. If y has one dimension, then model w has one dimension. if y has multi-dimensions, then model w has multi-dimension as well.

For training, when **λ** = 0, I just use Pseudo-inverse formula to calculate weight:

$$\mathbf{w^* = \tilde{X}^{\dagger} y}$$

$\tilde{X}^{\dagger}$ is the pseudo-inverse matrix. In our case, we firstly need to insert a column of ones into training samples matrix **X**. Because weight vector (or matrix) $\mathbf{w^*}$ begins with $\mathbf{w_0}$, we need a $\mathbf{x_0=1}$ to get:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d$$

Then we get $\tilde{X}$**,** which is a **N x (d+1)** matrix. **N** is the number of training samples and **d+1** is 1025. 1025 means there are 1024 features of each sample plus a number 1 at the beginning. To get $\tilde{X}^{\dagger}$, NumPy provides a simple built-in method numpy.pinv(X).

When **λ** > 0, I need to calculate the pseudo-inverse matrix by myself:

$$\boldsymbol{w^* = (\tilde{X}^T \tilde{X} + \lambda I) \tilde{X}^T y}$$

where I is an identity matrix, that has the same shape as $\tilde{X}^T \tilde{X}$. Thus, I get my weight vector(matrix) by training. For predicting, I need to calculate $\tilde{X}$ @ $\mathbf{w^*}$ to get my predict values. Then I need to classify the predict values by thresholding.

3.2 Experiment 1
Explain the classification steps:

1. Split data into training data and test data.
2. Set up training labels according to the instruction. There are 3 different kinds of training labels for 3 different classifiers.
3. For each classifier, pass training samples and labels to function l2_rls_train() to get our model.
4. Pass our model and test samples to function l2_rls_predict() to get our predicted values.
5. For each classifier, use the corresponding threshold to classify the test samples by predicted values. (e.g. for classifier 1, if predicted value <=0, it is classified as “1”, if predicted value>0, it is classified as “30”).

Does changing the class labels impact the model performance? Explain why it does/doesn’t impact:
The performances of the three classifiers basically same. They all have similar error rates and variances. So, changing the labels doesn’t impact the model performance. This is because we have balanced training data. We have 3 samples from subject “1”, and 3 samples from subject “30”. It doesn’t matter what values we give to these two classes. Because the two class values have equal impact on weight vector. As long as our threshold is a median value, the two class values will be treated as “True” and “False”. The actual values don’t impact classifier’s performance.

What training accuracies do you obtain with your linear classifier? Analyse the reason:
Repeat evaluating process for 50 times for each classifier.
Overall test accuracies of each classifier: 0.923, 0.919, 0.924;
Overall training accuracies of each classifier: 1, 1, 1.

Training error is 1 means our model is overfitting. This is because our linear model is derived by making the gradient of sum-of-square error loss 0 without regularization. Because our training system is undetermined as I mentioned in foreword, we can surely get solutions of w when gradient is 0. Then our model fits 100% to the training data, which is too much. Thus, our model got 0 training error rate but is not quite good at test data.

Experiment 2
Explain the classification steps:
1. Split data set into training data set (200) and test data set (200).
2. For each sample, my training label becomes a vector with length 40. Only the element whose index is this sample’s class is number 1. All the other elements in the vector are number 0. For example, sample

A belongs to subject 1, then it's training label is [1,0,0,0…0]. Sample B belongs to subject 2, then it's training label is [0,1,0,0…0].
3. Then I do hyper-parameter selection, I have 100 candidate λ value (1 to 100). For each λ, I perform 10 trails. For each trail, I use random subsampling to take 20 samples out from the training data set to use as test samples, and the remaining to train. Then I select a λ with the lowest error loss.
4. After training, I get a 1025 x 40 weight matrix (model). It can map each sample vector to 40 values. Each value is the predicted value for that class.
5. When testing, I multiply my test samples with my weight matrix. I have 200 test samples, so I get a 200 x 40 output matrix. Because in my training labels, I make the class value 1 and all the other classes value 0, in my output matrix, for each row I pick the element that is the closest to 1 and make its value 1, and all the other elements value 0.
6. Then I get my 200x40 predicted classes for all test samples. To make a 40x40 confusion matrix, I add every 5 rows up in the predicted classes matrix, because every 5 rows belong to a same class. Then in the confusion matrix, the index of a row is the actual class, the index of a column is the predicted class. For example confusion-matrix[2][5] means the number of samples from class 2 that are classified to class 5.

How do you pick the most difficult and easiest subjects to classify, analyse the results:
To pick the most difficult subjects to classify, I get the difference between my predicted classification result and test labels. Then I get the index of misclassified images. I do integer division by 5 on these indices to get the class they belong to. Therefore, I get the subjects that are difficult to be classified and misclassified images from them. To pick subjects that are easiest to classify, I look at my confusion matrix and find that subjects 36 to 39 are never misclassified. So, I pick them as easiest subjects to classify.

I look at the images that are misclassified. I find that these images don't have any special features to tell them apart from other subjects. They look very similar to others, so their data vectors are like others, such that they are more likely to be misclassified. Then I look at the image from subjects 36 to 39. It's obvious that these images have special features that can tell them apart from other subjects. For example, subject 36 has big beard and wears glasses. His data vector must be quite different from others and such that is less likely to be misclassified.
(I did both Experiment 2 and 3 in my code, but I only explain experiment 2 here in my report)

4. Gradient descent for training linear least squares model.
How did you choose the learning rate and iteration number, explain your result:
With learning rate 0.001 and 200 iteration times. My my error loss smoothly decreases to 0 and stays unchanged. Decreasing speed gets lower and lower with iteration number (i.e. derivative changes from a negative value to 0 and stays unchanged). Classification test accuracy is around 100% after 200 iterations.

With 0.01 learning rate and 200 iteration times, I get a very large error loss value that overflows. This is because the learning rate is too large, it takes us to a point with huge error loss. Theoretically, it still can go back to 0 with sufficient iteration times. However, python cannot handle the huge error loss value for me.

Difference between binary classification and multi-class classification:
Binary classification is single-output case, multi-class classification is multi-output case. When doing binary classification, our training labels are a one-dimension vector (i.e., one sample is corresponding to one single value). We need a threshold to determine which class this sample belongs.

When doing multi-class classification, our training labels are a N x 40 matrix (N is the number of training samples). Each row in this matrix contains two different values (e.g. 0 and 1, the class it belongs to is 1 and all the other values are 0). The model we got can map every sample to 40 values. Each value's index means a class. Then we don't need a threshold. We need to pick the one among the 40 values that is the closest to the original class value (e.g. 1). The index of this value is the class it belongs to.

5.2 Experiment Design:
I use stochastic gradient descent (SGD) and gradient descent (GD) to do binary classification (same as experiment4), I run SGD with learning rate 0.001, 0.004, with iteration number 1000. And I run GD with

learning rate 0.001,0.002 with iteration number 1000. And I record the running time and the iteration times to reduce error loss to 10 when learning rate is 0.001 for both methods.

Comparison 1: With the same learning rate 0.001, GD takes 167 iterations (update) to make error loss to 10. However, SGD takes 371 iterations (update) to make it 10. This is because GD is calculated by using the sum of all samples' squared error loss. However, SGD only picks one sample in each iteration to calculate the gradient of error loss. With a proper error loss, we may decrease the error loss of this single sample but increase the error loss of other samples. And we may get overall increasing error loss. That's why GD's graph is smooth, but SGD's graph is not. In general, GD takes fewer updates than SGD to reduce error loss.

Comparison 2: GD overflows with 0.002 learning rate, SGD doesn't overflow even with learning rate 0.004. This is because GD calculates the sum of error loss of all samples. It's much larger than a single sample's error loss that *is used in SGD. Error loss value in GD is more likely to overflow.*

Comparison 3: For SGD, it takes 2.93s to finish 1000 iterations with 0.001 learning rate (0.00053s each iteration). However, for GD, it takes 2.14s to finish 1000 iterations with 0.001 learning rate (0.00288s each iteration). Theoretically, SGD Should take less running time than GD, because GD spans over entire dataset within each iteration (each update), whereas SGD randomly takes just one sample data for each update.

However, here our SGD takes longer time than GD. Firstly, this is because when doing SGD, we need to generate a random integer in each iteration and index this integer to get a row of training sample matrix. We don't have these operations in GD. Secondly, we only have 6 training samples, calculating all 6 sample's error loss doesn't take much longer time than calculating one. These two reasons make SGD slower than GD.

- *Here are some of my own opinions about this course work. the data set in this course work has 400 samples (N) and each sample has 1024 features (d). Therefore N<d, which means our training system is underdetermined and this is very uncommon in real world cases. When it's undetermined, we can get many solutions of* $\boldsymbol{w}$ *when gradient is 0, and we need the one with the least* l_2 *norm. Since we can directly derive* $\boldsymbol{w}$*, it's not necessary to use gradient descent (GD) to perform more calculation to get* $\boldsymbol{w}$*. GD is usually for the case that we cannot find a solution of* $\boldsymbol{w}$ *when gradient is 0, which should be an overdetermined system (N>d). I hope this course work can be improved next year. It should use an overdetermined system to help us to understand gradient descent better.*