AMERICAN UNIVERSITY OF BEIRUT

MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

Department of Industrial Engineering & Management

ENMG 616 – Advanced Optimization Techniques & Algorithms

Assignment 3 – Logistic Regression on a real dataset

By:

Ralph Mouawad – ID 202204667

November 04th, 2023

To:

Dr. Maher Nouiehed

Please refer to the MATLAB codes submitted on Moodle.

Logistic Regression on a real dataset

Question 1 –

I stored my MATLAB file, as well as the 4 datasets provided in the same file (MATLAB) so that my code could read them. MATLAB code:

```
Xtrain = importdata('Xtrain.mat');
Xtest = importdata('Xtest.mat');
Ytrain = importdata('Ytrain.mat');
Ytest = importdata('Ytest.mat');
```

Question 2 –

To normalize the features of my data, I will use the following method on each column separately:

$$X_{normalized} = \frac{X - Xmin}{X_{max} - X_{min}}$$

Each value of Xtrain & Xtest in each column will now be normalized. The *y* values are already normalized. MATLAB code:

```
Xtrain_normal = Xtrain;
Xtest_normal = Xtest;
for i = 1:30
    a = max(Xtrain_normal(:,i));
    b = min(Xtrain_normal(:,i));
    for j = 1:length(Xtrain_normal)
            Xtrain_normal(j,i) = (Xtrain_normal(j,i) - b)/
     (a - b);
    end
end
for i = 1:30
    a = max(Xtest_normal(:,i));
    b = min(Xtest_normal(:,i));
```

```
    for j = 1:length(Xtest_normal)
            Xtest_normal(j,i) = (Xtest_normal(j,i) - b)/
     (a - b);
    end
end
min(Xtrain_normal); max(Xtrain_normal);
min(Xtest_normal); max(Xtest_normal);
```

For each of Xtrain_normal and Xtest_normal, I computed the maximum and minimum values to check they're 0 & 1. This means that all my values are now in the interval (0,1).

Question 3 –

To implement classical Gradient Descent algorithm, I must compute first the gradient of my objective function.

The objective function here is the following:

$$f(\mathbf{w},b) = \sum_{i=1}^{n} \left(-y_i \left(\mathbf{w}^T\mathbf{x}_i + b\right) + \log\left(1 + \exp\left(\mathbf{w}^T\mathbf{x}_i + b\right)\right)\right) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

$$\nabla f(w,b) = \begin{pmatrix} \partial f/\partial w \\ \partial f/\partial b \end{pmatrix} \begin{matrix} \leftarrow \text{Partial derivative with respect to } w \\ \leftarrow \text{Partial derivative with respect to } b \end{matrix}$$

$$\nabla f(w,b) = \begin{pmatrix} \sum_{i=1}^{m} \left[-y_i x_i + x_i \frac{e^{w^T x_i + b}}{1 + e^{w^T x_i + b}}\right] + \lambda w \\ \sum_{i=1}^{m} \left[-y_i + \frac{e^{w^T x_i + b}}{1 + e^{w^T x_i + b}}\right] \end{pmatrix}$$

we can also say that $\nabla f(w,b) = \sum_{i=1}^{n} \nabla f_i(w,b)$.

We computed the partial derivative of our function first with respect to **w**, and then with respect to *b*.

The gradient here is the summation of gradients at each point *i*.

I had to change some multiplications, such as $\mathbf{w}^T x$, that I replaced with x**w**, so that I don't get multiplication errors with matrix sizes. It didn't change anything since we're still multiplying each feature by its corresponding weight at each x.

MATLAB code:

```
n = 30; % number of features in X
w = zeros(n,1);
b = rand(1,1);
lambda = 0.01;
alpha = 0.1;
nb_iterations = 3000;
nb_rows = length(Xtrain);
accuracy_training = zeros(nb_iterations, 1);
for i = 1:nb_iterations
    dw = rand(n,1);
    db = 0;
    for j = 1:nb_rows
        dw = dw + (-Xtrain_normal(j,:) * Ytrain(j))' +
(Xtrain_normal(j,:)*exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b)))';
        db = db + (-Ytrain(j) + (exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b))));
    end
    % update w and b
    dw = dw + lambda*w;
    w = w - alpha*dw;
    b = b - alpha*db;
    prediction_training = (1 ./( 1 + exp(-(Xtrain_normal*w +
b))) > 0.5);
    accuracy_training(i) =
sum(prediction_training==Ytrain)/nb_rows; % see how the
accuracy will change at each iteration
end
accuracy_training(3000);
```

```
nb_rows_test = length(Xtest);
prediction_testing = (1 ./( 1 + exp(-(Xtest_normal*w + b)))
> 0.5);
accuracy_testing =
sum(prediction_testing==Ytest)/nb_rows_test;
% loss function
Loss = 0;
for i = 1:nb_rows_test
    Loss = Loss - Ytest(i)*(Xtest_normal(i,:)*w + b) + log
(1 + exp(Xtest_normal(i,:)*w + b));
end
Loss = Loss + (lambda/2)*norm(w)^2;
Loss/nb_rows_test
```

I ran the code and got as a result:

Accuracy of training = 0.968 = 96.8%

Accuracy of training = 0.913 = 91.3%

I replaced the generation of numbers by 0 and obtained the following results:

Accuracy of training = 0.984 and for testing = 0.8551. We can clearly see that the point where we start plays a role with the accuracy of our results. What also plays a role is the value of our step size, the lambda, number of iterations… We can tune the model several times to find better accuracies.

For the loss, I computed the loss function at each iteration of my testing data and summed them all. I obtained 0.6542 as an average loss. The total loss is 45.14.

When I decreased a little bit lambda and the step size, there was a better convergence (see the plotting). I also decreased the number of iterations.

I did this for the last question (the updated code is on MATLAB): I took alpha and lambda 0.001 and 1000 iterations to get a better plot.

Question 4 –

Now we have to implement the Stochastic Gradient Descent algorithm, where we evaluate the gradient at one point instead of all the sample. We will update the previous code for Gradient Descent.

MATLAB code:

```
n = 30; % number of features in X
w = zeros(n,1);
b = rand(1,1);
lambda = 0.01;
alpha = 0.1;
nb_iterations = 3000;
nb_rows = length(Xtrain);
accuracy_training = zeros(nb_iterations, 1);
for i = 1:nb_iterations
    j = randi(nb_rows); % generate a random index from the
500 rows, j being an integer not decimal
    dw = (-Xtrain_normal(j,:) * Ytrain(j))' +
(Xtrain_normal(j,:)*exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b)))';
    db = (-Ytrain(j) + (exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b))));
    % update w and b
    dw = dw + lambda*w;
    w = w - alpha*dw;
    b = b - alpha*db;
    prediction_training = (1 ./( 1 + exp(-(Xtrain_normal*w +
b))) > 0.5);
    accuracy_training(i) =
sum(prediction_training==Ytrain)/nb_rows; % see how the
accuracy will change at each iteration
end
accuracy_training(3000);% check the accuracy at the last
iteration
nb_rows_test = length(Xtest);
prediction_testing = (1 ./( 1 + exp(-(Xtest_normal*w + b)))
> 0.5);
accuracy_testing =
sum(prediction_testing==Ytest)/nb_rows_test;
% loss function
```

```
Loss = 0;
for i = 1:nb_rows_test
    Loss = Loss - Ytest(i)*(Xtest_normal(i,:)*w + b) + log
(1 + exp(Xtest_normal(i,:)*w + b));
end
Loss = Loss + (lambda/2)*norm(w)^2;
Loss/nb_rows_test;
```

I obtained the following results at the end, at the optimal solution:

Accuracy training = 98%

Accuracy testing = 86%

Average Loss = 0.2165, and total loss = 14.94

There's also a significant increase in the speed of the algorithm: With the basic Gradient Descent, it would take around 10-15 seconds to give an output. Using the Stochastic Gradient Descent algorithm, the output was given in 1 second but the accuracy of the testing data is somewhat lower so it's not really converging, especially when lowering the value of lambda.

Question 5 –

We have now to implement Stochastic Gradient Descent with diminishing step-size. I update my step size each time as 0.1/sqrt(i) instead of 0.1/I, and I also changed the value of my lambda so that my algorithm gives me a good accuracy.

```
n = 30; % number of features in X
w = zeros(n,1);
b = rand(1,1);
lambda = 0.01;
alpha = 0.1;
nb_iterations = 30000;
nb_rows = length(Xtrain);
accuracy_training = zeros(nb_iterations, 1);
for i = 1:nb_iterations
    alpha = 0.1/sqrt(i);
```

```
    j = randi(nb_rows); % generate a random index from the
500 rows, j being an integer not decimal
    dw = (-Xtrain_normal(j,:) * Ytrain(j))' +
(Xtrain_normal(j,:)*exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b)))';
    db = (-Ytrain(j) + (exp(Xtrain_normal(j,:)*w
+b)/(1+exp(Xtrain_normal(j,:)*w +b))));
    % update w and b
    dw = dw + lambda*w;
    w = w - alpha*dw;
    b = b - alpha*db;
    prediction_training = (1 ./( 1 + exp(-(Xtrain_normal*w +
b))) > 0.5);
    accuracy_training(i) =
sum(prediction_training==Ytrain)/nb_rows; % see how the
accuracy will change at each iteration
end
accuracy_training(3000);% check the accuracy at the last
iteration
nb_rows_test = length(Xtest);
prediction_testing = (1 ./( 1 + exp(-(Xtest_normal*w + b)))
> 0.5);
accuracy_testing =
sum(prediction_testing==Ytest)/nb_rows_test
% loss function
Loss = 0;
for i = 1:nb_rows_test
    Loss = Loss - Ytest(i)*(Xtest_normal(i,:)*w + b) + log
(1 + exp(Xtest_normal(i,:)*w + b));
end
Loss = Loss + (lambda/2)*norm(w)^2;
Loss/nb_rows_test;
```

I got:
Accuracy training = 91.8% and accuracy testing = 92.75%
Average error = 0.3321 with total error = 23. Here there's convergence since both accuracies are high and close to each other. Even when decreasing lambda, it didn't affect the convergence.

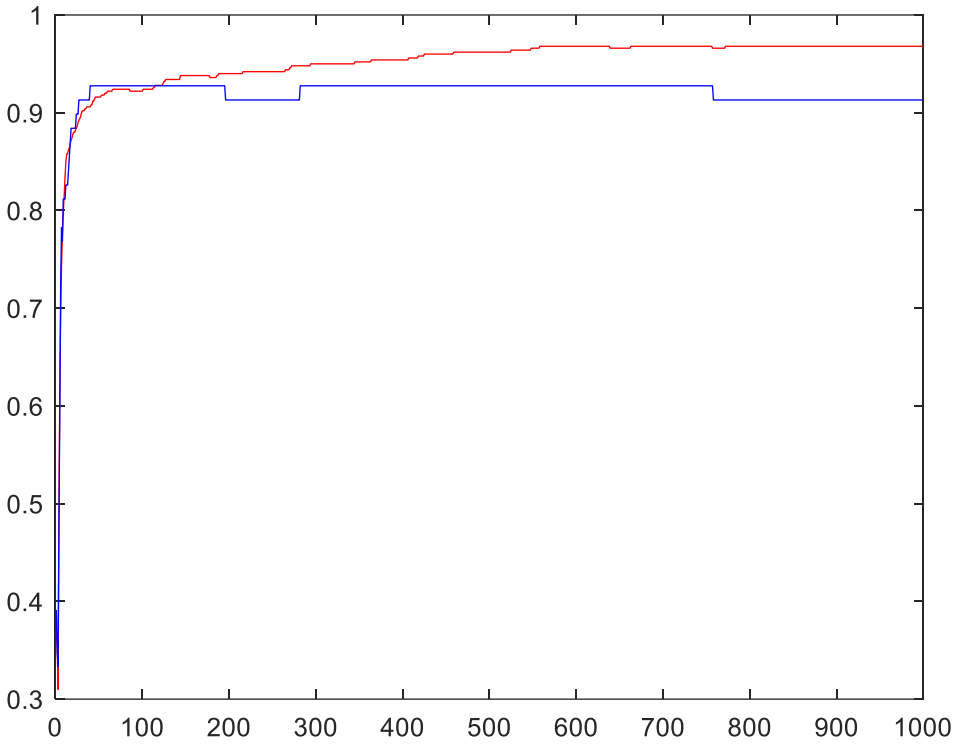
Question 6 –

For the plotting I will insert the following code into each of my previous codes and upload a picture of each.
MATLAB code –

I changed the other codes in a way that will let me plot the training data: I included computation of training accuracy in the for loop (See MATLAB codes).

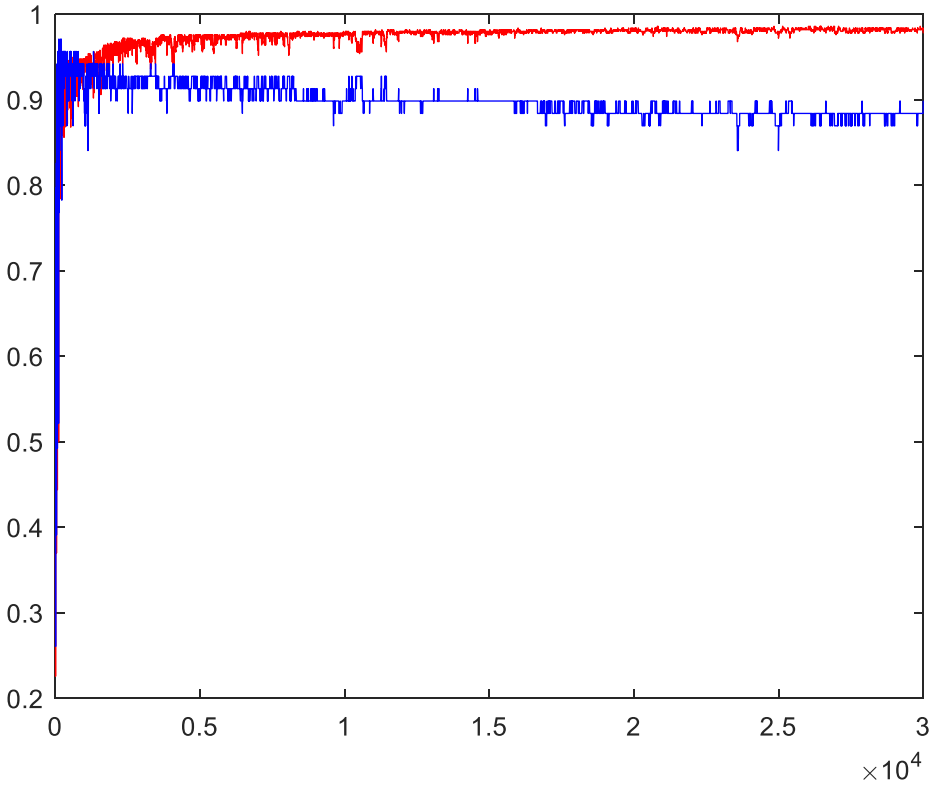
```
figure
plot(accuracy_training, 'red')
hold on
plot(accuracy_testing,'blue')
```

Classical Gradient Descent with constant step size:



Here I changed a little bit the parameters: I decreased the number of iterations, as well as the step size and lambda. I did this to get a better plot. Here we can see convergence to a certain point, but not as good as SGD with diminishing step size.

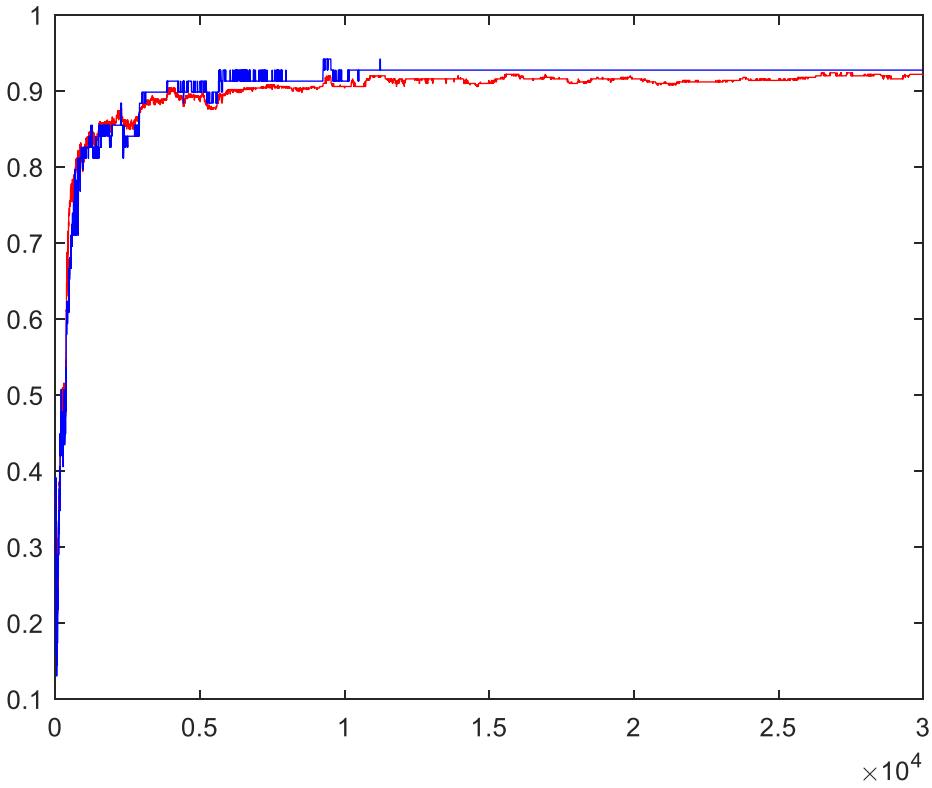
Stochastic Gradient Descent with constant step size:



Here, even though both accuracies are high, they're not the same and it's diverging. I used a bigger lambda and obtained good accuracies for both training and testing data, but in the plotting there were a lot of fluctuations. I decreased the value of lambda, and I noticed that the algorithm here is diverging. We have to choose the diminishing step size for a better convergence.

it's converging. but accuracy is than dim. step size.

Stochastic Gradient Descent with diminishing step size:



Here we can see the convergence. The accuracies of both training and testing data are high and close to each other. Even when I changed the values of lambda and step size, I could still see the strong convergence of the algorithm.

We can conclude here that diminishing step size is better for the convergence of Stochastic Gradient Descent algorithm.