Rishith Kyatham
CSE353
11/27/2022

1. A) This is a bad idea to create a modal on all his 100 data points because there would be no data to test then. Due to this, you wouldn't be able to test with real data. This will cause an overfit for the training data and would not represent well as there would be no available data to train. It would be also harder as the models will have a high variance and low bias.

B) His best friend scowls at him as there will be a danger of overfitting as the model is formed by using only the testing data. There is no validation set mentioned here. An appropriate way to partition the data would be to create another set from the training set and make that set a validation set. Once we made this set a validation set, we can use the set to know an approximate place to stop the gradient descent. Overall, the model would not gain new information from just the test data. Splitting the validation set the same as how we split the training and testing set would be a first good course of action.

C) There would still be a danger of overfitting since we are still using the testing data in a repeated pattern to check the classification rate of our data. The problem that comes with this is that we don't account for all the factors such as the environments around when testing our accuracy. We adhere our model too specific to the test data given so when other factors come and different environments come, the model would have a tough time adapting to it. I agree with the fear of the danger of overfitting. An example of this would be, suppose we train our self driving car to drive in New York City and around some highways and over some weeks, the car perfectly does this. Now, suppose it randomly started snowing and then there would be an overfit due to the external factors that arose as the car might start losing some control, skidding, or other effects. These external factors should be included in the training or testing but they weren't there originally.

2. Part A) downloading the dorothea dataset

Part B) I believe in this exercise we have an unbalanced distribution of labels than the previous binary classification tasks we've done in this class. This exercise has sparse matrices involved as well such as X.

Part C)

```
...   0.04125 0.7724137931034484
      0.07428571428571429 0.5806451612903226

      <matplotlib.legend.Legend at 0x1e214365310>
```

The misclassification rate of the train is 0.04125 and the misclassification rate for the test is 0.0742857 etc. I observed that this misclassification rate is not a good classifier as no operations were performed and the rate is already too low.

Part D)

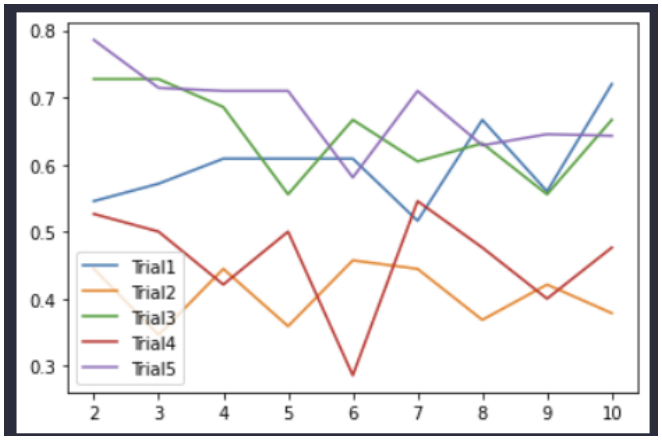
```
0.04125 0.7724137931034484
0.07428571428571429 0.5806451612903226

<matplotlib.legend.Legend at 0x1e214365310>
```

After looking at the metrics, I believe the F1 scores are much more accurate than misclassification rates because the data is unbalanced. The misclassification rate is a poor representation classifier as most of the data is 0 so when all of the values in a particular row is 0 and the y label is also 0 , it thinks that it classified it correctly. It misrepresents the data.
F1 Scores :
Training F1 Score : 0.7724137 etc… (Shown in picture above)
Testing F1 Score : 0.580645161 etc… (Shown in picture above)

Part E)

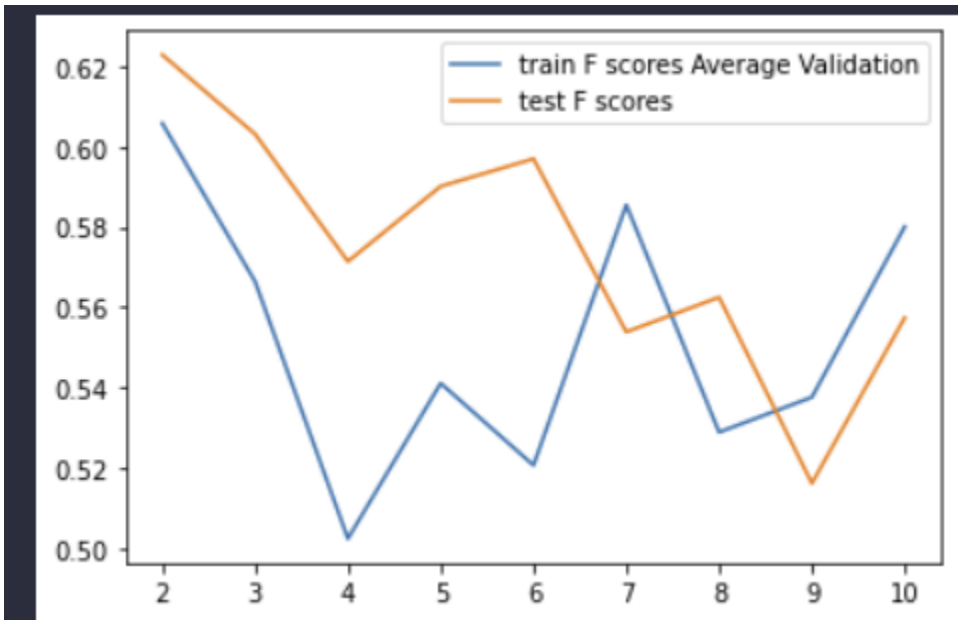
After plotting the train and test F1 scores for the sweep, I noticed that the training F1 scores increased almost constantly as the testing F1 scores went up and down around 0.6. As the model's accuracy increases, the F1 scores for the training set increase. There seems to be an overfit there due to the differences between the trainFScores and testFscores increasing.

Part F)



There is no stable trend here as the spikes fluctuate up and down. At each depth it seems to fluctuate at a similar area.

Part G)



In my graph it seems that near the second half of the graph, the train F scores for average validation seem to correlate very well with the test F scores. Overall, they correlate even though there seems to be some differences at peaks and troughs. From depths 6-10 is where the closest correlation lies. The test F scores were better for the most part of the first half of the graph and in the second half, the train F scores started performing better than test F scores overall. This was a successful venture since near the end, the train F scores are performing better than the test F scores.

3. Part A)

a) $f(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \log(\sigma(y_i x_i^T \theta))$ $\sigma(s) = \frac{1}{1+e^{-s}}$

for $y_i \in \{-1, 1\}$

$\nabla f_i(\theta) = (\sigma(y_i x_i^T \theta) - 1) y_i x_i$

$\nabla f(\theta) = \frac{\sum_{i=1}^{m} \nabla f_i(\theta)}{m} = \frac{1}{m} \cdot \sum_{i=1}^{m} \nabla f_i(\theta)$

Part B)

```
def getLossFunction(theta):
    totalSum = 0
    for i in range(0, m):
        firstMult = ytrain[i] * Xtrain[i].T
        secondMult = firstMult@theta
        res = sigmoid(secondMult)
        totalSum += np.log(res)
    totalSum = totalSum / m
    totalSum = totalSum * -1
    return totalSum

def sigmoid(s):
    return 1 / (1 + np.exp(-s))

def getGradient(theta):
    totalSum = 0
    for i in range(0, m):
        totalSum += helperGradient(theta, i)
    return totalSum / m

def helperGradient(theta, i):
    firstMult = np.dot(ytrain[i],Xtrain[i].T)
    secondMult = np.dot(firstMult, theta)
    resbefore = sigmoid(secondMult) - 1
    resMultExtra = np.dot(resbefore, ytrain[i])
    resFinalMult = np.dot(resMultExtra, Xtrain[i])
    return resFinalMult

# TEST SCRIPT. DO NOT MODIFY!
theta = np.linspace(-.1,.1,n)
print('Check number: ', getLossFunction(theta),np.sum(getGradient(theta)))
```

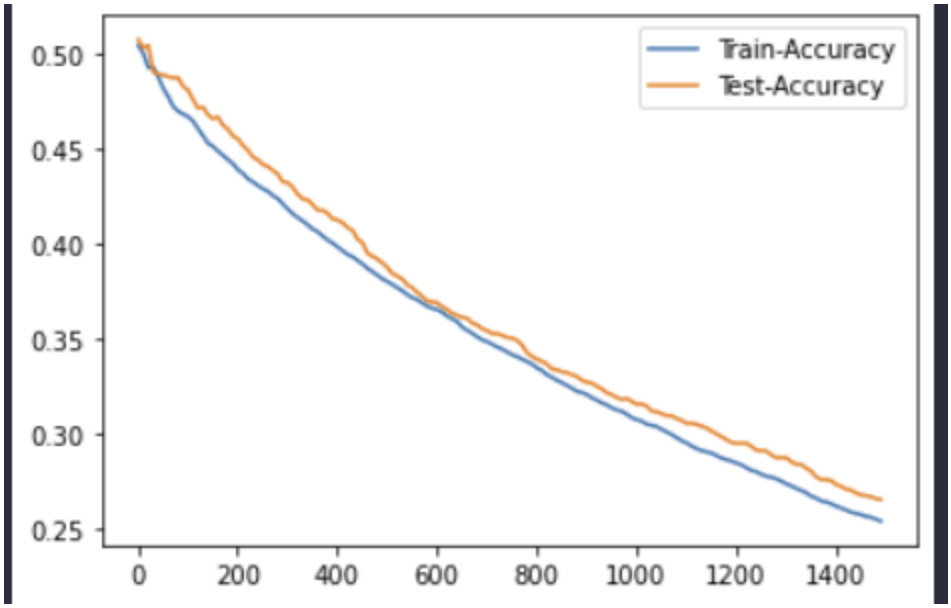
```
Check number:  45.19215648734918 12343.17694760448
```

The check number matches the same.

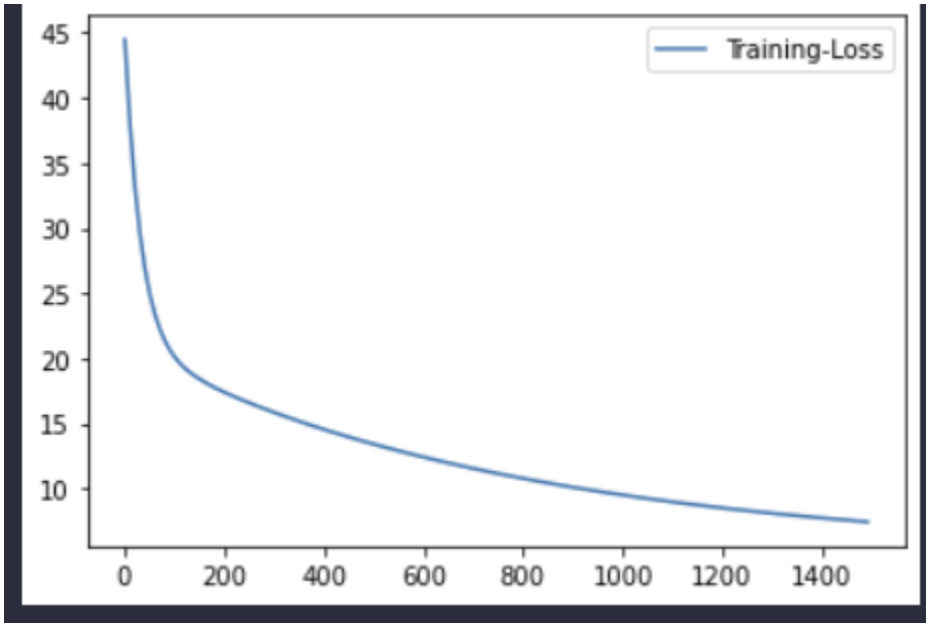
The classification accuracy I got is

```
[258]  ✓  3m 48.6s

...   Classification Accuracy is : 0.49546264099737086
```

Then after using gradient descent to minimize the logistic loss for this problem and running it for 1500 iterations, I got



This is the plot of the Train/Test misclassification rates plotted as a function of iterations from 0 to 1500.



This is the training loss plotted as a function of iterations from 0 to 1500.

```
The time taken to do gradient descent :  228.57369270000163
Final Train MisClassification Accuracy: 0.25443134594182004
Final Test MisClassification Accuracy: 0.26569563033651433
Final Train Loss is : 7.429541037850054
```

These are the results I got for the final train Misclassification accuracy, Misclassification test, final train loss, and the time to do this gradient descent.
Side Comment : (Not sure to do either misclass or class because the document does not specify as the plots required to be done in misclass. if you are only considering classification, then it's just 1 - what I got)

Part C)

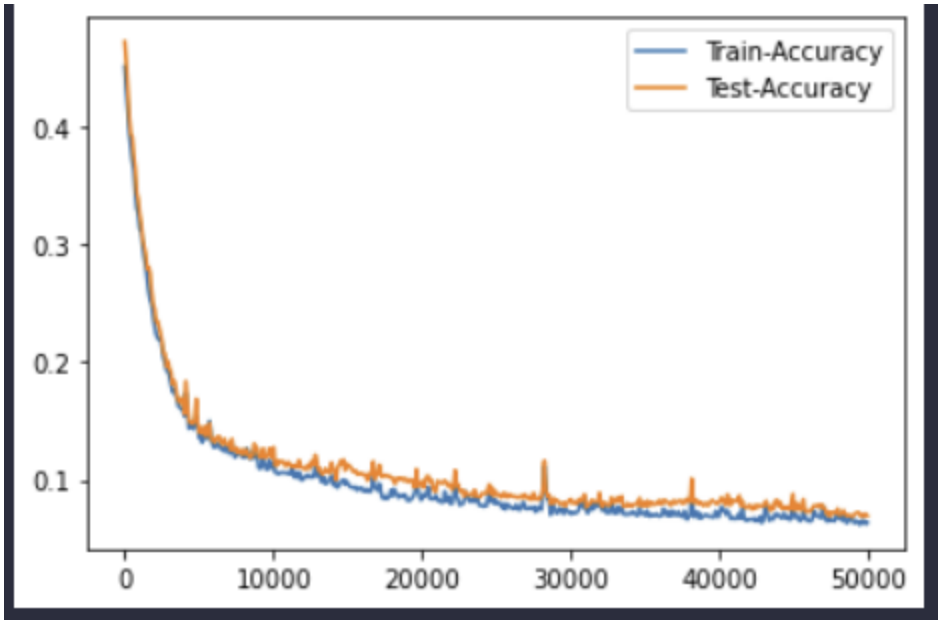
```
def getStochGradient(theta, minibatch):
    totalSum = 0
    for i in minibatch:
        totalSum += helperGradient(theta, i)
    return totalSum / len(minibatch)

def stoch_gradient_descent(theta, minibatch,stepsize):
    res = theta - (stepsize*getStochGradient(theta, minibatch))
    return res

# TEST SCRIPT. DO NOT MODIFY!
theta = np.linspace(-.1,.1,n)
print('Check number: ',np.sum(getStochGradient(theta,[1,4,6,2])))
```

✓ 0.3s

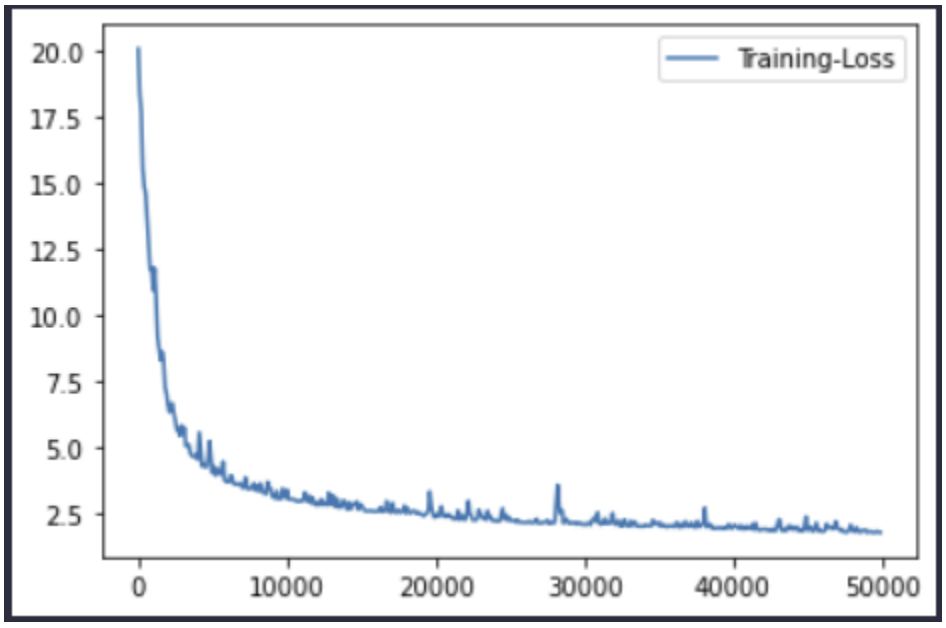
```
Check number:  5803.5
```

The Check number is stated below.

Part D)
After coding the stochastic gradient descent, I used a minibatch size of 50 data samples and ran it for 50000 iterations.
The plot for train accuracy and test accuracy comparison is :



The plot for the training Loss is :



The statistics that I got from this were :
The Time to do the Stochastic gradient descent : 82.75658759999897
The Final Train Misclassification Accuracy is: 0.06360783648545498
The Final Test MisClassification Accuracy is : 0.06931190356604722
The Final Train Loss is : 1.7629675157017
(If more information needed, check code, there's a small typo of it saying Classification accuracy when it should have been Misclassification Accuracy)

Commenting on the differences :

The Stochastic gradient descent seems to be a much better algorithm as it has a lower misclassification accuracy than normal gradient descent and a faster runtime as well. Even though you run the Stochastic gradient descent for more iterations, it gives you a lower misclassification and runs way faster. In my statistics, the Stochastic runtime ran in 82.756 seconds while the gradient descent ran in almost 3 times longer, 228.5736 seconds. The Misclassification accuracies for train and test are also almost 4 - 5 times higher than the Stochastic Gradient Descent which is bad. A con for the stochastic algorithm could be it takes too many iterations and uses elements of randomizations so results will change a lot. The stochastic algorithm uses a random array and if you don't do a lot of iterations, the sample size is not large enough so you will get graphs that vary a lot. Some pros of gradient descent is that fewer iterations and works better with smaller sample sizes.

Gradient descent :

```
The time taken to do gradient descent :  228.57369270000163
Final Train MisClassification Accuracy: 0.25443134594182004
Final Test MisClassification Accuracy: 0.26569563033651433
Final Train Loss is : 7.429541037850054
```

Stochastic Gradient Descent :
The Time to do the Stochastic gradient descent : 82.75658759999897
The Final Train Misclassification Accuracy is: 0.06360783648545498
The Final Test MisClassification Accuracy is : 0.06931190356604722
The Final Train Loss is : 1.7629675157017