Assignment 4

Marius Aarsnes

April 2018

1 Gradient Descent (GD) and Perceptron Algorithm

In this exercise we will explore and implement gradient descent algorithm and perceptron learning. You can implement this in any language you want, but official help will only be available for implementation in Python, Java and C++. For this assignment you should consult Lecture 9 slides and Chapter 18.6 and 18.7 on the book.

2 Gradient descent

$$\text{Sigmoid:} \quad \sigma(w, x) = \frac{1}{1 + e^{w^T x}} \tag{1}$$

$$w^T x = \sum_{i=1}^{d} (w_i w_j) \tag{2}$$

$$\begin{aligned} \text{Loss Function:} \quad L_{simple}(w) = {} & [\sigma(w, [1, 0]) - 1]^2 \\ & + [\sigma(w, [0, 1])]^2 + [\sigma(w, [1, 1]) - 1]^2 \end{aligned} \tag{3}$$

2.1 Tasks

2.1.1 Plotting

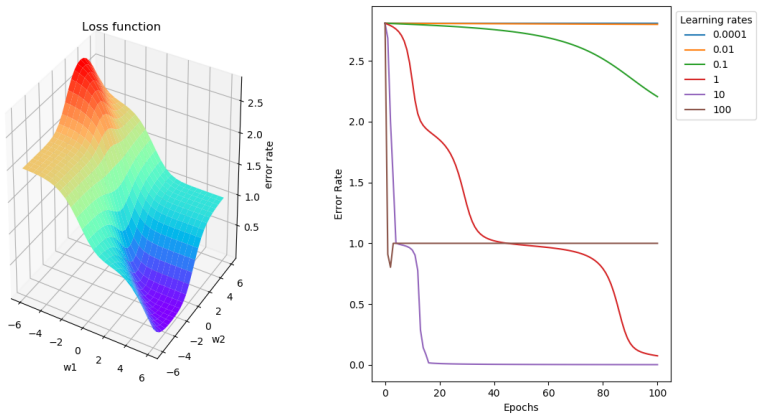

Figure 1: Loss function and learning rates

Figure 1 Shows the results from plotting the loss function withing the range of $w_i \in [-6, 6]$. From the figure, we also see that the minimum of the loss function (within the specified range) is $(w_1, w_2) = (6, -3)$ resulting in $L_{simple} = 0.0045$

2.1.2 Calculate derivative

Since $L_{simple}(w)$ consists of three separate joints we can find the derivatives of the joints separately. Therefore, since the function for $\frac{\partial L_{simple}(w)}{\partial w_i}$ will be the same for any arbitrary i, the calculations will be shown for w_i instead of both w_1 & w_2.

$$\begin{aligned}
\frac{\partial}{\partial w_i} L_{simple}(w) &= \frac{\partial}{\partial w_i}([\sigma(w,[1,0]) - 1]^2 + [\sigma(w,[0,1])]^2 + [\sigma(w,[1,1]) - 1]^2) \\
&= \frac{\partial}{\partial w_i}[\sigma(w,[1,0]) - 1]^2 + \frac{\partial}{\partial w_i}[\sigma(w,[0,1])]^2 \\
&\quad + \frac{\partial}{\partial w_i}[\sigma(w,[1,1]) - 1]^2 \\
&= 2\ [\sigma(w,[1,0]) - 1]\ \frac{\partial}{\partial w_i}[\sigma(w,[1,0]) - 1] \\
&\quad + 2\ [\sigma(w,[0,1])]\ \frac{\partial}{\partial w_i}[\sigma(w,[0,1])] \\
&\quad + 2\ [\sigma(w,[1,1]) - 1]\ \frac{\partial}{\partial w_i}[\sigma(w,[1,1]) - 1] \qquad \text{(Chain Rule)}
\end{aligned} \tag{4}$$

To get any further, we need to find the derivative of $\sigma(w, x)$:

$$\begin{aligned}
\frac{\partial}{\partial w_i}\sigma(w,x) &= \frac{\partial}{\partial w_i}(\frac{1}{1+e^{-w^T x}}) \\
&= \frac{\partial}{\partial w_i}(1+e^{-w^T x})^{-1} \\
&= -(1+e^{-w^T x})^{-2} \times (-x_i e^{-w^T x}) \\
&= x_i \frac{e^{-w^T x}}{(1+e^{-w^T x})^2} \\
&= x_i \frac{1}{1+e^{-w^T x}} \times \frac{e^{-w^T x}}{1+e^{-w^T x}} \\
&= x_i \frac{1}{1+e^{-w^T x}} \times \frac{(1+e^{-w^T x})-1}{1+e^{-w^T x}} \qquad (1-1=0) \\
&= x_i \frac{1}{1+e^{-w^T x}} \times (1 - \frac{1}{1+e^{-w^T x}}) \\
&= \underline{x_i \sigma(w,x) \times (1-\sigma(w,x))}
\end{aligned} \tag{5}$$

Using these results we can easily find an expression for $\Delta L_{simple}(w)$. First we insert what we found in equation 5 into equation 4:

$$\begin{aligned}
\frac{\partial}{\partial w_i} L_{simple}(w) = {} & 2x_i \ [\sigma(w,[1,0]) - 1] \ \sigma(w,[1,0]) \ [1-\sigma(w,[1,0])] \\
& + 2x_i \ [\sigma(w,[0,1])] \ \sigma(w,[0,1]) \ [1-\sigma(w,[0,1])] \\
& + 2x_i \ [\sigma(w,[1,1]) - 1] \ \sigma(w,[1,1]) \ [1-\sigma(w,[1,1])]
\end{aligned} \tag{6}$$

The Final result is as follows:

$$\nabla L_{simple}(w) = [\frac{\partial L_{simple}(w)}{\partial w_1}, \frac{\partial L_{simple}(w)}{\partial w_2}] \tag{7}$$

$$\begin{aligned}
\frac{\partial L_{simple}(w)}{\partial w_1} = {} & 2 \ [\sigma(w,[1,0]) - 1] \ \sigma(w,[1,0]) \ [1-\sigma(w,[1,0])] \\
& + 2 \ [\sigma(w,[1,1]) - 1] \ \sigma(w,[1,1]) \ [1-\sigma(w,[1,1])]
\end{aligned} \tag{8}$$

$$\begin{aligned}
\frac{\partial L_{simple}(w)}{\partial w_2} = {} & 2 \ [\sigma(w,[0,1])] \ \sigma(w,[0,1]) \ [1-\sigma(w,[0,1])] \\
& + 2 \ [\sigma(w,[1,1]) - 1] \ \sigma(w,[1,1]) \ [1-\sigma(w,[1,1])]
\end{aligned} \tag{9}$$

2.1.3 Apply Gradient Descent

The implementation of Gradient Descent can be found in the accompanying code in $task1.py$. The results from running Gradient Descent with different learning rates can be seen in figure 1.

I chose to start from the "worst" position as standard, in stead of a random position - $w = [-6, 3]$. Also, Gradient Descent has been run over 100 iterations. The graph shows that using learning rates of 0.0001, 0.01 and 100 show no sign of converging at a low error/loss. There are two different reasons for this.

- First of, for 0.0001 and 0.01 we see that there is little to no improvement. This is due to that fact that the learning rate simply is to small, not being able to impact the weight-change enough.
- When it comes to 100 as learning rate, we see that at first it is very promising. However, it gets stuck at 1.0. This is probably because the descent gets stuck at a plateau, unable to find where to go to get further down.

Using a learning rate of 0.1 shows that it takes some time before we see a decline in loss. This is due to the fact that the learning rate still is very small, so it takes time before we get down the slopes.

Using a learning rate of either 1 or 10 shows good results - Both reaching a fairly low error rate in 100 iterations.

3 Perceptron

$$\text{Loss function - datapoint:} \quad L_n(w, x_n, y_n) = \frac{1}{2}[\sigma(w, x_n) - y_n]^2 \tag{10}$$

$$\text{Loss function - complete dataset:} \quad L(w, D) = \frac{1}{N} \sum_{(x_n, y_n) \in D} L_n(w, x_n, y_n) \tag{11}$$

3.1 Tasks

3.1.1 Calculating Derivative

From equation 5 we get the following:

$$\begin{aligned} \frac{\partial L_n(w, x_n, y_n)}{\partial w_i} &= [\sigma(w, x_n) - y_n) \times \frac{\partial \sigma(w, x_n)}{\partial w_i}] \\ &= \underline{x_i \ [\sigma(w, x_n) - y_n] \ \sigma(w, x_n) \ [1 - \sigma(w, x_n)]} \end{aligned} \tag{12}$$

3.1.2 Implementation

The implementation can be found in *skeleton_v2.py*

3.1.3 Experiments

For the first experiment, i ran both batch- and stochastic gradient descent with the following parameters:

- $learn\_rate = 0.1$
- $niter = 200$
- $training\_num = 10$ (number of times the perceptron was trained for each data set)

Data set	Stochastic		Batch	
	Avg. Error(%)	Training Time (s)	Avg. error(%)	Training Time (s)
Big_nonsep	21.2	0.015	19.9	39.9
Big_sep	1.9	0.022	0.6	42.1
Small_nonsep	19.9	0.011	19,5	9.5
Small_sep	1.9	0.013	1.4	8.6

Table 1: Results from running gradient descent

From table 1 we see that SGD (Stochastic Gradient Descent) is much faster than BGD (Batch Gradient Descent). However, BGD performs slightly better than SGD. This makes sense when you look at the algorithms. SGD calculates

$T \times d$, while BGD calculates $T \times d \times |D_{train}$ (where T is the number of iterations, d is the dimension of a datapoint, and $|D_{train}|$ is all the training datapoints.) This not only explains why SGD is so much faster than BGD, since SGD does not depend on the size of the training set, but is also explains why BGD shows such varying Training Time. As expected the separable data sets have better results when it comes to classification. The gain from using BGD seems very small compared to SGD, *on this data set*. Based on the results it seems like one could run SGD over a larger number of iterations and this way have better performance than BGD, while still using less time.

For the second experiment i ran SGD on the large separable data set. Figure 2, below, shows the results.
Note: for some reason the label on the right would not update for the testing data, so both show a gradient of copper. However, the black and copper are the training data, while the red and blue are the testing data

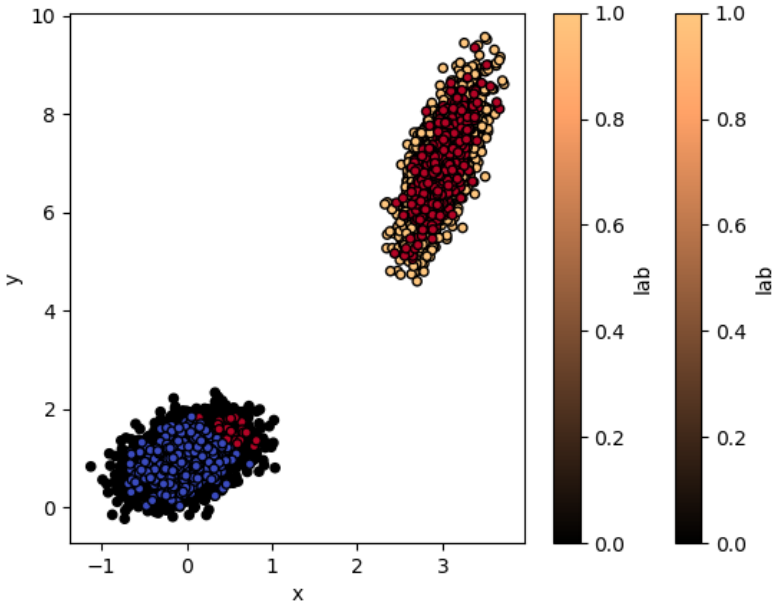

Figure 2: Experiment 2: Running SGD on a large separable data set.

The Third experiment was run on the same data set as experiment 2, also running with the SGD.

Iterations	Error (%)	Training Time (s)
10	49.8	0.007
20	49.7	0.007
50	44.3	0.007
100	03.7	0.015
200	01.7	0.02
500	00.0	0.03

Table 2: Experiment 3: Results from running SGD on a large separable data set with varying iterations

For iterations ranging from 10 to 50 show little change it either Training Time or results. A note that should be made however is that since i am using SGD, the results are from 5 different initial weights. Therefore there is a hint of uncertainty/randomness that affects the results. In general though, we see a trend of decreased error as the number of iterations are increased. We also see that when running 100+ iterations the running time starts to increase. The reason for the trend is the tweaking of weights as the training goes on. Since the input is a 2D vector ($w = (w_1, w_2)$)

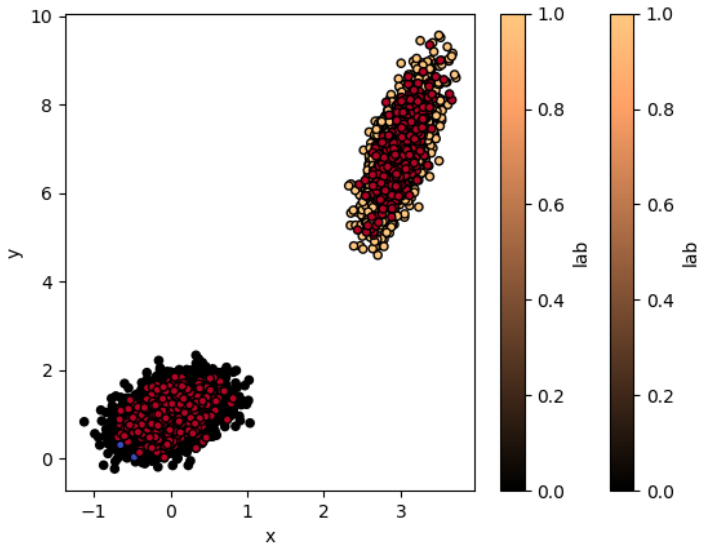

Figure 3: Experiment 3: Running 10 iterations of SGD on a large separable data set.

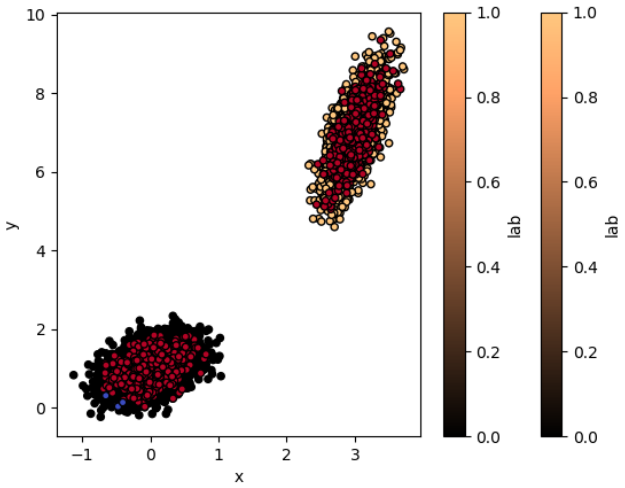

Figure 4: Experiment 3: Running 20 iterations of SGD on a large separable data set.

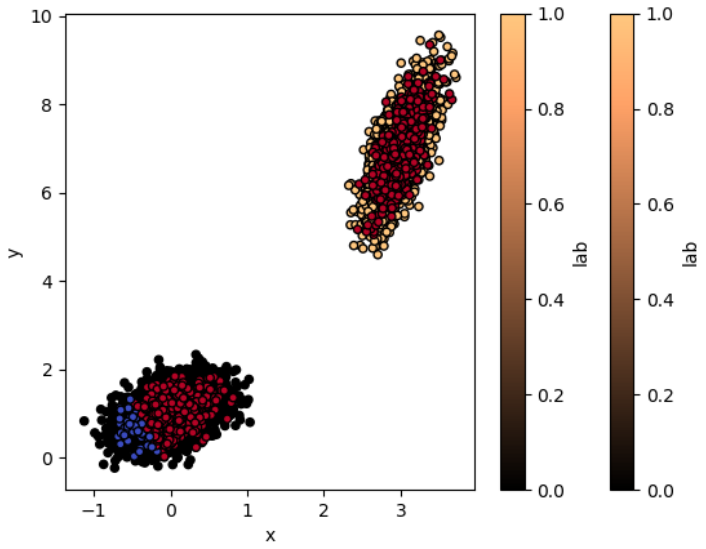

Figure 5: Experiment 3: Running 50 iterations of SGD on a large separable data set.

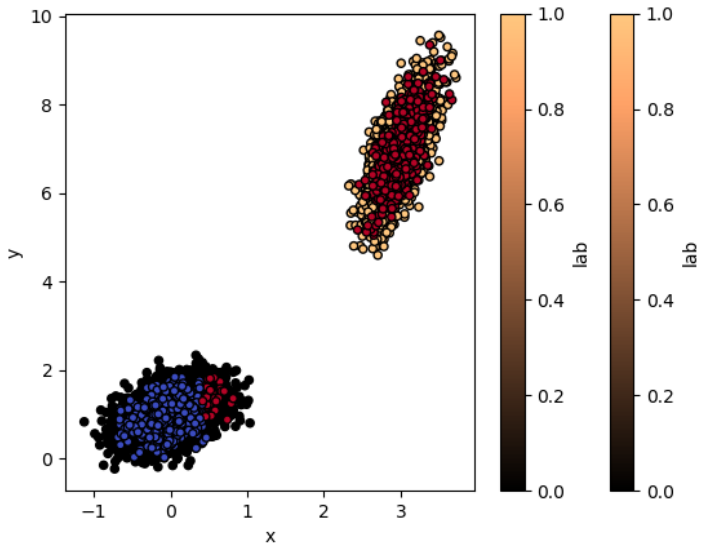

Figure 6: Experiment 3: Running 100 iterations of SGD on a large separable data set.

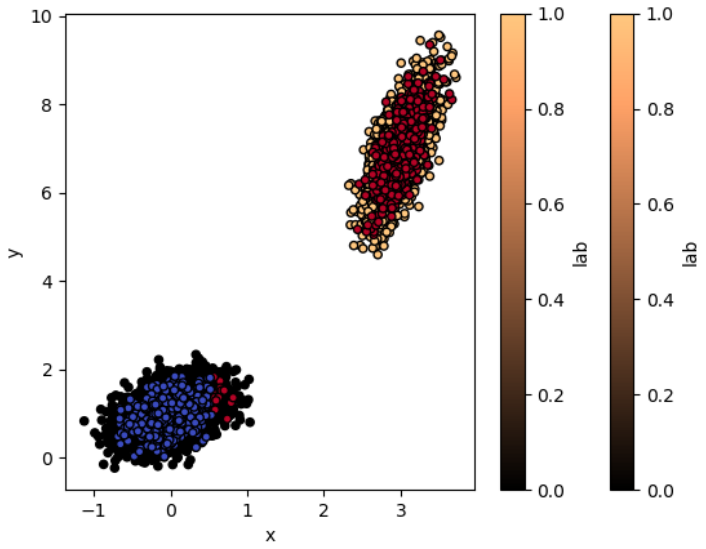

Figure 7: Experiment 3: Running 200 iterations of SGD on a large separable data set.

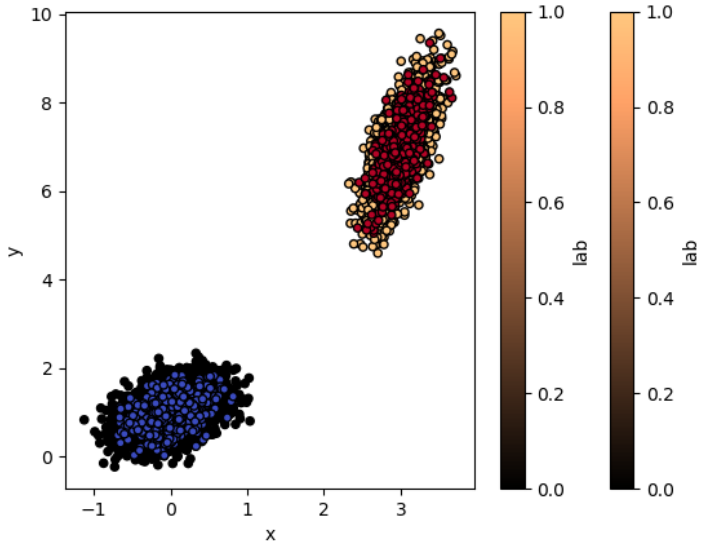

Figure 8: Experiment 3: Running 500 iterations of SGD on a large separable data set.