CS57800: Statistical Machine Learning

Homework 3

Due: Nov 13, 2018 on Tuesday

1 Open Ended Questions

1.1 first

The gradient of the loss function can be calculated as follows:

- The Loss function is

$$L = -\sum_n y^i \log(g(w, x_i)) + (1 - y^i) \log(1 - g(w, x_i))$$

- Now to calculate the gradient, we take the derivative of Loss function with respect to weight vector w. We know that derivative with respect to w is sum of partial derivatives with respect to $w_1, w_2 ... w_k$.
- For demonstration, we calculate derivative with respect to w_1. Now, this will be the gradient for weight component w_1. Similarly, we can find gradients for all other components $w_2, w_3 ... w_k$.
- Now,

$$\frac{\partial L}{\partial w_1} = -\sum_n y_i (\frac{(1 + e^{-z})^{-2} e^{-z} x_1}{g(w, x_i)}) + (1 - y_i)(\frac{(1 + e^{-z})^{-2}(-e^{-z}) x_1}{(1 - g(w, x_i))})$$

- This can be simplified to

$$\frac{\partial L}{\partial w_1} = -\sum_n (\frac{y_i(1 + e^{-z})}{1 + e^{-z}} x_1 - \frac{x_1}{1 + e^{-z}})$$

- i.e.

$$\frac{\partial L}{\partial w_1} = -\sum_n (y_i - g(w, x_i)) x_1$$

- Therefore, we have the gradient as

$$\delta w_1 = \sum_n (g(w, x_i) - y_i)x_1$$

for batch gradient descent.

- For stochastic gradient descent, we update weights every example, so it will be

$$\delta w_1 = (g(w, x_i) - y_i)x_1$$

after every example i.

1.2 second

- Now, we can prove that the loss function is convex by taking the second derivative. As seen in the previous part, we have the first derivative as:

$$\frac{\partial L}{\partial w_1} = \sum_n (g(w, x_i) - y_i)x_1$$

- Now, we see that taking second derivative gives us

$$\frac{\partial^2 L}{\partial^2 w_1} = \sum_n (\frac{(x_1)^2}{(1 + e^{-z})^2 e^z}$$

- Now, it is easy to see that the above expression is always greater than or equal to 0. Now, as the second derivative of L with respect to vector w is sum of partial derivatives of $w_1, w_2 ... w_k$, therefore the sum of all these greater than or equal to 0 values will also be greater than or equal to 0.
- Now, we know that if the second derivative of a function is always non-negative, then the function is convex.
- Hence, we prove that the logistic loss function is convex.

1.3 third

Regularization is basically a way to avoid overfitting our model on the training data. In other words, there will be some noisy data set in our training examples and we want to avoid overfitting on those noisy examples so that we can generalize well during test time. Regularization avoids having model with high weight values for all features which might get very high accuracy on training examples but not so on validation data set. In other words, the regularization term penalizes for higher weight values and encourages the model to give higher weight values to features only when they help in generalizing on a large number of training examples rather than few noisy points. L1 regularization can help in direct feature selection since the optimal combined loss function encourage the model to have some features with weight 0 so as to minimize the l1 loss. We tune the regularization factor λ to change the impact of regularization. L2 regularization is a more stable regularization which we use in the present assignment.

1.4 fourth

So, as we can see in our derivation in the first part, there will be one more term i.e.

$$\frac{1}{2}\lambda\|w^2\|$$

added to the loss function. So, we directly add the derivative of this term with respect to $w_1, w_2...w_k$ for respective gradients. Thus, The gradient for w_1 will be

$$\delta w_1 = \sum_n (g(w, x_i) - y_i)x_1 + \lambda w_1$$

1.5 fifth

Now, the intuition behind knowing if our model has converged is based on if the training loss is sufficiently low so that our model is not underfitting on the training data and is also performing well on the validation dataset. In our scenario, one way of knowing if the training loss is sufficiently low is knowing the delta loss by comparing the loss from the previous epoch with that of the loss of current epoch. If the difference in the change in loss is very low and the number of epochs is high, then this is a sufficient condition to know that our model has converged on the training dataset.
The stopping criteria which I have used for batch gradient descent is the delta loss to be less than $8 * 10^{-5}$ and the number of epochs to be greater than 100 i.e. atleast 100 complete passes on the entire training data set of 10,000 examples.
For stochastic gradient descent, we expect the function to converge faster since we are doing weight updates after every example. So, for SGD, I have used the stopping criteria to be delta loss to be less than $2 * 10^{-4}$ and number of epochs to be greater than 200. Note, that in SGD, I consider one epoch to be 1000 examples as suggested by the TA. Since, we will be printing too many outputs if we consider 1 epoch to be just 1 example. So, that means that SGD converges after just $20 - 30$ complete passes on the entire training data set of 10,000 examples.

1.6 sixth

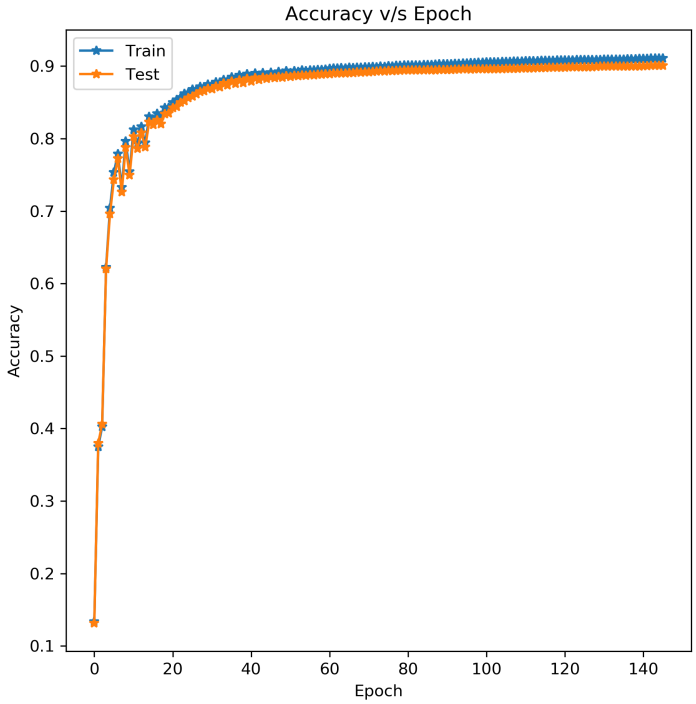
The bias term will help our activation function to shift towards right or left which can be useful in training our model. To understand this, we can visualize that as we change the values of weights for the features x_1, x_2 and so on, we are actually changing the steepness of the sigmoid curve. Now, there may be scenarios where we would just want to shift this curve towards right or left so that on a particular value of the features x_i, we want the function to give a specific value for sigmoid function. Now, the usefulness of bias term is that depending on the weight corresponding to the bias term, we can shift the sigmoid curve to the left or towards the right thus exhibiting desired behavior.

Thus, I believe that its useful so for type 1 feature set, my feature vector is of dimension 785 including 1 for the bias term and for type 2 feature set, my feature vector is of dimension 197 (196+1).

2 Batch Gradient Descent with Logistic Function

Write down the batch gradient descent algorithm with logistic function in appro- priate algorithmic format in LATEX. The algorithm is as given in ALGORITHM 1.

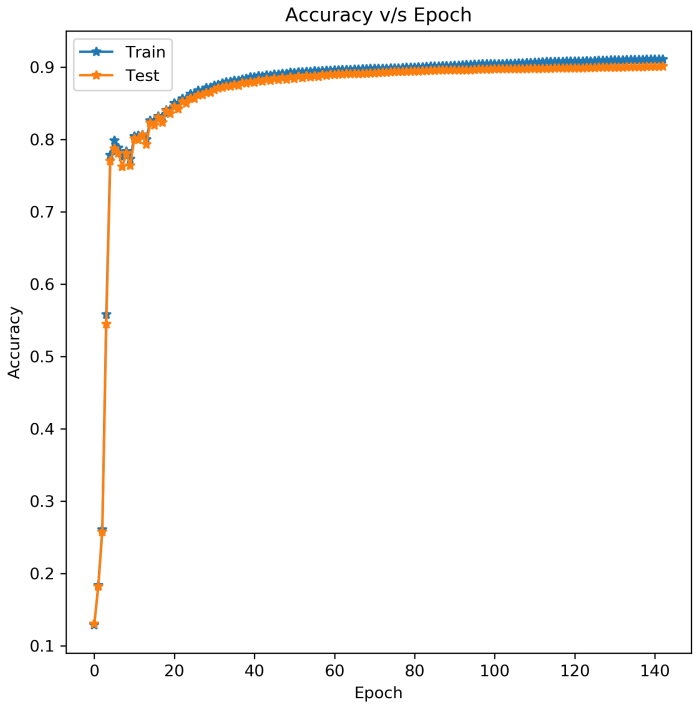
2.1 Analyze the convergence for batch gradient descent



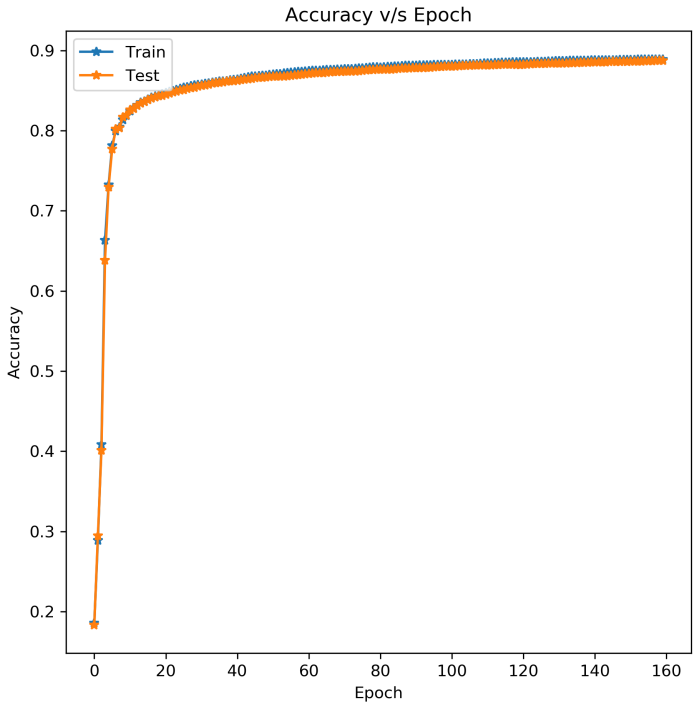
The implementation is in gd.py file. Now, the tuned values for λ is 0.0001, learning rate is 0.0001. As mentioned before the stopping criteria is delta loss to be less than 0.00008. The first image is showing the graph for change in accuracy on training and test data set with the number of epochs. This is for feature type 1 and without regularization. The test accuracy is around 0.902 and training accuracy is around 0.91.

Algorithm 1 Batch Gradient Descent with Logistic Function

```
1: procedure INFERENCE(dataset, weights)
2:     datasize = len(dataset)
3:     store features in examples from dataset
4:     store labels in labels from dataset
5:     initialize correct = 0
6:     for i, example in enumerate(examples) do
7:         activationvalues = sigmoid(np.sum(weights*example, axis = 1))
8:         prediction[i]=np.argmax(activationvalues)
9:         if prediction[i]==labels[i], then increment correct by 1
10:    accuracy = correct/datasize
11:    return accuracy
12:
13: procedure GRADIENTDESCENT(traindata, maxepochs, learningrate, lambda, testdata)
14:    datasize = len (traindata)
15:    featurelength = traindata.shape[1] (traindata will be 10000*785 for type1)
16:    weights=np.random.uniform(-0.1,0.1,[10,featurelength]
17:    bias=np.ones((datasize,1))
18:    prevloss = 1000
19:    for epoch in range(maxepochs) do
20:        create a list loss of size 10 initialized to 0
21:        shuffle the training data
22:        store features from traindata in examples
23:        store labels from traindata in labels
24:        store corresponding classifier labels in classifierlabels matrix
25:        initialize deltaweights = np.zeros((10,featurelength))
26:        for i, example in enumerate(examples) do
27:            z = np.sum(weights*example, axis=1)
28:            ypred = sigmoid(z)
29:            deltaweights += learningrate * np.outer(ypred-classifierlabels[:,i],example)
30:            loss+=classifierlabels[:,i]*log(ypred)+(1-classifierlabels[:,i]*log(1-ypred))
31:        weights = weights - deltaweights - learningrate*lamda*weights
32:        loss=-loss/datasize + lamda*np.sum(np.square(weights), axis=1)/2.0
33:        loss = np.sum(loss)/10
34:        Now we calculate the accuracy for training data set as well as test
35:        data set from procedure Inference i.e.
36:        trainaccuracy = INFERENCE(traindata, weights)
37:        testaccuracy = INFERENCE(testdata, weights)
38:        print loss, trainaccuracy, testaccuracy as calculated above, calculate deltaloss below
39:        deltaloss = prevloss - loss
40:        prevloss = loss
41:        if deltaloss is greater than 0 and less than 0.00008 and epoch is greater than 100,
42:        then return weights
    return weights
```

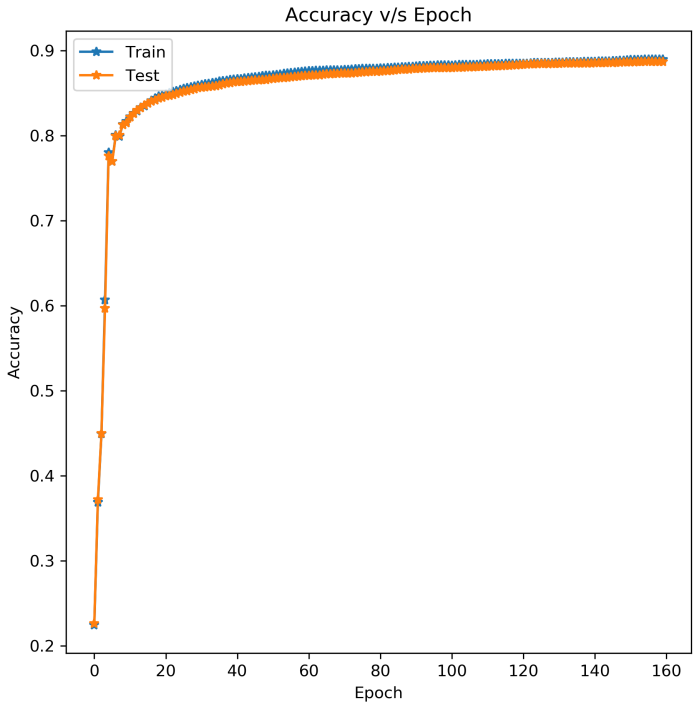


The second image is with regularization. As we can observe there is minimal performance difference since the model was having similar training and test accuracies without regularization. The final test accuracy is around 0.903 with training accuracy being around 0.91.



The third graph is for type 2 features without regularization. We observe that for type 2 features, both the train and test accuracies are marginally lower i.e. around 0.893 for training data set and 0.889 for test data set. This can be attributed to slight loss of information since we are subsampling the image from 28*28 to 14*14 size.

On a side note, preprocessing of type 2 feature set takes slightly more time since I had to run nested for loops to subsample the 28*28 image to 14*14 image size.



The fourth graph is for type 2 features with regularization. We observe that with regularization the results are similar. As observed before, the model is not overfitting on training data set since the accuracies are similar for training and testing data set without regularization i.e. 0.891 for training data set and 0.889 for test data set.

3 Stoachastic Gradient Descent with Logistic Function

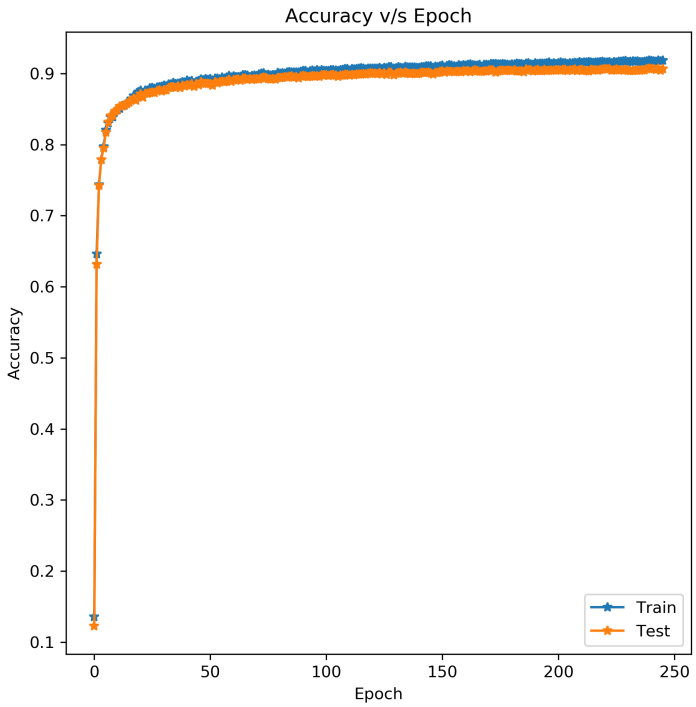
Write down the stochastic gradient descent algorithm with logistic function in appro- priate algorithmic format in LATEX

Algorithm 2 Stochastic Gradient Descent with Logistic Function

procedure INFERENCE(dataset, weights)
    datasize = len(dataset)
    store features in *examples* from dataset
    store labels in *labels* from dataset
    initialize *correct* = 0
    for i, example in enumerate(examples) **do**
        activationvalues = sigmoid(np.sum(weights*example, axis = 1))
        prediction[i]=np.argmax(activationvalues)
        if prediction[i]==labels[i], then increment correct by 1
    accuracy = correct/datasize
    return accuracy

procedure STOCHASTICGRADIENTDESCENT(traindata, maxepochs, learningrate, lambda, testdata)
    datasize = len (traindata)
    featurelength = traindata.shape[1] (traindata will be 10000*785 for type1)
    weights=np.random.uniform(-0.1,0.1,[10,featurelength]
    bias=np.ones((datasize,1))
    prevloss = 1000
    epoch = 0
    while epoch is less than maxepochs) **do**
        create a list *loss* of size 10 initialized to 0
        shuffle the training data
        store features from traindata in examples
        store labels from traindata in labels
        store corresponding classifier labels in classifierlabels matrix
        batchsizeloss = 1000
        for i, example in enumerate(examples) **do**
            initialize deltaweights = np.zeros((10,featurelength))
            z = np.sum(weights*example, axis=1)
            ypred = sigmoid(z)
            deltaweights += learningrate * np.outer(ypred-classifierlabels[:,i],example)
            loss+=classifierlabels[:,i]*log(ypred)+(1-classifierlabels[:,i]*log(1-ypred))
            weights = weights - deltaweights - learningrate*lamda*weights
            If i modulus batchsizeloss == 0, then run the below lines from 35-41
            loss=-loss/batchsizeloss + lamda*np.sum(np.square(weights), axis=1)/2.0
            loss = np.sum(loss)/10
            Calculate accuracy from procedure Inference i.e.
            trainaccuracy = INFERENCE(traindata, weights), testaccuracy = INFERENCE(testdata, weights)
            print loss, trainaccuracy, testaccuracy as calculated above, calculate deltaloss below
            deltaloss = prevloss - loss, loss=np.zeros(10),prevloss = loss, epoch += 1
            if deltaloss is > 0 and < 0.0002 and epoch is greater than 200, then return weights
    return weights

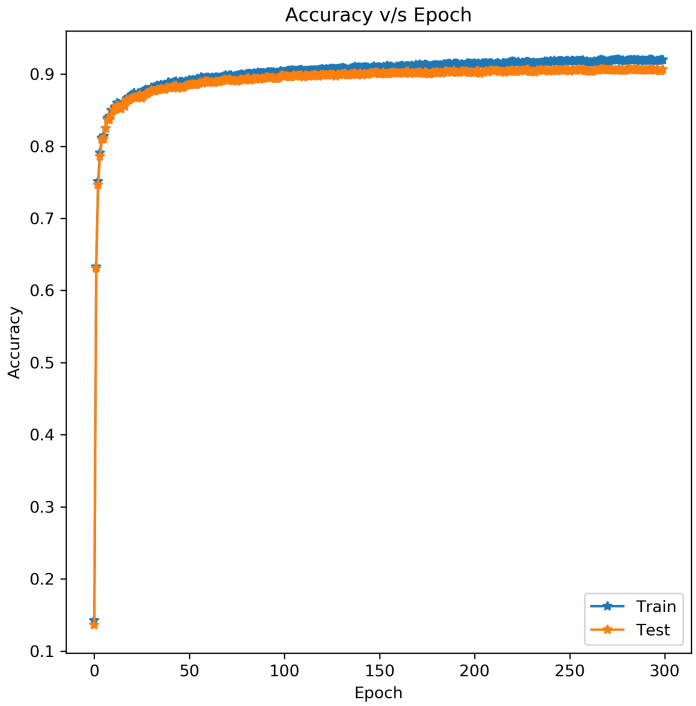
3.1 implement stochastic gradient descent



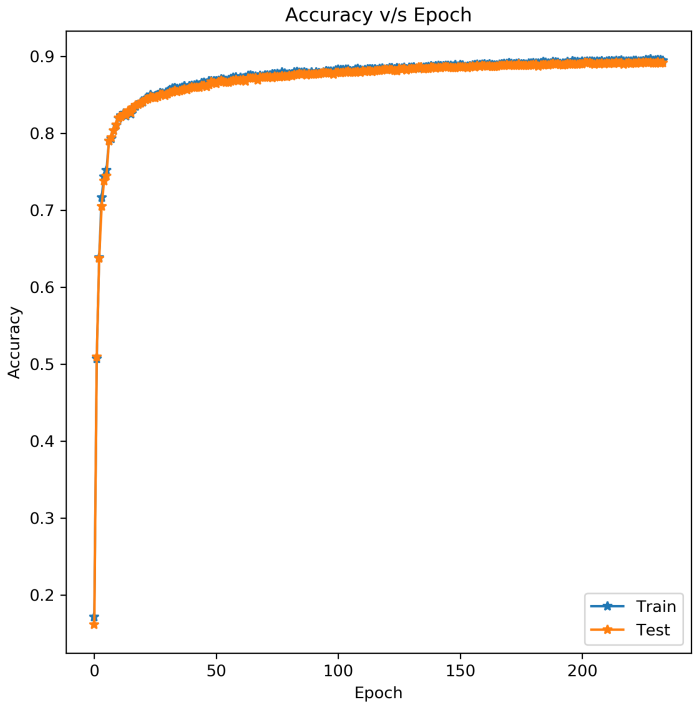
The implementation is in sgd.py file. Now, the tuned values for λ is 0.0001, learning rate is 0.001. The stopping criteria is delta loss to be less than 0.0002 and max epochs is 300. 1 epoch is defined as 1000 examples. Therefore 300 epochs means 30 complete passes on training data set.

The first image is showing the graph for change in accuracy on training and test data set with the number of epochs. This is for feature type 1 and without regularization.

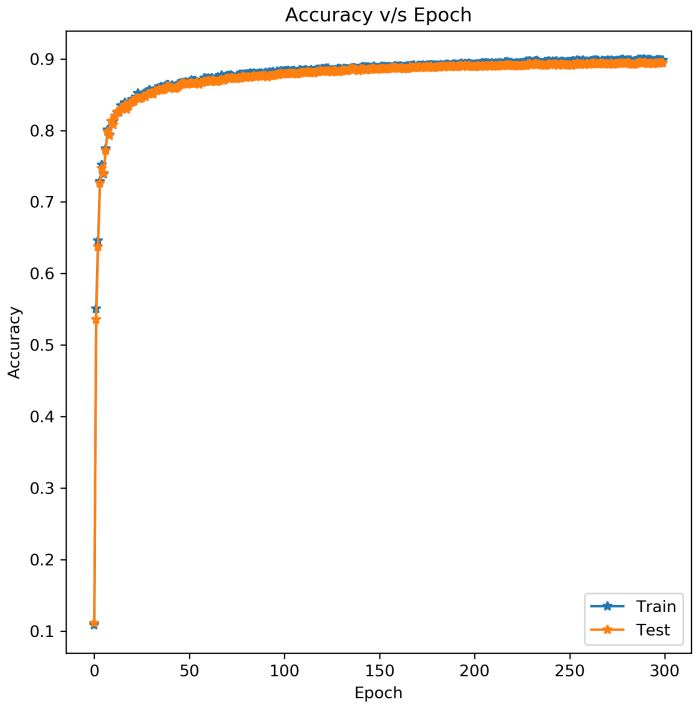
The main observation is that in comparison to batch gradient descent, SGD converges much faster i.e. just in 30 complete passes for the entire training data set, we get a training accuracy of around 0.92 and test accuracy of around 0.907.



The second image is with regularization. As we can observe there is minimal performance difference since the model was having similar training and test accuracies without regularization. The final test accuracy is around 0.9074 with training accuracy being around 0.92.



The third graph is for type 2 features without regularization. We observe that for type 2 features, both the train and test accuracies are marginally lower i.e. around 0.898 for training data set and 0.894 for test data set.



The fourth graph is for type 2 features with regularization. We observe that with regularization the results are similar. As observed before, the model is not overfitting on training data set since the accuracies are similar for training and testing data set without regularization.

3.2 Batch gradient descent vs stochastic gradient descent

We can observe that stochastic gradient descent has marginally higher test scores 0.9074 compared to batch gradient descent 0.903, the primary reason being that batch gradient descent requires much higher number of complete training data set passes to converge compared to stochastic gradient descent. SGD converges faster primarily because we update the weights after every example.