Homework 2 - Logistic Regression

Isu Kim 32190984

October 6, 2022

1 Task 1.

Let's see how `train.csv` looks like in scatter plot. Since we are going to train our model using logistic regression with data `train.csv`, let's ignore `test.csv` for now.

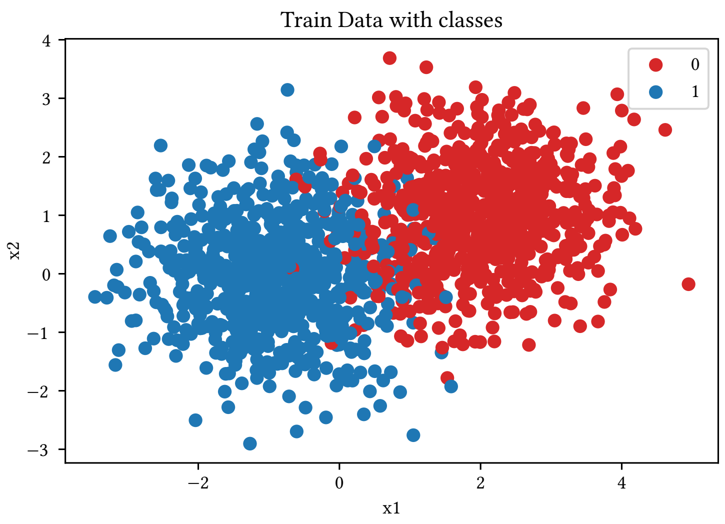


Figure 1: Scatter plot of `train.csv`

Scatter plot of `train.csv` looks something like this. Now, let's train our model using logistic regression. For logistic regression's cost function, we are going to use cross entropy. Also, for optimization, we will be using gradient descent method for getting $h_\theta(x)$.

Let our model's exponent as it follows. For now, we are going to use linear function. However, in the future when competing accuracy, we might be using some other functions.

$$\theta x^i := \theta_0 + \theta_1 x_1^i + \theta_2 x_2^i$$

That will make our model as following equation.

$$h_\theta(x^i) = \frac{1}{1 + e^{-\theta x^i}} = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1^i + \theta_2 x_2^i)}}$$

It is well known fact that using gradient descent as optimizing algorithm for logistic regression with cross entropy will derive an update rule like below.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} [h_\theta(x^i) - y^i] x_j^i$$

So, we can make a pseudo-code with using the update rule.

Algorithm 1 Logistic Regression

```
while Not converged do
    tmp = {0 , 0, 0}
    for $j = 0$ to 2 do
        $sum \leftarrow 0$
        for $i = 1$ to $m$ do                                ▷ $m$ is total data count
            $sum \leftarrow sum + [h_\theta(x^i) - y^i]x_j^i$
        end for
        $gradient \leftarrow sum/m$
        $tmp_j \leftarrow \theta_j - \alpha \cdot gradient$                ▷ $\alpha$ is learning rate
    end for
    $\theta \leftarrow tmp$                                  ▷ Update weights simultaneously
end while
```

Let's implement the pseudo-code using `Python`. As I have mentioned before, `Python` offers us powerful libraries that can process large data as well as data visualization, we will be using `Python` over `C` or `C++`. The code can be found in `Task1.ipynb` which is attached with this document.

For executing `Task1.ipynb`, you need `Jupyter Notebook` installed in your PC. Once you have installed `Jupyter Notebook`, you can execute my code line by line with simply hitting `SHIFT + Enter`. The code has lots of comments which will help you understand the code, therefore I will not provide detailed description on each function or each feature in this document except major functions.

Those are the list of functions in `Task1.ipynb`:

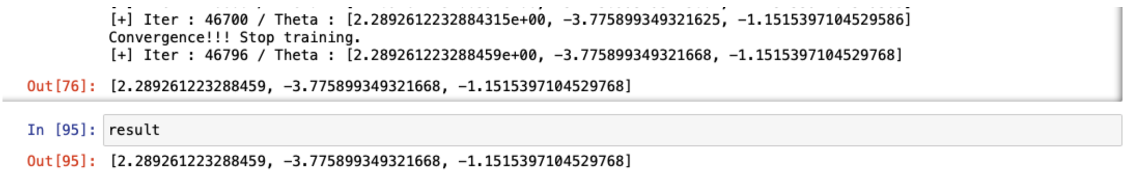
- `sigmoid`: For calculating sigmoid function.
- `linear_model`: For calculating θ_x^i.
- `hypothesis`: For calculating $h_\theta(x)$.
- `get_gradient`: For calculating gradient for specific j.
- `do_gradient_descent`: For processing gradient descent.
- `predict`: For predicting results with the model.
- `calculate_accr`: For calculating accuracy of model. This is measured as $Correct/Total$.
- `calculate_cost`: For calculating cost of model. This is measured with cross entropy from page 25 from `[Slide] 03. Logistic Regression.pdf`.

If you would like to take a look on the implementation of each function, please check `Task1.ipynb` for more information. Each function has detailed information on arguments, return values and what it does. Thus please take a look into each code if you like to know more about the code.

The code `Task1.ipynb` was checked running in my own private server with following specifications:

- `Python 3.9`
- `Numpy 1.12.5`: For processing math operations.
- `Pandas 1.4.2`: For loading `.csv` file.
- `Matplotlib 3.5.1`: For visualization.
- `Jupyter Notebook 6.4.8`: For general execution as well as `.ipynb` file.

Now, let's execute code by using `do_gradient_descent()` from `Task1.ipynb`.



```
           [+] Iter : 46700 / Theta : [2.2892612232884315e+00, -3.775899349321625, -1.1515397104529586]
           Convergence!!! Stop training.
           [+] Iter : 46796 / Theta : [2.289261223288459e+00, -3.775899349321668, -1.1515397104529768]

Out[76]:   [2.289261223288459, -3.775899349321668, -1.1515397104529768]

In [95]:   result

Out[95]:   [2.289261223288459, -3.775899349321668, -1.1515397104529768]
```

Figure 2: Training with Gradient Descent

As you can see, with `Task1.ipynb`, it took around 46796 iterations to converge all θs. The total number of iterations might vary time to time depending on your arguments. In my case, for extra precision, I gave no rounding for checking convergence. Meaning that the process was iterated until all 16 fractional digits were identical. As a result, each θs has value of following table.

θ_0	θ_1	θ_2
2.289261223288459	-3.775899349321668	-1.1515397104529768

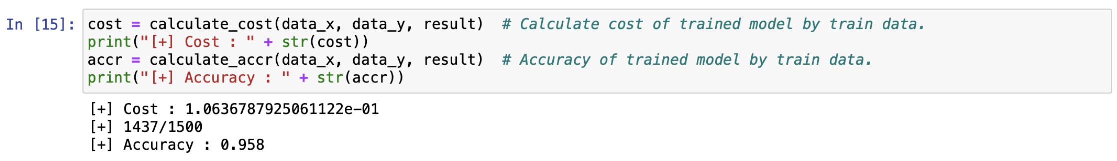
Therefore, our model is something like this equation.

$$h_\theta(x^i) = \frac{1}{1 + e^{-(2.289261223288459 + -3.775899349321668 x_1^i + -1.1515397104529768 x_2^i)}}$$

Meaning that

$$\theta_j = 2.289261223288459 - 3.775899349321668 x_1^i - 3.775899349321668 x_2^i$$

Now, let's check our model's accuracy and cost using `calculate_accr` and `calculate_cost` respectively.



```
In [15]: cost = calculate_cost(data_x, data_y, result)  # Calculate cost of trained model by train data.
         print("[+] Cost : " + str(cost))
         accr = calculate_accr(data_x, data_y, result)  # Accuracy of trained model by train data.
         print("[+] Accuracy : " + str(accr))

         [+] Cost : 1.0636787925061122e-01
         [+] 1437/1500
         [+] Accuracy : 0.958
```

Figure 3: Accuracy and Cost with `train.csv`

Since 1437 data were classified correctly out of 1500 total data, the model had accuracy of 95.8%. Also, the cost measured with cross entropy was 0.10636787925061122. In summary we can represent our training model with `train.csv` as following table:

Accuracy	Cost
95.8%	0.10636787925061122

Now, with those θ values, let's plot decision boundary with `train.csv`.

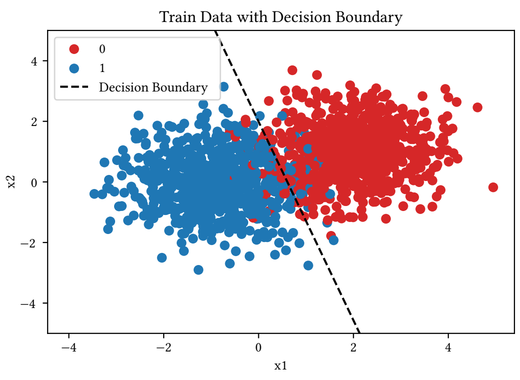


Figure 4: Decision Boundary with `train.csv`

For correct and wrong classifications, the plot looked something like following plot.

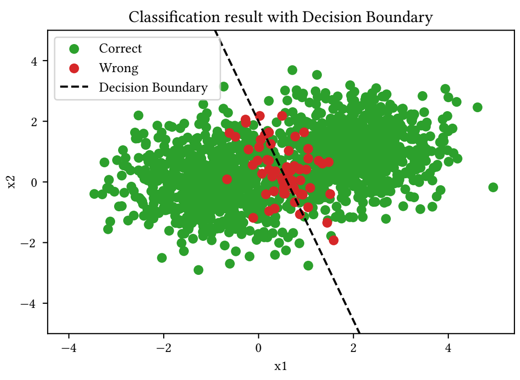


Figure 5: Classification Results with `train.csv`

Now, let's take a validate our model using `test.csv` which was not used in training phase. Meaning that the model will be classifying data from `test.csv` which the model never had seen before.

2 Task 2.

In task 2, our goal is to validate our trained model with `test.csv`. Let's load `test.csv` and check accuracy and cost using `calculate_accr` and `calculate_cost` respectively.

```
In [63]: # This is for Task 2. from homework 2.
         # This will load hw2_test.csv and validate cost and accuracy.
         df_test = pd.read_csv("hw2_test.csv")  # Read csv file.
         data_x = df_test[["x1", "x2"]]  # Select x1 and x2 col from data
         data_y = df_test["y"]  # Select y col from data
```

```
In [64]: print("[+] Test data")
         cost = calculate_cost(data_x, data_y, result)  # Calculate cost of trained model by test data.
         print("[+] Cost : " + str(cost))
         accr = calculate_accr(data_x, data_y, result)  # Accuracy of trained model by test data.
         print("[+] Accuracy : " + str(accr))

         [+] Test data
         [+] Cost : 8.807312045946367e-02
         [+] 483/500
         [+] Accuracy : 0.966
```

Figure 6: Accuracy and Cost with `test.csv`

Since 483 data were classified correctly out of 500 total data, the model had accuracy of 96.6%. Also, the cost measured with cross entropy was 0.08807312045946367. In summary, with `test.csv`, our model's performance can be represented as following table:

Accuracy	Cost
96.6%	0.08807312045946367

Also, let's plot decision boundary with data from `test.csv`.

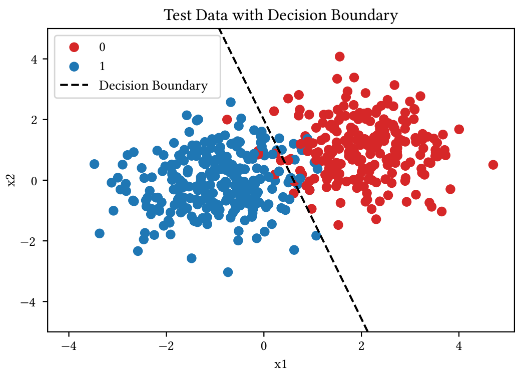


Figure 7: Decision Boundary with `test.csv`

As you can see, the decision boundary could divide each classes quite reasonably just like it did with `train.csv`.

For correct and wrong classifications, the plot looked something like following plot.

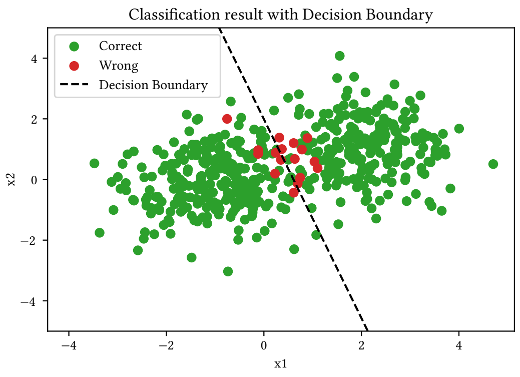


Figure 8: Classification Results with `test.csv`

3 Beating Accuracy

Now, since professor offered us extra points if we could get the highest accuracy with test data, let's try beating accuracy. The method that beats accuracy that I can think of right now is 'overfitting'. Now, I will be increasing each polynomial's degree and see how its results goes.

As we all know, with Task 1, we have used degree of 1 function which was:

$$\theta x^i := \theta_0 + \theta_1 x_1^i + \theta_2 x_2^i$$

If we generate some functions with higher degree, I am expecting the decision boundary to be a bit more smooth. Therefore, we can hopefully get some model with higher accuracy. So that I can earn extra points.

So there are following equations that I am going to try fitting our model into:

1. $\theta x^i := \theta_0 + \theta_1 (x_1^i)^2 + \theta_2 x_1^i x_2^i + \theta_3 \cdot (x_2^i)^3$
2. $\theta x^i := \theta_0 + \theta_1 (x_1^i)^3 + \theta_2 (x_1^i)^2 x_2^i + \theta_3 x_1^i \cdot (x_2^i)^2 + \theta_4 (x_2^i)^3$
3. $\theta x^i := \theta_0 + \theta_1 (x_1^i)^4 + \theta_2 (x_1^i)^3 x_2^i + \theta_3 (x_1^i)^2 (x_2^i)^2 + \theta_4 x_1^i (x_2^i)^3 + \theta_5 (x_2^i)^4$
4. $\theta x^i := \theta_0 + \theta_1 (x_1^i)^5 + \theta_2 (x_1^i)^4 x_2^i + \theta_3 (x_1^i)^3 (x_2^i)^2 + \theta_4 (x_1^i)^2 (x_2^i)^3 + \theta_5 x_1^i (x_2^i)^4 + \theta 5 (x_2^i)^5$
5. $\theta x^i := \theta_0 + \theta_1 (x_1^i)^7 + \theta_2 (x_1^i)^6 x_2^i + \theta_3 (x_1^i)^5 (x_2^i)^2 + \theta_4 (x_1^i)^4 (x_2^i)^3 + \theta_5 (x_1^i)^3 (x_2^i)^4 + \theta_6 (x_1^i)^2 (x_2^i)^5 + \theta_7 (x_1^i)^1 (x_2^i)^6 + \theta_7 (x_2^i)^7$

3.1 Degree 2

With degree of 2, we got following plots:

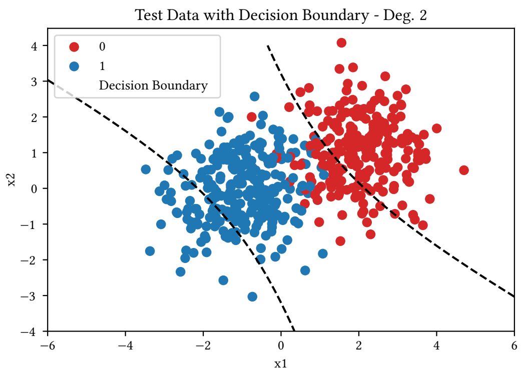


Figure 9: Degree 2 Decsion Boundary with `test.csv`

You can find this code in `./overfit/Overfit1.ipynb`. The θ values was like below. (All in order from θ_0 to θ_3)

```
[1.476914991886026e+00, -0.320823258042601, -0.6262254831483871,
-0.14402101072483853]
```

With training data, the accuracy was 0.766. At the same time, with test data, the accuracy was also 0.766 as well. As we can see easily, since the hyperbola did not represent decision boundary well, the results were not that good. Even compared to the linear model that we observed in Task 1, this had worse accuracy.

3.2 Degree 3

With degree of 3, we got following plots:

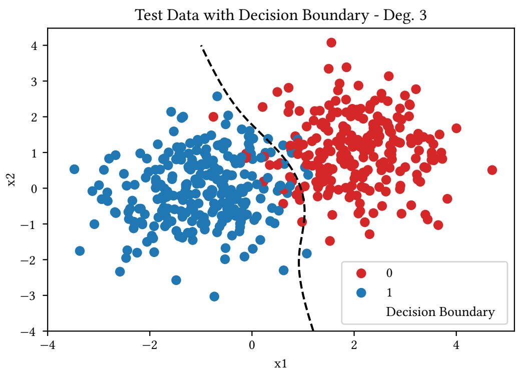


Figure 10: Degree 3 Decsion Boundary with `test.csv`

You can find this code in `./overfit/Overfit2.ipynb`. The θ values was like below. (All in order from θ_0 to θ_4)

```
[1.7569448834471597e+00, -2.0049664463859713, -1.3747949223559732,
-1.3747949223559732, -0.3156989588444056]
```

With training data, the accuracy was about 0.949. With test data, the accuracy was about 0.954. As you we can see easily, the model was not that bad compared to degree 1. It has a decent decision boundary.

3.3 Degree 4

With degree of 4, we got following plots:

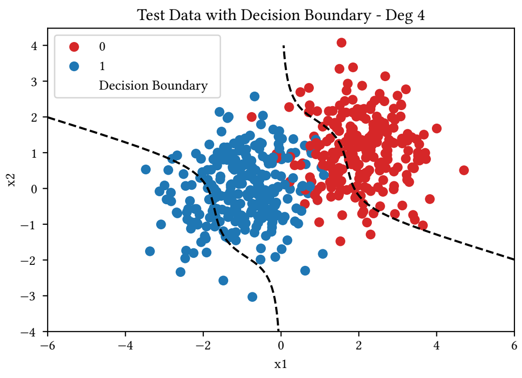


Figure 11: Degree 3 Decsion Boundary with `test.csv`

You can find this code in `./overfit/Overfit3.ipynb`. The θ values was like below. (All in order from θ_0 to θ_5)

```
[1.4362249202776316e+00, -0.11420703312960229, -0.21657500346596437,
  0.2767189454647682, -0.2989477106750623, -0.0008083725381591508]
```

As by now, we can see that degree of even numbers tend to have a bad decision boundary. The model had accuracy about 0.768 for train data, and 0.768 as well for train data. We can see that if the degree was even numbers, the model tends to become a hyperbola. So, in next cases, we will skip even numbers.

3.4 Degree 5

With degree of 5, we got following plots:

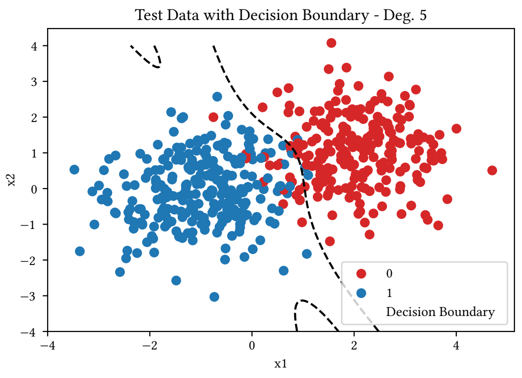


Figure 12: Degree 5 Decsion Boundary with `test.csv`

You can find this code in `./overfit/Overfit4.ipynb`. The θ values was like below. (All in order from θ_0 to θ_6)

```
[1.5194339749778079e+00, -1.3108311748288604, -0.6242050482171536,
-0.021951487687504257, -0.428930990806587, -0.27626922544025784,
-0.03803860853867751]
```

As we all can see, the decision boundary was clean. Resulting accuracy of 0.936 for train data and 0.958 for test data. So, let's try degree of 7.

3.5 Degree 7

With degree of 7, we got following plots:

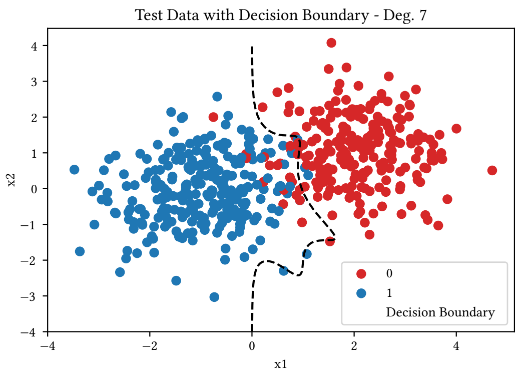


Figure 13: Degree 7 Decsion Boundary with `test.csv`

You can find this code in `./overfit/Overfit5.ipynb`. The θ values was like below. (All in order from θ_0 to θ_8)

```
[1.552072642268509e+00, -1.866146335486137, -3.4279440602850113,
0.22602513651005718, 2.634078213556618, 0.27558394167076933,
-0.6166279573455136, -0.16420138646465027]
```

As we all can see, the decision boundary divided each classes very well. Resulting accuracy of 0.932 for train data and 0.948 for test data.

3.6 Comparison

Now, let's compare the results of each degrees.

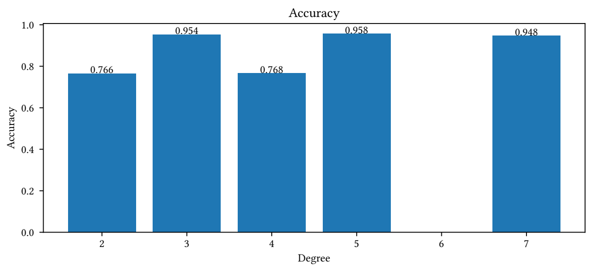


Figure 14: Accuracy for Test data

We can tell that using complex function as exponent of sigmoid function is not the key point for getting good accuracy. All those complex functions failed to beat linear function that was used in Task 1, which had accuracy of 0.966.

Not only did complex functions took more time for convergence, but it also needed more computing power since the derivatives were not just x_j^is. Also, it did not have better results than the original linear function. So in summary, I learned something new: Using complex function is not the key point for success.

4 Conclusion

As a conclusion, my attempt to overfit our model into data using complicated functions failed. Thus, I will be submitting the result with linear equation that was used in Task 1. So, let's compare model's performance between `train.csv` and `test.csv`.

Type	Accuracy	Cost
`train.csv`	95.8%	0.10636787925061122
`test.csv`	96.6%	0.08807312045946367

The model performed quite well with both `train.csv` and `test.csv`. However, there was a major problem with the model that was found. Which was the fact that it took so much time to train the model. As I have mentioned before, the model took 46796 iterations to converge every θs with learning rate $\alpha = 0.1$. For last homework's task 1, which was a linear regression with gradient descent algorithm, the model needed 685 iterations with learning rate $\alpha = 0.1$.

Compared to last homework's task 1, task 1 from this homework needed 68 times more iterations for it to converge. It is an obvious fact that more iterations mean more time for training. So, I looked into the reason why it needed so much more iterations compared to linear regression.

In short, the reason turned out to be the gradient being too small. Let's imagine that we are going to update a θ value using the update rule mentioned in Task 1. In that case, the gradient is following equation:

$$gradient = \frac{1}{m}\sum_{i=1}^{m}[h_\theta(x^i) - y^i]x^i$$

As we all know, $h_\theta(x^i)$ is a sigmoid function. Thus, we can derive following result.

$$0 < h_\theta(x^i) < 1$$

This will derive following inequations:

$$\begin{cases} 0 < h_\theta(x^i) - y^i < 1 & y^i = 0 \\ -1 < h_\theta(x^i) - y^i < 0 & y^i = 1 \end{cases}$$

Let's assume that all $x_j^i > 0$. With the `train.csv`, this is not the case, however this assumption is for demonstration. Then, this also derive following inequations.

$$\begin{cases} 0 < (h_\theta(x^i) - y^i)x_j^i < x_j^i & y^i = 0 \\ -x_j^i < (h_\theta(x^i) - y^i)x_j^i < 0 & y^i = 1 \end{cases}$$

With the inequation that was derived right before, let's add up all those values from $i = 0$ to $i = m$. In our case, m will be 1500 since `train.csv` had total data of 1500. That will derive following inequations:

$$\begin{cases} 0 < \sum_{i=1}^{1500}(h_\theta(x^i) - y^i)x_j^i < \sum_{i=1}^{1500} x_1^i & y^i = 0 \\ -\sum_{i=1}^{1500} x_j^i < \sum_{i=1}^{1500}(h_\theta(x^i) - y^i)x_j^i < 0 & y^i = 1 \end{cases}$$

Also, with our `train.csv`, we can know following values:

$$\sum_{i=1}^{1500} x_1^i \approx 813, \sum_{i=1}^{1500} x_2^i \approx 710$$

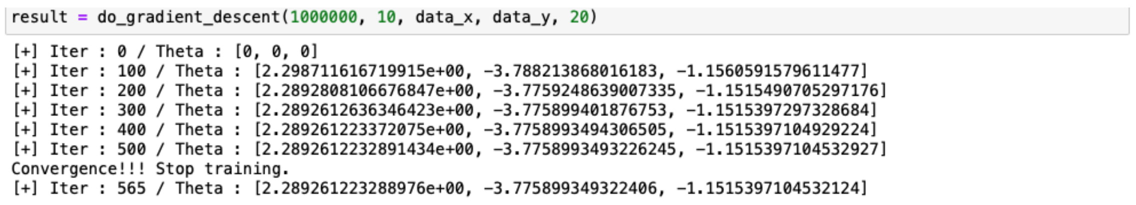
This means following equation:

$$gradient = \begin{cases} 0 < \frac{1}{1500}\sum_{i=1}^{1500}(h_\theta(x^i) - y^i)x_j^i < \frac{1}{1500}\sum_{i=1}^{1500} x_1^i & y^i = 0 \\ -\frac{1}{1500}\sum_{i=1}^{1500} x_j^i < \frac{1}{1500}\sum_{i=1}^{1500}(h_\theta(x^i) - y^i)x_j^i < 0 & y^i = 1 \end{cases}$$

This means that even if we assume that all x_j^is had the same signs, as we assumed $x_j^i > 0$, the absolute value of gradient will be less than $\frac{813}{1500}$ when updating θ_1 and will also be less than $\frac{710}{1500}$ when updating θ_2.

Also, we should not forget the fact that, we even apply α for learning rate when using gradient descent method for updating each θs. Thus, when we have $\alpha = 0.1$ as learning rate, the absolute value of each update will be less than 0.0542 for updating θ_1 and will also be less than 0.0473 for updating θ_2.

If we go towards the minimum value of our cost function, it is well known fact that the gradient will decrease. Thus, resulting slower training. To solve this, I used large value for learning rate $\alpha = 10$. With $\alpha = 10$, let's train our classifier again.



```
result = do_gradient_descent(1000000, 10, data_x, data_y, 20)

[+] Iter : 0 / Theta : [0, 0, 0]
[+] Iter : 100 / Theta : [2.298711616719915e+00, -3.788213868016183, -1.1560591579611477]
[+] Iter : 200 / Theta : [2.2892808106676847e+00, -3.7759248639007335, -1.1515490705297176]
[+] Iter : 300 / Theta : [2.289261263634642 3e+00, -3.775899401876753, -1.1515397297328684]
[+] Iter : 400 / Theta : [2.289261223372075e+00, -3.7758993494306505, -1.1515397104929224]
[+] Iter : 500 / Theta : [2.2892612232891434e+00, -3.7758993493226245, -1.1515397104532927]
Convergence!!! Stop training.
[+] Iter : 565 / Theta : [2.289261223288976e+00, -3.775899349322406, -1.1515397104532124]
```

Figure 15: Training with large learning rate

As you can see, the θs converged after iteration of 500. Which was far less than original iteration count with learning rate of 0.1.

Thus, I learned following things:

- When training logistic regression, the absolute value of update will be small. If we use learning rate that is too small, it will even slow down the training process as well. Therefore, picking a decent learning rate will accelerate training and converge all θs faster.
- Using overcomplicated model is not the key for accuracy and better results.