Shiraz University

Statistical Pattern Recognition Homework (1)

Linear & Logistic Regression

Abbas Mehrbanian[1], Reza Tahmasebi[2]

[1] abbasmehrbanian@gmail.com, StuId: 40130935

[2] saeed77t@gmail.com, StuId: 40160957

Abstract

Linear and logistic regression are supervised machine learning algorithms that differ in how they address specific problems. Linear Regression is used to handle regression problems whereas Logistic regression is used to handle the classification problems. Learning these algorithms is an essential step to enter the world of machine learning and artificial intelligence. The goal of this report is to discuss how these algorithms work and how can we implement them using python language.

1 Introduction

In this homework we're implementing Linear and Logistic regression using Python language. Linear regression is used for predicting the continuous dependent variable using a given set of independent features whereas Logistic Regression is used to predict the categorical. In this report after introducing both methods and its variants, we implement and compare them.

2 Linear regression

Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

This form of analysis estimates the coefficients of the linear equation, involving one or more independent variables that best predict the value of the dependent variable. Linear

regression fits a straight line or surface that minimizes the discrepancies between predicted and actual output values. There are simple linear regression calculators that use a "least squares" method to discover the best-fit line for a set of paired data. You then estimate the value of X (dependent variable) from Y (independent variable).[1]

2.1 Data pre-processing

Before implementing the algorithms, there are a few parts of the code which is same for all of our linear regression implementations and that is data pre-processing. Data preprocessing is the concept of changing the raw data into a clean data set. The dataset is preprocessed in order to check missing values, noisy data, and other inconsistencies before executing it to the algorithm.[2] In this section we'll discuss what's been done for data pre-processing.

In the first step we need to read and split our data into two data sets, train and test. First we'll read the data using *read_csv* function from *Pandas* library. Then we split test and train data using *train_test_split* function from *Sklearn* library. In the first argument of *read_csv* function we pass the file path and since the given data doesn't have headers we have to set *header* argument to None and to give each column a name we use *names* argument which accepts an array of strings containing the given names of columns. Then to split our data we use *train_test_split* with our data as It's first input. And we specify test data size by setting *test_size* argument to 0.3.

In the next step we need to normalize our data, in this project we used the scale to range method which means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range-usually 0 and 1 (or sometimes -1 to +1). We'll normalize both train and test data by implementing formula ((1).

$$x_{norm} = \left(\frac{x - x_{min}}{x_{max} - x_{min}}\right) \qquad (1)$$

The following formula is implemented as a function name *normalize* in *utils.py* file inside of Linear Regression folder. You'll notice in one line to the last of *normalize* function we used *reshape* function from Numpy library to reshape flat array (m,) to matrix (m, 1), where m is size of flat array, for the future algebraic calculations.

2.2 Closed-form Solution

Closed-form solutions are a simple yet elegant way to find an optimal solution to a linear regression problem. In most cases, finding a closed-form solution is much faster than optimizing with iterative optimization algorithms such as gradient descent. [3]

To calculate theta, we take the partial derivative of the MSE loss function ((2) with respect to theta and set it equal to zero. Then, do a little bit of linear algebra to get the value of theta. Which will result to equation ((3). This matter has been discussed in classroom sessions.

$$MSE = j(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 \qquad (2)$$

$$\hat{y}^{(i)} = h_\theta\left(x^{(i)}\right) = \theta^T X$$

$$\theta = (X^T X)^{-1} X^T y \qquad (3)$$

In this project, we use *Numpy* library to perform algebraic calculations. *Dot* function computes and returns dot product of two arrays.

The *inv* function which is from numpy.linalg computes and returns the (multiplicative) inverse of a matrix. The numpy array has an attribute named *T* which returns the transpose of the given numpy array. For example, x.T will return the transpose of numpy array x. Using these features we implemented equations (2) and (3) in form of functions *clac_theta*, *calc_h_theta*, *calc_cost*.

calc_theta is the implementation of (3), it receives features vector(X) and label vector(y) then calculates and returns a (2, 1) array containing the value for parameters θ_0 and θ_1.

calc_h_theta is the implementation of hypothesis function[1] which receives feature vector(X) and thetas as input and returns a vector containing values of hypothesis function.

calc_theta is the implementation of (2), it receives label vector(y) and predicted values from *calc_h_theta* and returns the cost value.

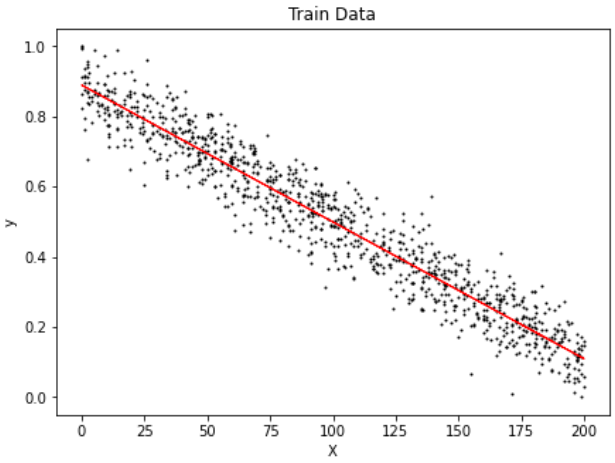

Figure 1. Plot of train data and predicted line using least squares method.

We plot our data using *pyplot* form *matplotlib* library. The result of plotting train and test data and the predicted y can is shown in Figure 1 and Figure 2.

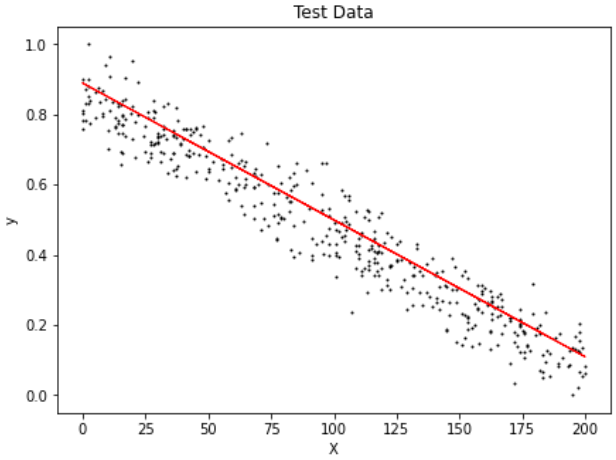

Figure 2. Plot of test data and predicted line using least squares method

The average results of running this algorithm for 10 times is shown in Table 2.

Table 1: Average results of running linear regression closed form solution for 10x times

Parameter	**Avg. value**
θ_0	0.866019919
θ_1	-0.760083107
Train cost	0.003697172
Test cost	0.004469439

The source code of these implementations can be found in *closed-form.ipynb* inside Linear Regression folder.

[1] Aka predict function.

Our decision boundary in format of y = ax + b is as follows:

$$\hat{y} = -0.760083107\ x_1 + 0.866019919$$

2.3 Gradient descent solution

In this section we'll implement a gradient descent solution for the same problem. Gradient descent is an algorithm that approaches the least squared regression line via minimizing sum of squared errors through multiple iterations. [4]

We calculate parameters using the following formula.

$$\theta_i = w_i - \alpha\ (\frac{1}{m}\sum_{i=0}^{m}\big(h_\theta\big(x^{(i)}\big) - y\big)x^{(i)}) \qquad (4)$$

We'll see later in section 3.2, implementation of logistic regression, we'll use two equations for calculating bias and weights. But here since we added a bias column to features vector in data pre-processing section there is no need to use two separate equations.

There are three common variants of gradient descent:

1. Batch gradient descent
2. Mini-batch gradient descent
3. Stochastic gradient descent

In this project we implement this two first variants and compare them to each other. The implementation can be found in *gradient-descent.ipynb* file inside of Linear Regression folder of source code.

2.3.1 Batch gradient descent

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch. [5]

We implemented Batch gradient descent as a function named *batch_gradient_descent* which accepts feature vector, labels vector, alpha[1] and number of iterations as an input and returns the final parameters as an array and also record of thetas and costs for each iteration. We implemented equation (4) to calculate our parameters.

After calculating parameters value, we plotted the train and test data and also the decision boundary which is in format of $ax + b$ and is with help of our hypothesis in (2).

You can see the decision boundary plot for test and train in Figure Figure 3 & Figure 4 and also plot of cost/iterations in Figure 5.

[1] Aka learning rate

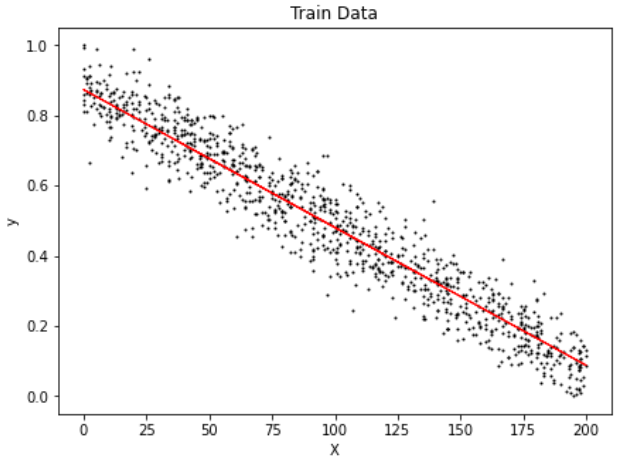

Figure 3: Plot of train data and predicted line using batch gradient descent solution

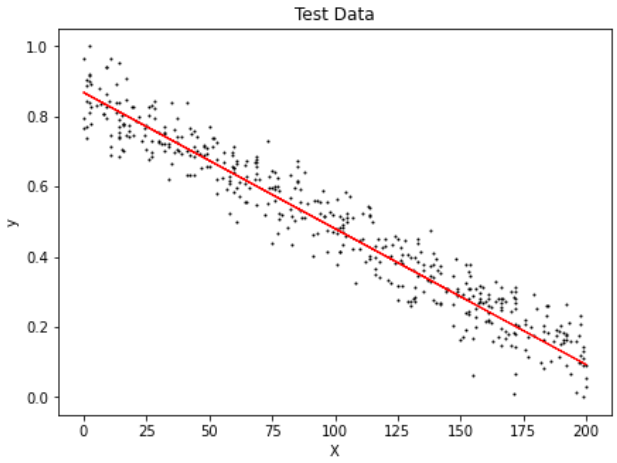

Figure 4: Plot of test data and predicted line using batch gradient descent solution

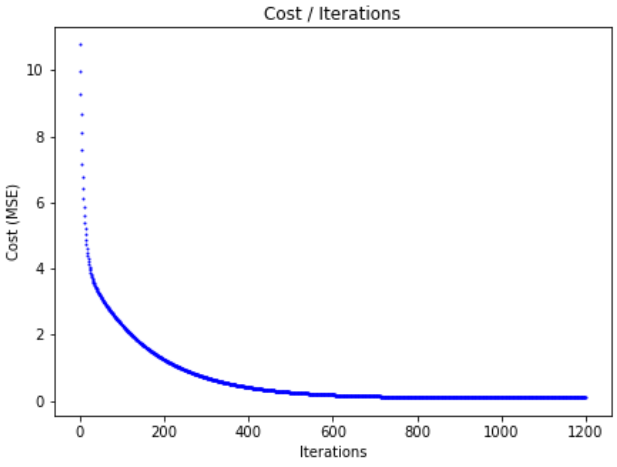

Figure 5: Plot of cost per each iteration

The average results of running this algorithm for 10 times is shown in Table 2. We set iteration to 1200 and learning rate of 0.05.

Table 2: Average results of running linear regression with batch gradient descent solution for 10x times

Parameter	Avg. value
θ_0	0.85950381
θ_1	-0.744989871
Train cost	0.003806385
Test cost	0.004380928

Our decision boundary in format of y = ax + b is as follows:

$$\hat{y} = -0.744989871\, x_1 + 0.85950381$$

2.3.2 Mini-batch gradient descent

The idea of having the whole dataset in memory to train is intractable for large datasets. In Mini-batch approach we will divide the entire data into a number of subsets. And for each subset of data, we compute the derivates for every point present in the subset and then update the parameters.[6]

We implemented Batch gradient descent as a function named *mini_batch_gradient_descent*. The implementation is pretty much same as *batch_gradient_descent* the only difference is that here we accept another parameter named *batch_size* and we separate our data set inside of inner loop according to given *batch_size* to make our batches and do the same calculations we done in 2.3.1, but this time on each mini-batch we created.

Calculating decision boundary line is same as what we done for Batch gradient descent in 2.3.1.

You can see the decision boundary plot in Figure 6 & Figure 7 and also plot of cost/iterations in Figure 8.

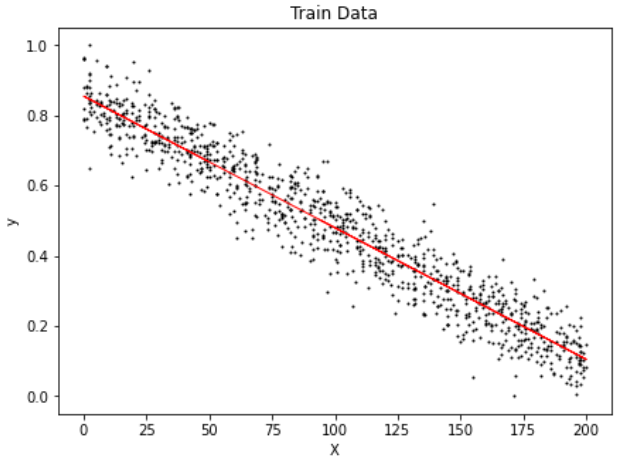

Figure 6: Plot of train data and predicted line using mini-batch gradient descent solution

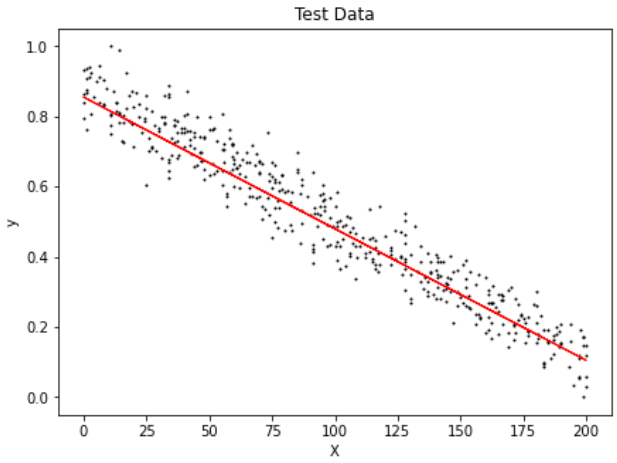

Figure 7: Plot of test data and predicted line using mini-batch gradient descent solution

Our decision boundary in format of y = ax + b is as follows. According to parameters reported in Table 3.

$$\hat{y} = -0.739103103\, x_1 + 0.852253452$$

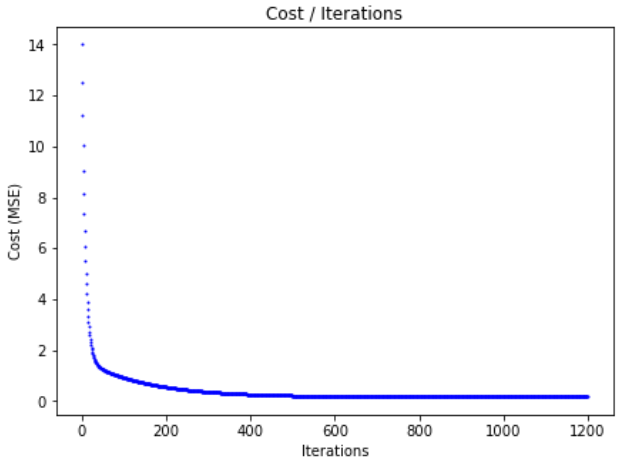

Figure 8: Plot of cost per each iteration

The average results of running this algorithm for 10 times is shown in Table 2. We set iteration to 1200, learning rate of 0.05 & batch size of 25.

Table 3: Average results of running linear regression with mini-batch gradient descent solution for 10x times

Parameter	Avg. value
θ_0	0.852253452
θ_1	-0.739103103
Train Cost	0.003759663
Test Cost	0.004906982

3 Logistic Regression

Logistic regression is the process of modeling the probability of discrete outcomes given input variables. The most common logistic regression models binary outcomes; things that can take on two values, like true/false, yes/no, etc.[7] Logistic regression is essentially a classification algorithm. The word "regression" in its name comes from its close sister in the regression domain known as linear regression.[8]

3.1 Data pre-processing

To read the data we'll again use *read_csv* function but with different inputs. First of all, the separator is different from previous problem. Data columns in this problem is separated by white spaces. So, we use a regex expression "\s+" as an input to *sep* argument of this function. The other difference is that this data set contains of 8 columns and we only need 3 of these columns to work with. So, to select desired columns we use *usecols* argument to reference the columns we want to select (which is column 3, 7 and 8). And finally, we'll use *names* argument to name these columns so we can reference them easily later.

As instructed in instruction file, we need to remove the 3rd class entities and we also changed the 1 & 2 classes into binary (0 and 1) values.

We used *train_test_split* method to split our data into test and train data sets. This time we used *stratify* argument as well. The *stratify* parameter makes a split so that the proportion of values in the sample produced will be the same as the proportion of values provided to parameter stratify. For example, here variable y is a binary categorical variable with values 0 and 1 and there are 40% of zeros and 60% of ones, stratify=y will make sure that your random split has 40% of 0's and 60% of 1's.

3.2 Algorithm implementation

In logistic regression we input our linear hypothesis function into a sigmoid function.

$$\hat{y} = h_\theta(x) = \frac{1}{1 + e^{-(\theta^T X)}} \tag{5}$$

In linear regression problem we added a column of ones as bias to our feature vector. But you may have already noticed we didn't do it in data pre-process section. As a result, we calculate bias and weights separately in our training process. Here the loss function (aka cost function) is a function that is used to measure how much our prediction differs from the labels. Binary cross entropy is the function that is used in this project for the binary logistic regression algorithm, which yields the error value.

$$J(y, \hat{y}) = \frac{-1}{m}\left(\sum_{i=1}^{m} y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})\right) \tag{6}$$

Looking at the plus sign in the equation, the left side equals 0 if y = 0, and the right side equals 0 if y = 1. In fact, this is how you measure the prediction ŷ unlike the label y, which can only be 0 or 1 in a binary classification algorithm.

$$\frac{\partial J(y, \hat{y})}{\partial w} = \frac{1}{m}(\hat{y} - y)x_i^T \tag{7}$$

$$\frac{\partial J(y, \hat{y})}{\partial b} = \frac{1}{m}(\hat{y} - y) \tag{8}$$

Now to calculate the gradients to optimize the weights using gradient descent, we need to calculate the derivative of the loss function. In other words, we need to calculate the partial derivative of the binary cross-entropy. Now that we have gradients ((7) & (8)), we can update our parameters (bias & weights) them in each epoch using (9) & (10).

$$w_i = w_i - \alpha\,(\frac{1}{m}(\hat{y} - y)x_i^T) \tag{9}$$

$$b = b - \alpha\ (\frac{1}{m}(\hat{y} - y)) \qquad (10)$$

No that we have enough knowledge about our functions we can implement them using the same tools(numpy) that we mentioned in previous sections. Full implementation can be found in *logestic-regression.ipynb* file inside Logistic Regression folder of source codes.

We implemented sigmoid function separately as a function named *sigmoid* which accepts an input and returns sigmoid function value of that input.

We implemented the logistic regression algorithm in a function named *train*. This function computes predicted values using equation (5) and then calculates parameters using equations (9) and (10). It also calculates cost using equation (6). Each epoch we save calculated cost inside of an array so we can later plot cost/iteration chart.

We also calculated cost in form of a function that accepts real and predicted values as inputs and returns the cost. We separated terms in functional implementation for better understanding, you might have also noticed we used @ operator instead of * which is an operator in python for matrix multiplications.

To plot decision boundary line $(ax + b)$ we need calculate a (slop) and b (intercept) which is calculated by the following formula. Parameter b is bias and w_0 and w_1 are our weights.

$$slope = \frac{-w_0}{w_1}\ , intercept = \frac{-b}{w_1} \qquad (11)$$

Note: *x_values* in the plotting part is our feature values upper and lower boundary.

We also implemented *decision_boundary* function which calculates and returns the decision boundary line. We used a simpler implementation in train data plot for better understanding of calculations and used the *decision_boundary* function in the test data plot.

We ran our algorithm 25000 times with learning rate of 0.25. The final average results are shown in Table 4.

You can see the decision boundary plot for test and train data in Figure Figure 9 & Figure 10 and also plot of cost/iterations in Figure 11.

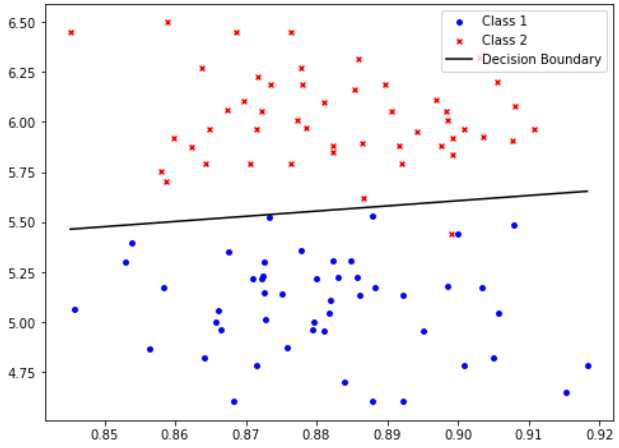

Figure 9: Plot of train data and predicted decision boundary using logistic regression

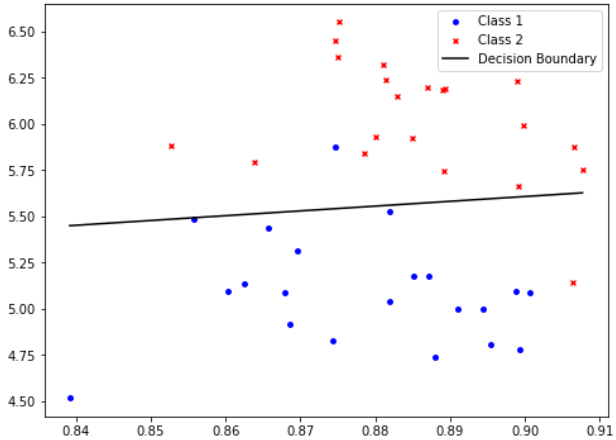

Figure 10: Plot of test data and predicted decision boundary using logistic regression

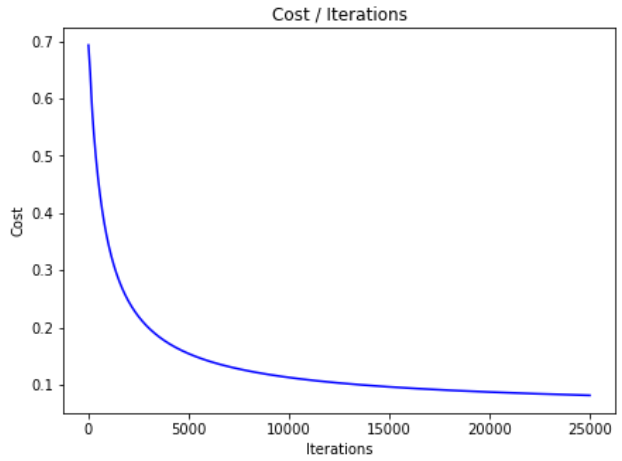

Figure 11: Plot of cost per each iteration

To calculate accuracy, we'll need the predicted labels given an input(features) so, we wrote a function named *predict* which accepts features vector, weights and bias calculated by *train* and returns the predicted values according to hypothesis function. There are entities that might have probability of 0.5, to classify such entities first we count them then we generate random classes of 0 and 1 and assign to them.

The average results of running this algorithm for 10 times is shown in Table 4.

Table 4: Average results of running batch gradient descent for 10x times

Parameter	Avg. value
$weight_0$	-19.41055889
$weight_1$	8.216118868
bias	-28.42156321
Train accuracy	96.19047619%
Test accuracy	96.93877551%
Train cost	0.13019244
Test cost	0.134739366

The accuracy is calculated by dividing number of correct predictions by number of labels and then multiplied by 100 to get percentage value.

Our decision boundary in format of y = ax + b is as follows:

$$\hat{y} = 2.362497330\ x_1 + 3.459244403$$

4 Conclusion

In the first part of this project, we worked with linear regression different solutions such as closed-form and gradient descent. We also got acquainted with gradient descent different variants and implemented two of them.

In the second part of this project, we got acquainted with the Logistic Regression and implemented it. We also learned how to calculate cost and accuracy for this particular kind of problem.

It's worth mentioning that we also tried different values for alpha (learning rate) and epochs, also different batch sizes in case of mini-batch gradient descent for linear regression, to see how these algorithms work and we finally reported the best value and their final results in the report text.

There is also a second implementation of this project existing in *alternate source codes* folder. And also, a detailed analysis of 10x times that we ran our algorithms in form of excel files inside of *Analysis* folder.

5 References

[1] IBM, "About Linear Regression," 2022. https://www.ibm.com/topics/linear-regression.

[2] K. K. Singh, M. Elhoseny, A. Singh, and A. A. Elngar, *Machine Learning and the Internet of Medical Things in Healthcare*. Academic Press, 2021.

[3] M. Hatcher, "Closed-Form Solution to Linear Regression," 2022. https://towardsdatascience.com/closed-form-solution-to-linear-regression-e1fe14c1cbef.

[4] L. Chen, "Linear regression and gradient descent for absolute beginners," 2022, [Online]. Available: https://towardsdatascience.com/linear-regression-and-gradient-descent-for-absolute-beginners-eef9574eadb0.

[5] S. Patrikar, "Batch, Mini Batch & Stochastic Gradient Descent," 2022, [Online]. Available: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a.

[6] I. Zafar, G. Tzanidou, R. Burton, N. Patel, and L. Araujo, *Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python*. Packt Publishing Ltd, 2018.

[7] T. Edgar and D. Manz, *Research methods for cyber security*. Syngress, 2017.

[8] V. N. Gudivada, M. T. Irfan, E. Fathi, and D. L. Rao, "Cognitive analytics: Going beyond big data analytics and machine learning," in *Handbook of statistics*, vol. 35, Elsevier, 2016, pp. 169–205.