**Assignment 2**

**N Kapil Chandra**

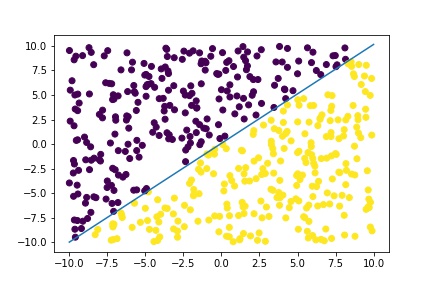
**MM14B022**

**Question 1:**

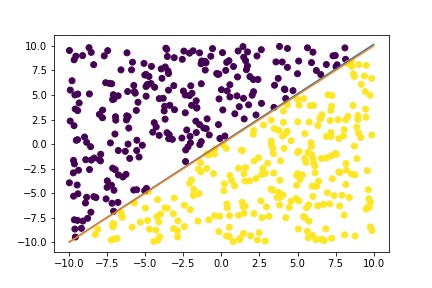
**(i) Since the format of the code is spoiling if I copy paste in .docx file I have put all the code in code.py file which can be found in the zipped submission folder.**

**(ii)** I generated my own data set by taking a line of my choice and then randomly generating points on either side of the line. I started of with the case where all my parameters are zeros and calculated the accuracy. My metric for accuracy being number of correctly predicted examples divided by total number of examples present. I calculated the accuracy when my model was zeros initially which came out to be 0.492, which is as expected. Now I ran my model to learn the parameters.

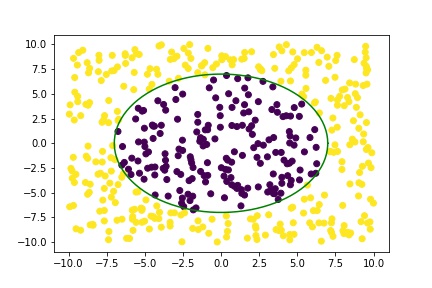
As I kept on increasing the number of iterations the accuracy score was also increasing. And in the case of batch gradient descent with a learning rate of 0.1 and 10000 iterations I achieved an accuracy of 1.0 which means the decision surface was able to divide the data set into two classes perfectly! I have attached a picture of my data and the decision surface I obtained below:



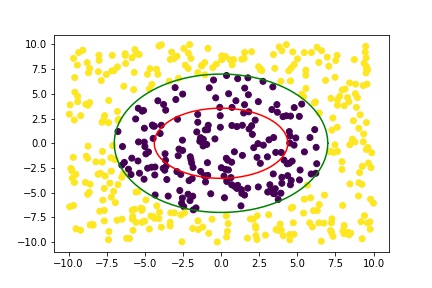
To generate my data set I had used the line y=x, so in the next plot I have plotted y=x line as red line and my decision surface as blue line:



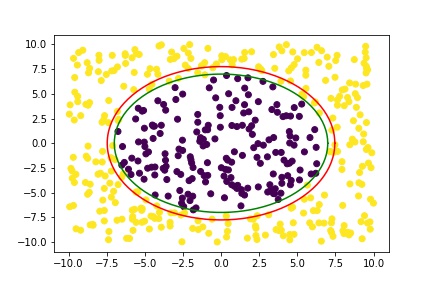
So from the above plot we can see that my decision surface is pretty close to y=x line, but it is not overlapping exactly. If I train my model on more number of data points then I can expect my decision surface to overlap with y = x line.

Coming to the non-linear case, I have made a function which takes parameters of an ellipse and generates data in and outside the ellipse with labels. My data looks something like this:  


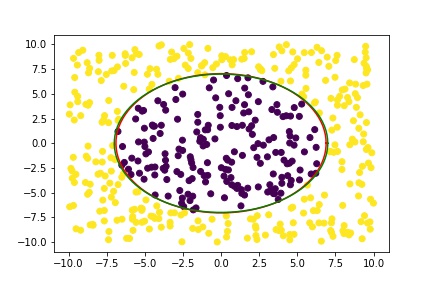
I ran my batch gradient descent algorithm on the above dataset with a learning rate of 0.1 and after 500 iterations my decision surface looked something like this: (The red ellipse is the decision surface)



The accuracy of the algorithm was 0.75. But if I run the algorithm for 1000 iterations I get an accuracy of 0.932 and the figure would look like this:



Now I ran the algorithm for 30000 iterations with the same learning rate and obtained an accuracy of 0.998 which is pretty good, the algorithm basically classified everything correctly except 1 data point out of 500 points. The decision surface looked like this:



Another way I can vouch for results are that I have used scikit-learn package which was taught in the class and that also got an accuracy of 1.0.

**(iii)** The major modification made in the case of non-linear case is the introduction of non linear features in the data set. Instead of taking x1 and x2 to train the model, I have taken x1\*\*2 and x2\*\* because I know that I cannot get a good accuracy with straight line, and I also know how the equation of ellipse looks like. If I use a linear model the accuracy I get will be around 0.5. So to introduce non-linear features I just squared the features in training data set and used the same code written above to train my model.

**(iv)**  For the data set I have created I felt that stochastic gradient descent worked better for me because in the linear case to get an accuracy of 1.0 for a learning rate of 0.1 I had to run the algorithm for 10000 iterations whereas when I used stochastic gradient descent I got an accuracy of 1.0 for 50 iterations itself. So stochastic gradient converged faster for me. In the non-linear case similar thing happened, in stochastic gradient descent with learning rate of 0.1 to achieve 0.92 accuracy 2000 iterations were needed, whereas in batch gradient descent for 2000 iterations only 0.86 was achieved, so I had to run it for 50000 iterations to get an accuracy of 0.998. But the time taken to run stochastic gradient descent for 2000 iterations was slightly more than batch gradient descent.

**So generally I would prefer to use batch gradient descent when there are large number of training examples because in that case looping over number of training examples would be very time consuming even though the algo might take less number of iterations to actually converge to the solution. And since the data set here is relatively less complex it makes sense to use batch over stochastic. But if the data is complex then I would prefer stochastic so as to avoid getting stuck at a local minimum.**

**Question 2 :**

**(i)Since the format of the code is spoiling if I copy paste in .docx file I have put all the code in code.py file which can be found in the zipped submission folder.**

**(ii)** I have trained a simple logistic model on the train data set provided in the website. I didn’t use stochastic gradient descent in this case because there we many training examples and it would be computationally time taking to iterate through training examples in the case of stochastic gradient descent, so I used batch gradient descent.

I ran the algorithm with a learning rate of 0.00001 and ran the algorithm for 100 iterations to get an accuracy of 0.9980260560600079 which is good. And as I am increasing the number of iterations the accuracy is also increasing by some value.

I used testing data set to check the accuracy of my model and got an accuracy of 0.9995271867612293 which means that my model is working pretty well on the testing data set. As I increase the number of iteration while training the accuracy on testing data set is also increasing.

I again use scikit-learn package to run logistic regression on the data set to check, and it gave an accuracy of 0.999921042242 on training data set and 0.998581560284 on test data set, and my results are pretty close to it. So my model seems to be fine.

**(iii)**  For normalization I have just divided the data by 255 because the minimum value in a pixel is 0 and the maximum value is 255.

**Without normalization:**

**Training accuracy :** 0.9980260560600079

**Test accuracy :** 0.9995271867612293

**With normalization:**

**Training accuracy :** 0.9903671535728386

**Test accuracy :** 0.9957446808510638

It appears that given a learning rate and a training accuracy applying normalization is not affecting the accuracy that much, the difference is only in the third decimal. One way of justifying this would be that all of our data is in the same range of (0,255) so applying normalization won’t make much of a difference, however when different features if different ranges are then normalization would be crucial and will give a significant difference because features will be given different preference based on their magnitude.

There is one more observation, we can see that even though the accuracy doesn’t change that much after feature normalization, it slightly decreases, one explanation can that when I am decreasing the feature range from (0,255) to (0,1), since the magnitude is decreasing of all features the gradients associated with them are also expected to decrease and hence it would take more number of iterations to reach that accuracy.