

# Compute Intelligence PS s7 Lab 2

Joris Plaščinskas

September 28, 2024

## Introduction

The goal of this laboratory work is to explore ML classification problems and gradient descent algorithm. My student number is 2016020, which means that I will be using batch gradient descent and a sigmoid function. I will be exploring two datasets: Iris and Breast Cancer Wisconsin. Each data-point in iris database is a plant. Iris dataset is one of the earliest datasets (1936) used in the literature on classification methods and is a popular choice for learning ML. Each data-point is classified into one of three classes which all represent a type of iris plant. Each data-point in Breast Cancer Wisconsin dataset represents a tumor that is either benign (0) or malignant (1)

## Code

The code is split into two parts: refine data and model training and execution.

## Refining Data

**Breast Cancer Wisconsin (in total 683 data-points with 9 features)** Using code I removed unnecessary columns and rows that contained missing values and relabeled the tumor class.

**Iris (in total 500 data-points with 4 features)** One of the requirements was to create artificial data-points for Iris dataset. I did this by generating a random value for each feature between its min and max. Also since we are doing a binary classification model, I only left the 'versicolor' and 'virginica' species. The images below show how the original Iris and artificial Iris datasets are distributed.

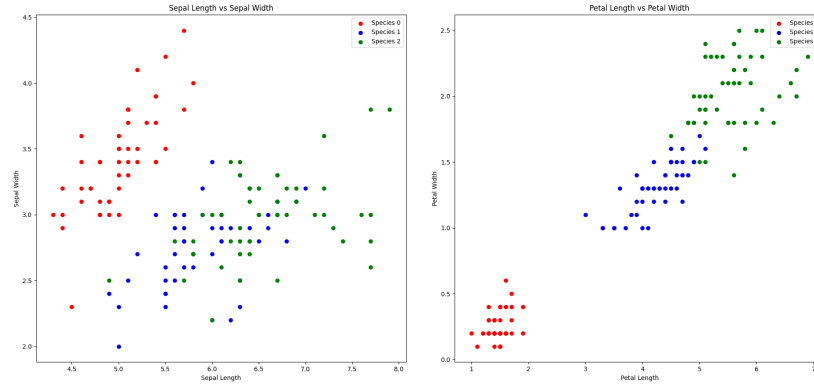


Figure 1: Iris Original Data Distribution

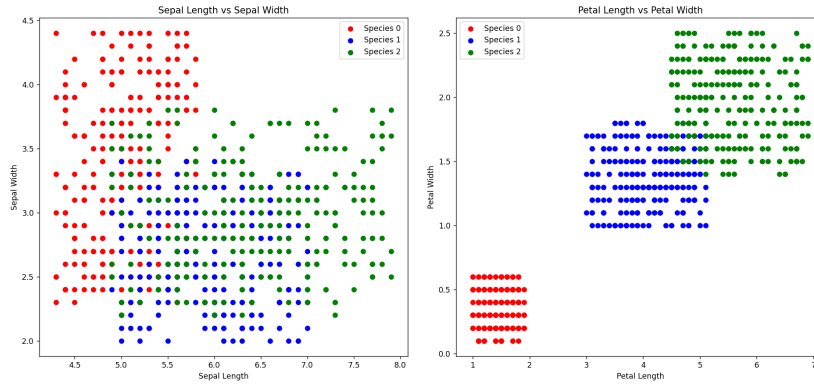


Figure 2: Iris Artificial Data Distribution

## Model Training and Execution

For this laboratory work I decided to only use 'numpy' and 'autograd' libraries, so that I could better understand the optimization methods used behind ML. I also chose not to use a sigmoid function on top of my neuron output, because that lead to very poor results.

---

```

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

# X - vertical vector of input values or multiple vertical vectors of input values that form a matrix
# W - vertical vector of input weights
# b - neuron bias float
def neuronFunction(X:np.array, W:np.array, b:float):
    return sigmoid(W.T @ X + b)

def modelFunction(X:np.array, W:np.array, b:float):
    return neuronFunction(X, W, b) # sigmoid(neuronFunction(X, W, b))

def costFunction(X:np.array, Y:np.array, model):
    return np.mean((model(X) - Y) ** 2)

def accuracyFunction(X:np.array, Y:np.array, model):
    return np.sum((np.where(model(X) > 0.5, 1, 0) - Y) == 0) / len(Y)

```

Figure 3: ML Functions

The first 80% of the data was put into the training set and the rest into the test set. I then created the accuracy and cost measuring functions for both sets and found the training set cost function derivative using 'autograd'. I initialized  $\vec{w}$  and  $b$  parameters to random values.

```

X = (data_csv.iloc[:, :-1].values).T
Y = (data_csv.iloc[:, -1].values).T

train_size = int(round(X.shape[1] * 0.8))

X_train = X[:, :train_size]
Y_train = Y[:train_size]

X_test = X[:, train_size:]
Y_test = Y[train_size:]

W = np.random.rand(X.shape[0], 1)
b = np.random.rand()

trainingSetCostFunction = lambda W, b: costFunction(X_train, Y_train, model = lambda X: modelFunction(X, W, b))
trainAccuracyFunction = lambda W, b: accuracyFunction(X_train, Y_train, lambda X: modelFunction(X, W, b))

testSetCostFunction = lambda W, b: costFunction(X_test, Y_test, model = lambda X: modelFunction(X, W, b))
testAccuracyFunction = lambda W, b: accuracyFunction(X_test, Y_test, lambda X: modelFunction(X, W, b))

trainingSetCostGrad_W = grad(trainingSetCostFunction, 0)
trainingSetCostGrad_b = grad(trainingSetCostFunction, 1)

learning_rate = 0.1

```

Figure 4: Data and Functions Preparation

Finally I wrote the gradient descent training loop, which involves calculating two gradients, because I had parameters  $\vec{w}$  and  $b$  separated.

```

for i in range(1000):
    W_grad = trainingSetCostGrad_W(W, b)
    b_grad = trainingSetCostGrad_b(W, b)
    W -= W_grad * learning_rate
    b -= b_grad * learning_rate

```

Figure 5: Training Loop

## Results

### Breast Cancer Wisconsin

The 4 graphs below show the accuracy and cost over each training epoch. These results were obtained with learning rate equals to 1 and 100 epochs. It's very visible from the bumps in the graphs that with such a high learning rate the descent isn't stable (but it works). Looking at figures 6 and 7 we can see that the accuracy graph results are very much related to the cost graph - the lower the cost is the better the model performs. Test and training cost graphs are almost identical (6 and 8), which means that the model should perform well on new data as well.

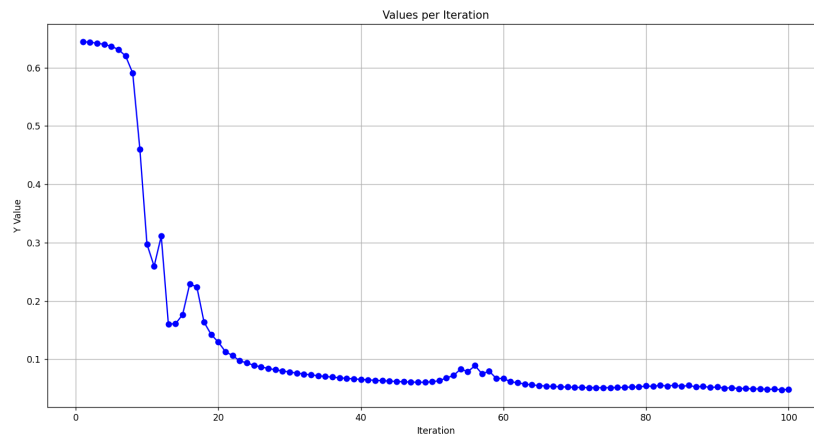


Figure 6: Training Cost

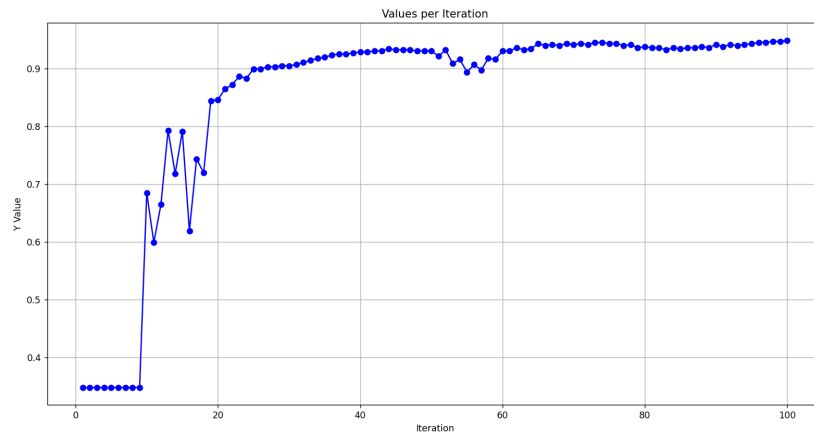


Figure 7: Training Accuracy

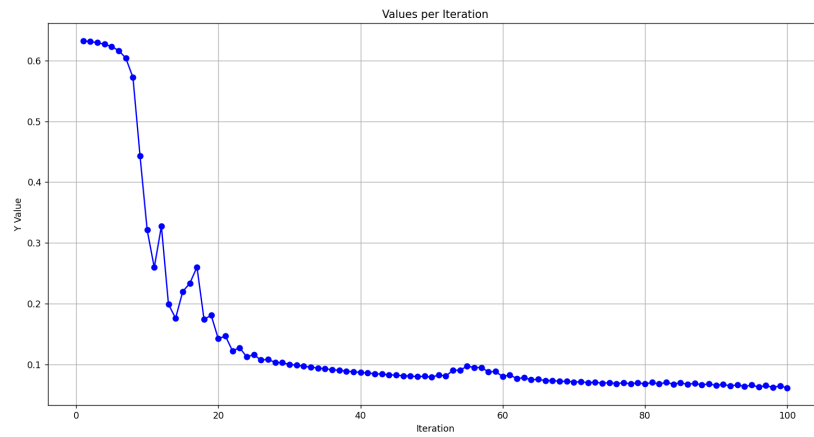


Figure 8: Test Cost

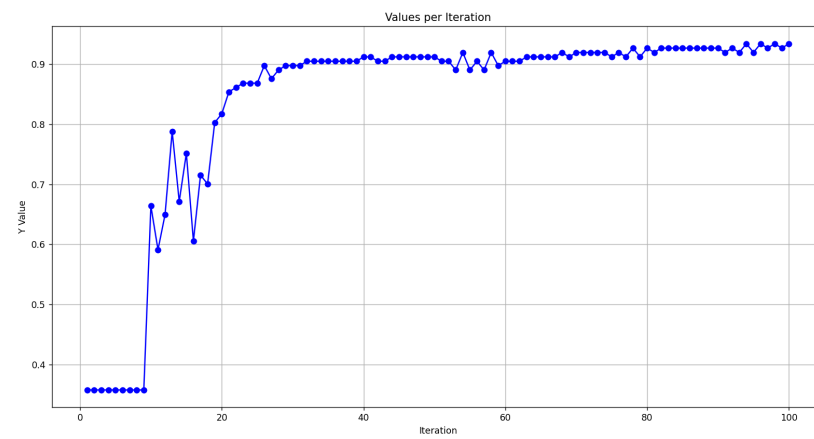


Figure 9: Test Accuracy

The graph below shows the training cost using a 0.1 learning rate and 1000 epochs. We can see

that the descent is a lot smoother.

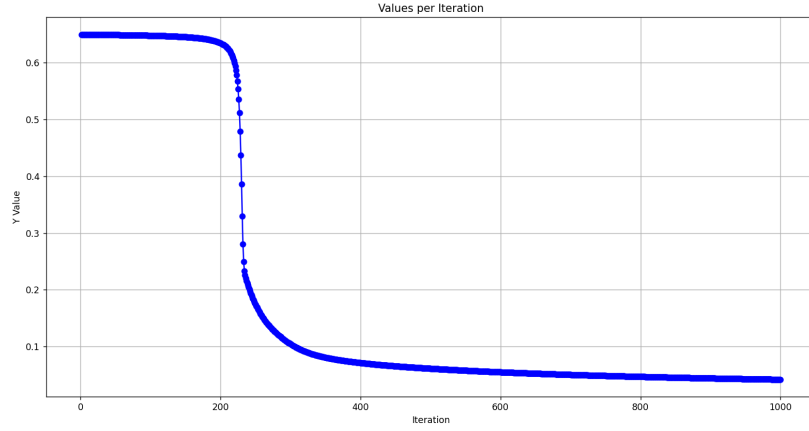


Figure 10: Low Learning Rate, Many Epochs Training Cost

Finally, the results highly depend on the initial values of  $\vec{w}$  and  $b$ , during some runs the cost function just got stuck at a local minimum or a saddle point as seen in figure-11.

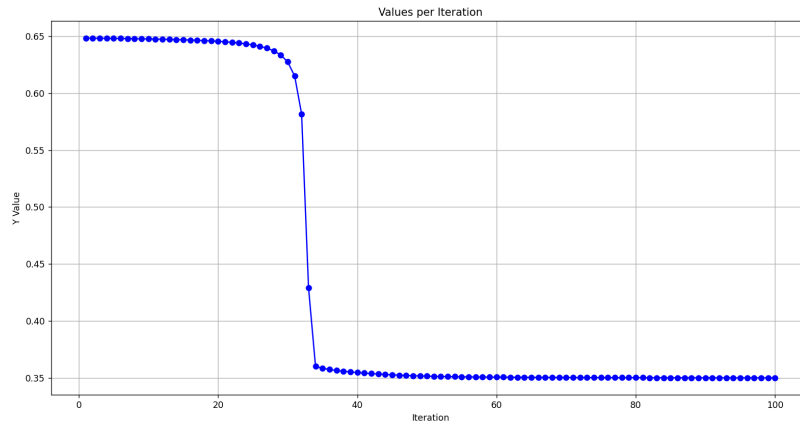


Figure 11: Failed Run

The weights and bias obtained after training with a learning rate of 1 and 100 epochs are seen in figure-12.

```

[ [-0.11610131]
  [ 0.74116477]
  [ 0.31742656]
  [ 0.17116127]
  [-0.59059225]
  [ 0.65034868]
  [-0.35032671]
  [ 0.42616729]
  [-0.1895247 ]]
-2.2943203642973655

```

Figure 12:  $\vec{w}$  and b values

## Iris

Training on Iris dataset required using a smaller learning rate than 1, because the descent was very unstable when using 1, so I ended up using 10000 epochs and a 0.001 learning rate for the Iris dataset. The 4 graphs below are very similar in shape to the Breast Cancer graphs, so the same insights apply here.

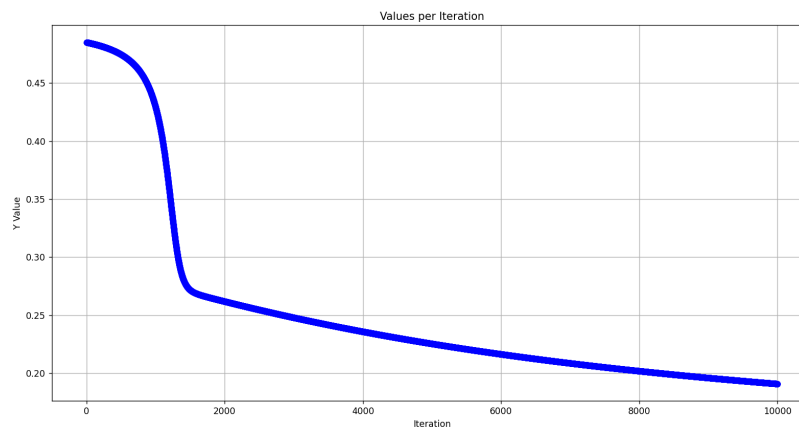


Figure 13: Training Cost

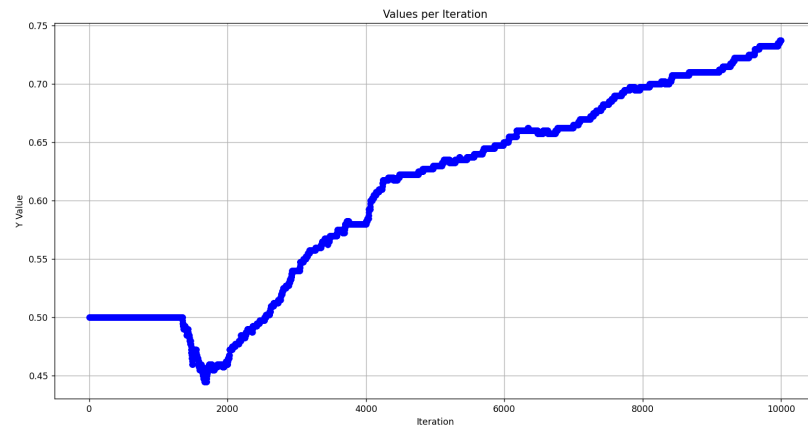


Figure 14: Training Accuracy

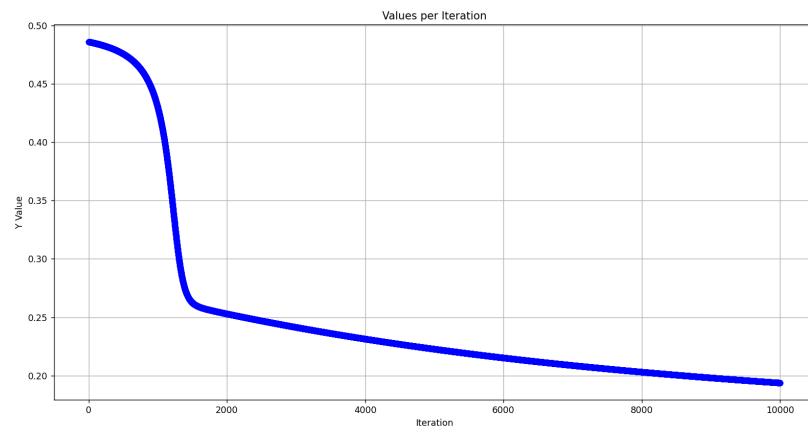


Figure 15: Test Cost

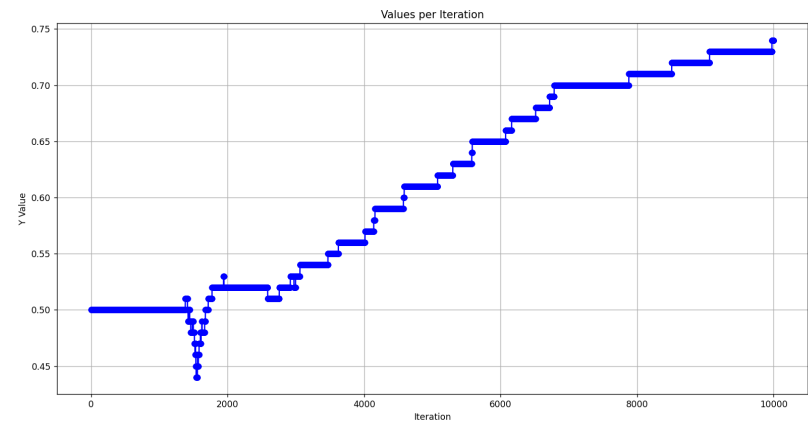
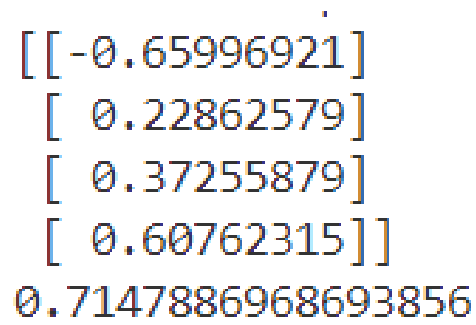


Figure 16: Test Accuracy

The weights and bias obtained using 10000 epochs and a 0.001 learning rate can be seen in figure-17.





```
[[ -0.65996921]
 [  0.22862579]
 [  0.37255879]
 [  0.60762315]]
0.7147886968693856
```

Figure 17:  $\vec{w}$  and  $b$  values

## Conclusion

After observing the results, we can state that a linear (single neuron) model with a sigmoid activation function worked very well for both the Breast Cancer and Iris datasets. Saddle points and local minimum are a big problem for the current simple implementation of batch gradient descent method, which made the final results very dependent on initial values of  $\vec{w}$  and  $b$ .