Machine learning with linear and logistic regression

Adshayan Karunananthan

January 30, 2019

1 Linear regression

In this section the desired outcome is to approximate a real-valued function $y = f(x)$ with a learned linear model h(x), such that $h(x) \approx f(x)$. The linear function $h(x)$ is given by:

$$h(\mathbf{x}) = [w_0 \quad w_1 \quad \cdots \quad w_d] \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} = \mathbf{w}^T\mathbf{x} \tag{1}$$

With linear regression over all the training data this function can be obtained. Further the training data can be defined as following:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2}$$

Finally the relation between the approximation and the training data is described by the mean squared error:

$$E_{mse}(\mathbf{w}) = \frac{1}{N}(\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) \tag{3}$$

The weights $\mathbf{w}$ is found by ordinary least squares, which is basically minimizing the mean squared error.

$$\frac{\partial E_{mse}(\mathbf{w})}{\partial \mathbf{w}} = 0 \tag{4}$$

$$\rightarrow \quad \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{5}$$

1.1 Implementing linear regression with ordinary least squares on a 2d training set

Linear regression on a 2-dimensional training set was implemented with ordinary least squares(OLS) in MATLAB. The MATLAB code is presented in in section

3.1. The implementation of the OLS from equation 5 can be found in codeline 20. The resulting model, $h(\mathbf{x})$, became

$$h(\mathbf{x}) = \begin{bmatrix} 0.2408 & 0.4816 & 0.0586 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} = 0.2408 + 0.4816x_1 + 0.0586x_2 \quad (6)$$

These values gave mean square error of 0.0104 and 0.0095 on the training set and test set, respectively. The model generalize quite good, regarding the low mean square errors. Especially considering that the errors were lower for the test data than the training data.

1.2 Implementing linear regression with ordinary least squares on an 1d training set

In this section OLS is implemented on an 1-dimensional training set. The resulting approximation was:

$$h(\mathbf{x}) = \begin{bmatrix} 0.1956 & 0.6129 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \end{bmatrix} = 0.1956 + 0.6129x_1 \quad (7)$$

The MATLAB code for the implementation can be found in 3.2. The linear regression function is presented with both the training data and test data in figure 1 and figure 2, respectively.

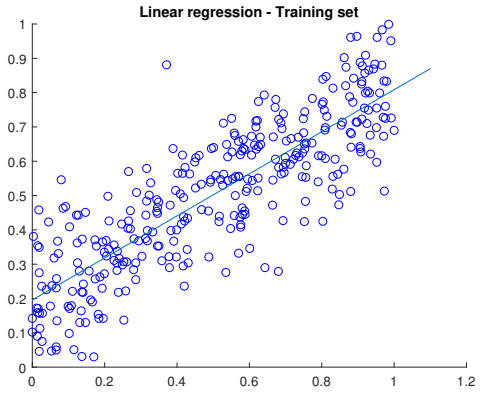

Figure 1: Linear regression on a 1d set - Training set

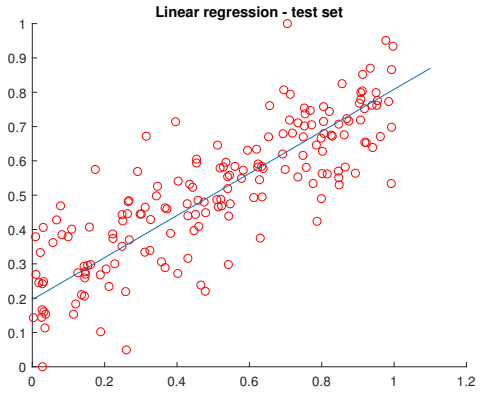

Figure 2: Linear regression on a 1d set - Test set

As can be observed from the figures the line-fitting fits the data quite well. On the other hand there are still several points that deviates from the line. In fact they are spread quite evenly on both sides, thus I would say that the curve fitting is satisfactory.

2 Logistic regression

Logistic regression is used for binary classification. In fact it is a model that fits a probability distribution $Pr(y|\mathbf{x})$ where $y \in \{0, 1\}$ and $\mathbf{x}$ is the input. In example this can be used to predict whether a mail is a spam or not, where the mail is the input and the classification is the output. The objective is to find a linear function h(x) as in equation 1, which predicts the true and the false outputs based on the input. To get good predictions the weight parameters have to be trained. Contrary from the linear regression the probability in this case can be captured by a nonlinear logistic function, also known as a sigmoid function.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad where \quad z = h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \tag{8}$$

As can be seen from the sigmoid function the output is between 0 and 1. In fact when z is large the sigmoid function is high and opposite when z is low. Hence the probability distribution can be described as following:

$$Pr(y \mid \mathbf{x}) = \sigma(z)^y(1 - \sigma(z))^{1-y} = \sigma(\mathbf{w}^T\mathbf{x})^y(1 - \sigma(\mathbf{w}^T\mathbf{x}))^{1-y} \tag{9}$$

In logisitc regression the weights are learned by maximizing the likelihood over all N training examples.

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} Pr(y_i \mid \mathbf{x}_i) = \prod_{i=1}^{N} \sigma(z)^{y_i}(1 - \sigma(z))^{1-y_i} \tag{10}$$

To get rid of the exponential one can take the logarithm of the likelihood, in equation 10, and get the log-likelihood. Since the logarithm is monotonically increasing the maximum will remain at the same $\mathbf{w}$. Moreover to get it on the

cross-entropy error form the negative log-likelihood is used. To clarify, minimizing the negative log-likelihood is the same as maximizing the log-likelihood. In addition to that we will also divide it by the number of training examples to scale it and the resulting error function we want to minimize becomes:

$$E_{ce}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i ln(\sigma(z)) + (1-y_i)ln(1-\sigma(z))) \tag{11}$$

As this equation do not have a closed form solution the solution has to be approximated nummerically. This can be done by the optimization algorithm gradient descent.

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \alpha\frac{\partial E_{ce}(\mathbf{w})}{\partial \mathbf{w}} \tag{12}$$

where α is the learning rate and $\mathbf{w}(k)$ is the model parameter at iteration k. After the derivations the final update rule becomes

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \alpha\sum_{i=1}^{N}((\sigma(\mathbf{w}(k)^T\mathbf{x}_i) - y_i)\mathbf{x}_i) \tag{13}$$

2.1 Implementing logistic regression on training set 1

In this section logistic regression, with gradient-descent, is used to classify the data in CLdata1. The initial weight used in this problem was $\mathbf{w}^T = [0 \quad 0 \quad 0]$ and the learning rate was 0.1. The result of the training after 1000 iterations is illustrated in the following figures.

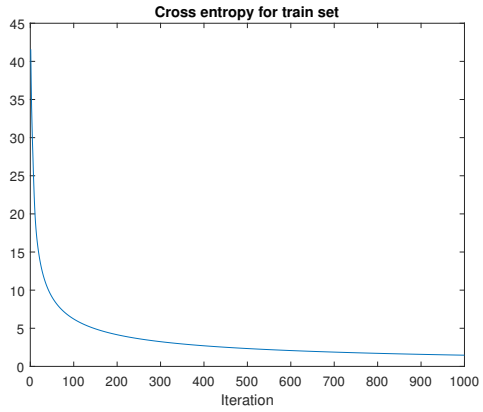

Figure 3: Logistic regression - Cross entropy on training set

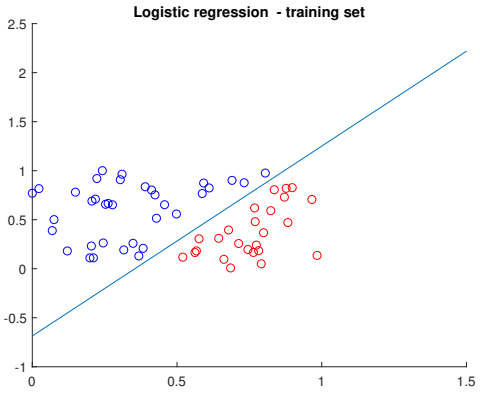

Figure 4: Logistic regression - Training set

As can be observed from the figure **??** the linear function classifies the training data very well. All the blue dots is on side and the red dots on the other. In addition to that figure **??** shows that the cross-entropy is decreasing for each iteration, meaning that the weight suits better and better for each iteration, and converging to a low number in the end, which means that the linear function classifies the data quite well.

Next part is to apply this to a test set and see if it is able to classify the test data. The result is showed in the following figures.

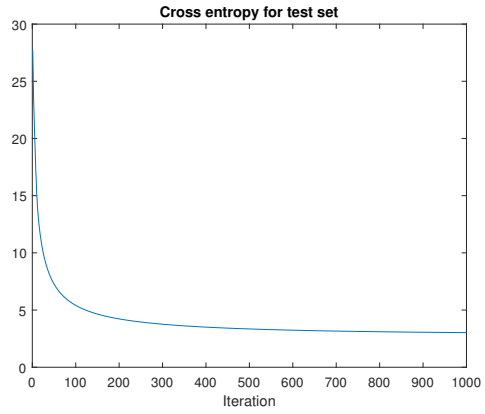

Figure 5: Logistic regression - Cross entropy on training set

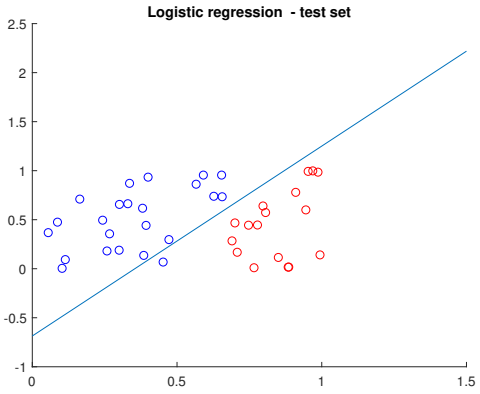

Figure 6: Logistic regression - Training set

The linear function is almost able to classify all the test data correctly. As can be seen in figure 6 only one of the blue dots was not classified correctly. Figure 5 shows that the cross-entropy reduces much in the start and converges to a relatively low number after around 300 iterations. Different from the test data the cross-entropy here is higher than the test data, which is obvious regarding that the line did not classify all the data correctly. The MATLAB-code written to solve this problem is listed in section 3.3.

2.2 Implementing logistic regression on training set 2

In this section the same logistic regression implementation from previous section is used on a new dataset. The resulting decision boundary with training data and test data is illustrated in the following figures.

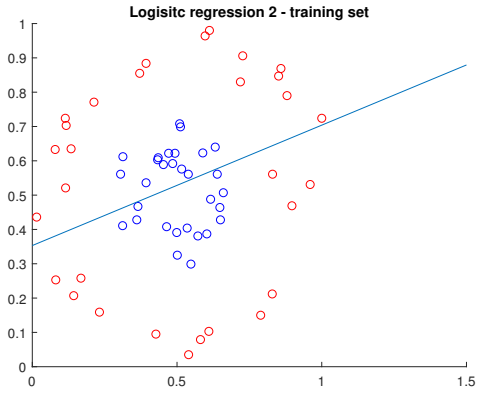

Figure 7: Logistic regression - Training set 2

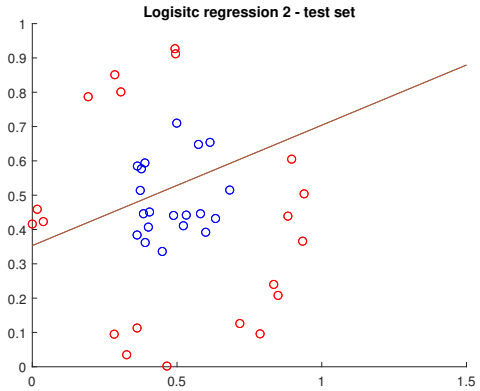

Figure 8: Logistic regression - test set 2

As can be observed from figure 7 and figure 8 the logistic regression performs very poorly to classify the data. In fact it is not possible to classify the data with a linear function as described in equation 1 due to the non-linearity. Hence a nonlinear boundary have to be used in the logistic regression to classify the data.

2.2.1 Implementing logistic regression with nonlinear boundary

From intuition it can be observed from the data that the data can be classified with a circular decision boundary, more precisely an ellipsoid can do the job. Therefore, the linear function z in equation 8 must be replaced by a nonlinear function which describes an ellipsoid. Consequently Z becomes:

$$z = [w_0 \quad w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5] \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \tag{14}$$

As z is changed the update rule in gradient descent from equation 13 also have to be changed. Finally from equation 12 the update rule is derived and becomes:

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \alpha \sum_{i=1}^{N} ((\sigma(\mathbf{w}(k)^T \mathbf{x}_i) - y_i) \frac{\partial z(\mathbf{w})}{\partial \mathbf{w}}) \tag{15}$$

$$\frac{\partial z(\mathbf{w})}{\partial \mathbf{w}}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \tag{16}$$

The resulting decision boundary after 1000 iteration with learning rate 0.1 is illustrated in the following figures.

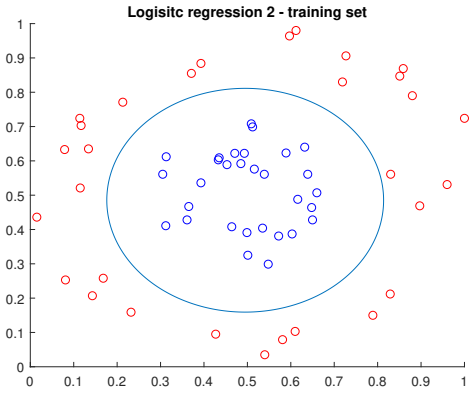

Figure 9: Logistic regression with nonlinear boundary - Training set 2

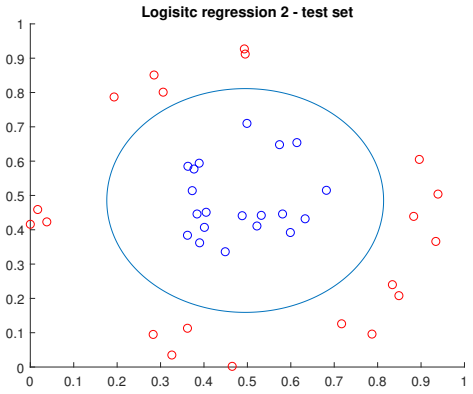

Figure 10: Logistic regression with nonlinear boundary - test set 2

The MATLAB code for the implementation can be found in 3.4. As can be observed from the figures the decision boundary classifies the data quite well. In fact all the data in both the training data and test data is classified correctly, thus I would say that the logistic regression works satisfactory.

3 Appendix

3.1 MATLAB code - Implementing linear regression with ordinary least on a 2d training set

```
clear all
load test_2d_reg_data.csv
load train_2d_reg_data.csv

reg_test =  test_2d_reg_data;
```

```
reg_train =  train_2d_reg_data;

dimension = size(reg_train,2)-1;
trainSize = size(reg_train,1);
testSize = size(reg_test,1);

X_train = reg_train(:, 1:dimension);
X_train = [ones(trainSize,1), X_train]; % add ones for
    the bias term to get it on general form
y_train = reg_train(:, dimension+1);

X_test = reg_test(:, 1:dimension);
X_test = [ones(testSize,1), X_test]; % add ones for the
    bias term to get it on general form
y_test = reg_test(:, dimension+1);

W = pinv(X_train'*X_train)*X_train'*y_train;  %pinv =
    pseudoinvers to circumvent the singular requirement on
     X^T*X

MSE_train = 1/trainSize*(X_train*W-y_train)'*(X_train*W-
    y_train); % Mean Square Error for the training set

MSE_test = 1/testSize*(X_test*W-y_test)'*(X_test*W-y_test
    ); % Mean Square Error for the test set
```

3.2 MATLAB code - Implementing linear regression with ordinary least on a 1d training set

```
clear all
load test_1d_reg_data.csv
load train_1d_reg_data.csv

reg_test =  test_1d_reg_data;
reg_train =  train_1d_reg_data;

dimension = size(reg_train,2)-1;
trainSize = size(reg_train,1);
testSize = size(reg_test,1);

X_test = reg_test(:, 1:dimension);
X_test = [ones(testSize,1), X_test];
y_test = reg_test(:, dimension+1);

X_train = reg_train(:, 1:dimension);
X_train = [ones(trainSize,1), X_train];
y_train = reg_train(:, dimension+1);

W = pinv(X_train'*X_train)*X_train'*y_train;  %pinv =
    pseudoinvers to circumvent the singular requirement on
```

```
    X^TX
21
22 %% plot
23 x1 = 0:0.01:1.1;
24 x2 = W(1) + W(2).*x1;
25
26 figure(1)
27 hold on
28 % training set
29 for j = 1:length(y_train)
30     plot(X_train(j,2), y_train(j), 'ob')
31 end
32 plot(x1,x2)
33 title('Linear regression - Training set')
34 hold off
35
36 figure(2)
37 hold on
38 % test set
39 for j = 1:length(y_test)
40     plot(X_test(j,2), y_test(j), 'or')
41 end
42 plot(x1,x2)
43 title('Linear regression - test set')
44 hold off
```

3.3 MATLAB code - Implementing logistic regression with linear boundary

```
1  clear all
2  load cl_train_1.csv
3  load cl_test_1.csv
4
5  cl_test = cl_test_1;
6  cl_train = cl_train_1;
7
8  x1_test = cl_test(:,1);
9  x2_test = cl_test(:,2);
10 y_test = cl_test(:,3);
11
12
13 x1_train = cl_train(:,1);
14 x2_train = cl_train(:,2);
15
16 y_train = cl_train(:,3);
17 w = zeros(1,size(cl_train,2))';
18
19 iterations = 1000;
20 alpha = 0.1; %learning rate
21 X_train = [ones(length(cl_train),1) cl_train(:,1:2)];
```

```
X_test = [ones(length(cl_test),1) cl_test(:,1:2)];

cross_entropy_train = zeros(1,iterations);
cross_entropy_test = zeros(1,iterations);
for k = 1:iterations
    %Calculating the Cross Entropy
    sum1 = [0;0;0];
    element_tr = (sigmf(X_train*w, [1,0]).^y_train).*((1
        - sigmf(X_train*w, [1,0])).^(1-y_train));
    element_tr = log(element_tr);
    neg_likelihood_train = - sum(element_tr);
    cross_entropy_train(k) = neg_likelihood_train;

    element_te = (sigmf(X_test*w, [1,0]).^y_test).*((1 -
        sigmf(X_test*w, [1,0])).^(1-y_test));
    element_te = log(element_te);
    neg_likelihood_test = -sum(element_te);
    cross_entropy_test(k) = neg_likelihood_test;

    % Gradient descent
    for i = 1:length(y_train)
        x_curr = [1 cl_train(i,1:2)]';
        y_curr = y_train(i);
        der_CE = sigmf(w'*x_curr,[1 0])-y_curr ;  %CE =
            cross entropy
        der_h = x_curr;
        sum1 = sum1 + der_CE*x_curr;
    end
    w = w-alpha*sum1;
end
```

3.4 MATLAB code - Implementing logistic regression with linear boundary

```
clear all
load cl_train_2.csv
load cl_test_2.csv

cl_test = cl_test_2;
cl_train = cl_train_2;

x1_test = cl_test(:,1);
x2_test = cl_test(:,2);
y_test = cl_test(:,3);


x1_train = cl_train(:,1);
x2_train = cl_train(:,2);

y_train = cl_train(:,3);
```

```
17 w = zeros(1,size(cl_train,2)+2)';
18
19 iterations = 1000;
20 alpha = 0.1; %learning rate
21
22 for k = 1:iterations
23     sum = [0;0;0;0;0];
24     for i = 1:length(y_train)
25         x_curr = [1 cl_train(i,1:2)]';
26         y_curr = y_train(i);
27         z=w(1)*x_curr(1) + w(2)*x_curr(2) + w(3)*x_curr
               (3) + w(4)*x_curr(2).^2 + w(5)*x_curr(3).^2;
28         der_CE = sigmf(z,[1 0])-y_curr ;
29         der_h = [x_curr(1);x_curr(2);x_curr(3);x_curr(2)
               .^2;x_curr(3).^2];
30         sum = sum + der_CE*der_h;
31     end
32     w = w-alpha*sum;
33 end
34
35 %% plot of decision boundary
36 f = @(x,y) w(1)+w(2)*x + w(3)*y + w(4)*x.^2 + w(5)*y.^2;
37 fimplicit(f);
```