ML Laboratory 02: Logistic Regression

1. Objective

Students should understand and be able use a logistic regression model in Matlab

2. Theoretical aspects

Logistic regression is a widely used model for estimating **probabilities** (e.g. values between 0 and 1). It is also widely used for **binary classification**.

Logistic regression: the model

We have an input vector x and a predicted value y:

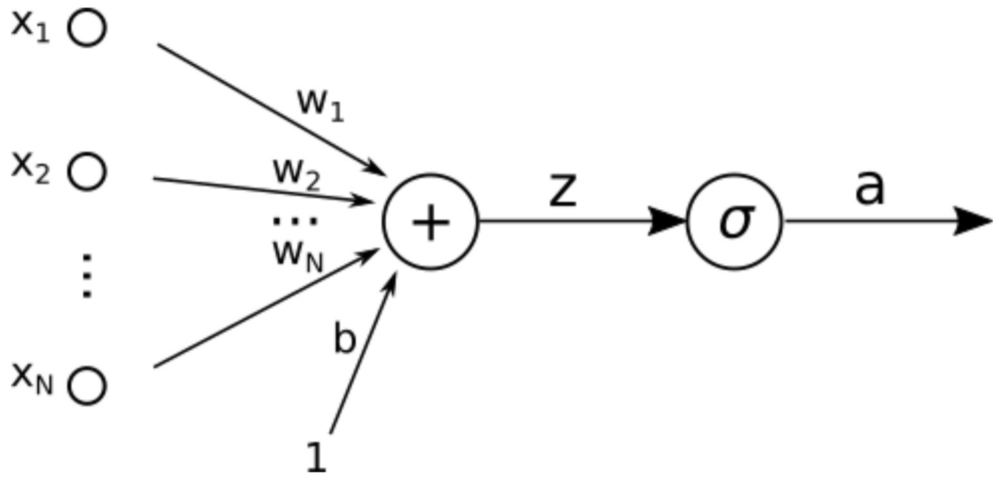
$$X = [\, x_1 \quad x_2 \quad \dots \quad x_N \,] \rightarrow y$$

The logistic regression model: the output is assumed to be a the **sigmoid function** applied to a **linear combination** of the inputs. The sigmoid function is also known as the **logistic function**, hence the name.

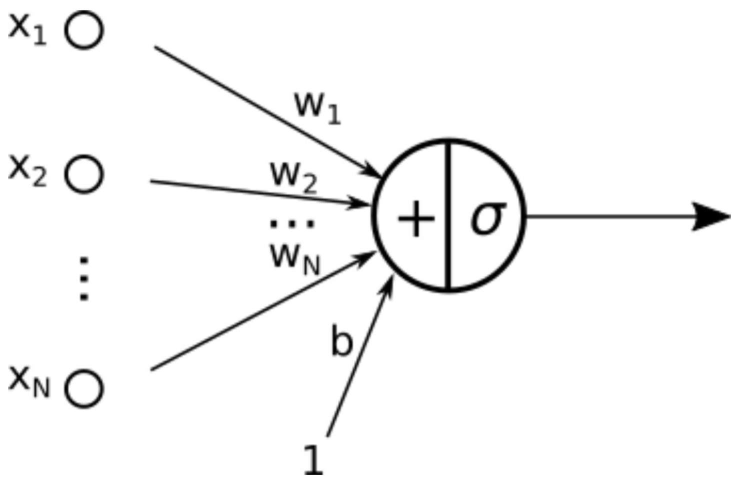
$$y \approx \frac{1}{1+e^{-w_1x_1-w_2x_2-\ldots-w_Nx_N-b}}.$$

This can be understood as a **sequence of two steps**:

1. Compute a **linear combination** z of the inputs:
$$z = w_1x_1 + w_2x_2 + \ldots + w_Nx_N + b$$
2. Pass z through the **sigmoid** function $\sigma(z)$:
$$y = \frac{1}{1+e^{-z}}$$

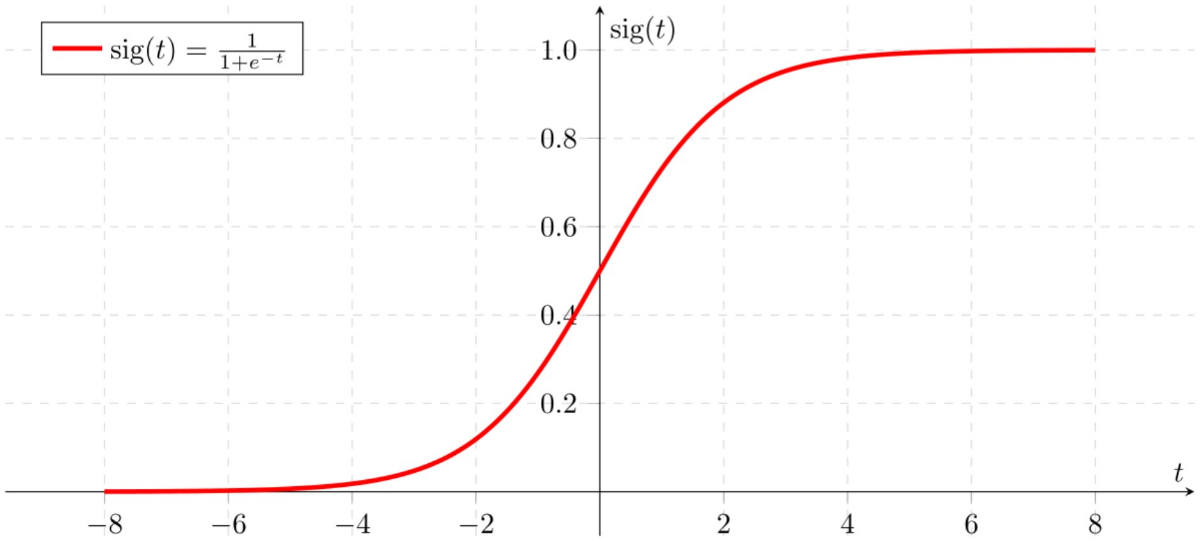


Both steps can be represented asn one "neuron" like this:



The sigmoid function

Let's take a look at the sigmoid function σz:



Notes:

- The output value is always between 0 and 1. This makes the result good for modeling **probabilities**, but not good for other types of numeric values.
- The sigmoid can be understood as follows:
  - when z is much bigger than 0, the result is practically 1
  - when z is much smaller than 0, the result is practically 0
  - when z is around 0, there is a "transition zone" from 0 to 1. In particular, when $z = 0$, the output is right at the middle, $\sigma(0) = 0.5$.

This makes it similar to classification: if z much larger than 0, data belongs to class 1; if z much smaller than 0, data belongs to class 0; $z = 0$ is the frontier. Around the frontier, we have less confidence in the classification (a sort of "gray area").

Logistic regression: the parameters

The parameters of the logisticr regression model are the **weights** $w_1, w_2, \dots w_N$ and the **bias** value b (also known as the **intercept**). This is similar to the linear regression.

In a similar way to linear regression, we can consider b as just another weight w_i which multiplies a constant input of 1. In this way, we can compute z as the inner product of two vectors:

$$\begin{bmatrix} x_1 & x_2 & \dots & w_N & b \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \\ b \end{bmatrix} = z$$

Cost function (loss function)

Any cost function can be chose (the cost function can always be chosen independent of the model). However, because the outputs are typically understood as **probabilities**, we should use a distance function which is appropriate to probabilities: the cross-entropy (also known as the Kullback-Leibler distance).

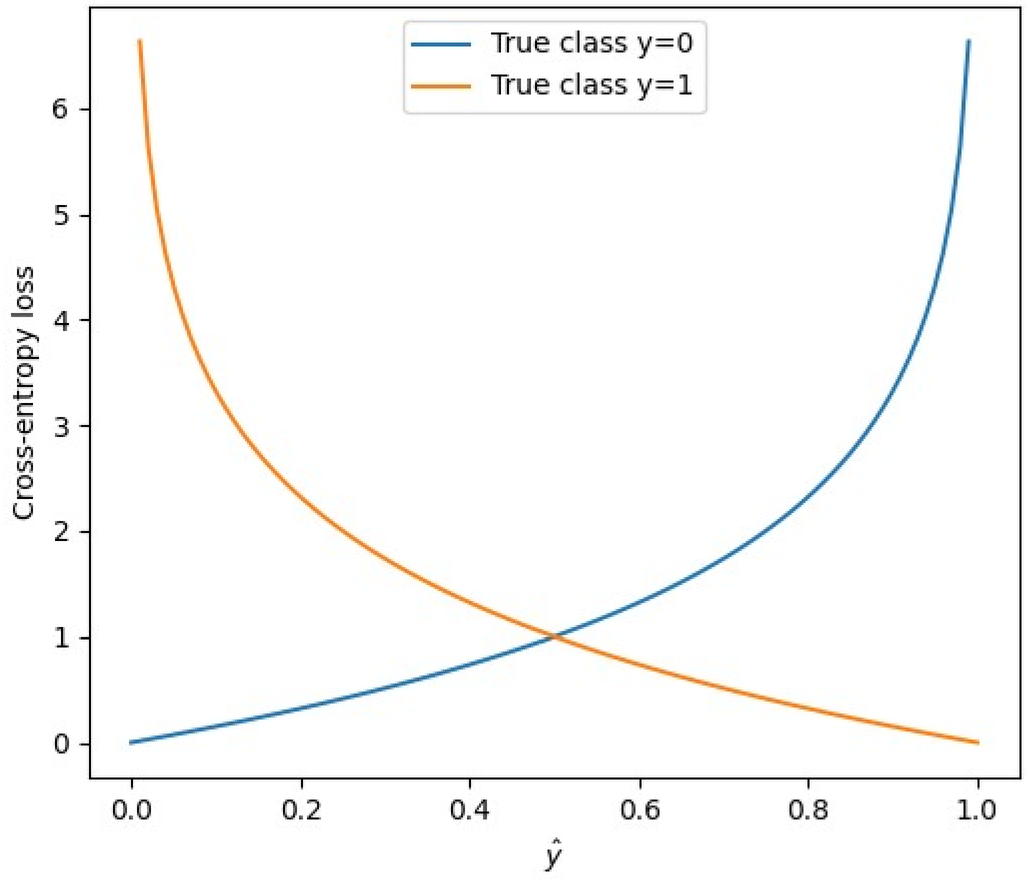
The **cross-entropy** loss function (cost function) for a single value y:

$$L(y, \hat{y}) = -y \log \hat{y} - (1-y) \log (1-\hat{y})$$

Usually the true value y is either 0 or 1 (e.g. in a classification: 0 = cat, 1 = dog). In this case the cross-entropy should be understood as:

$$L(y, \hat{y}) = \begin{cases} -\log(\hat{y}) = \log \frac{1}{\hat{y}}, & \text{when true output is } y = 1; \\ -\log(1-\hat{y}) = \log \frac{1}{1-\hat{y}}, & \text{when true output is } y = 0; \end{cases}$$

In both cases, the cost is 0 if $\hat{y} = y$, and it is ∞ when $\hat{y}$ is exactly the opposite, as depicted below:



When there are N input-output pairs, the overall cost is the average for all:

$$J = \frac{1}{N} \sum_i L(y^i, \hat{y}^i) = \frac{1}{N} \sum_i (-y \log \hat{y} - (1-y) \log 1 - \hat{y})$$

How to train logistic regression?

Training = **learning** = finding good values for the unknown parameters.

There is no closed-form solution. The solution is found using numerical algorithms.

Gradient Descent

We can use the same Gradient Descent approach to train the parameters of the model.

- Initialize parameter vector W with random values
- Repeat:
  - Compute cost J (forward pass)
  - Compute the gradient with vector of J with respect to parameters W, $\frac{dJ}{dW}$
  - Update parameters: $W = W - \mu \frac{dJ}{dW}$

For the logistic regresion with the cross-entropy loss, the gradient is equal to:

$$\frac{dJ}{dW} = X^*(\hat{Y} - Y).$$

(TODO: prove this).

That's right, the gradient of logistic regression with cross-entropy loss is the same as the gradient of linear regression with quadratic loss.

Visualization of logistic regression in 2D

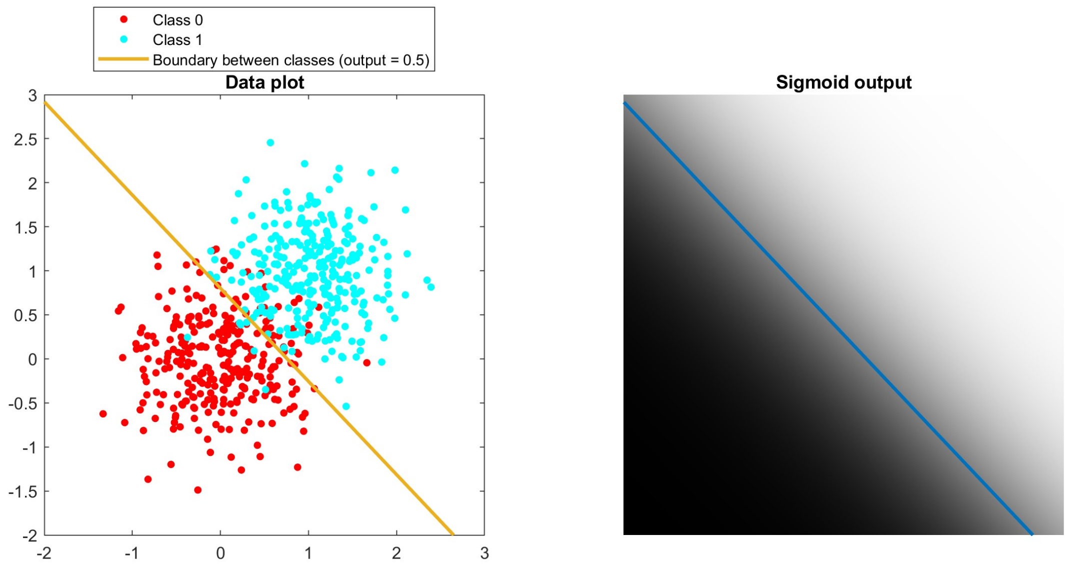
Below is an example of logistic regression in 2D (there are 2 inputs $[x_1, x_2]$), trained with Gradient Descent.

Since we have two inputs, there will be 3 parameters in the W vector (including b).

Note: for a nice graphical animation, please run the code below **directly in Matlab** (file `L2_LogisticRegression_Visualize2D.m`).

In [ ]:
```
clear all
close all

% Load some data
load('LogisticReg.mat');  % the inputs are X, the outputs are Y

% Extend X with a column of 1
N = size(X,1);
X = [X, ones(N,1)];

% Initialize the three parameters in W
W = randn(3,1);

% Repeat Gradient Descent iterations
for iter=1:1000
    % Predict
    z = X * W;
    y_pred = 1./(1 + exp(-z));

    % Cost function
    J(iter) = 1/N * sum(-Y .* log(y_pred) - (1-Y).* log(1-y_pred));

    % Gradient (derivatives)
    dW = X' * (y_pred - Y);

    % Update
    mu = 0.0001;
    W = W - mu*dW;

    %====================================
    % Plotting stuff
    %====================================

    % Plot decision boundary
    subplot(1,2,1)
    gscatter(X(:,1),X(:,2),Y)
    title('Data plot');
    hold on

    % Plot decision line on top of points
    xx = linspace(-2, 3, 1000);
    yy = -W(1)/W(2) * xx - W(3)/W(2);
    hold on
    plot(xx, yy, 'LineWidth',2);
    legend('Class 0', 'Class 1', 'Boundary between classes (output = 
0.5)');
    hold off
    axis([-2, 3, -2, 3])
    axis square

    % On right side, plot grayscale regions
    subplot(1,2,2)
    I = zeros(500,500);
    x_values = linspace(-2, 3, 500);
    y_values = linspace(-2, 3, 500);
    for i = 1:length(x_values)
        x = x_values(i);
        for j = 1:length(y_values)
            y = y_values(j);
            % Compute prediction in point (i,j)
            z = [x, y, 1] * W;
            I(501-i,j) = 1 / (1 + exp(-z));    % 0 = black,  1 = whit
e, in-between = gray
        end
    end
    I = fliplr(flipud(I'));
    imshow(I);
    title('Sigmoid output');
    hold on

    % Plot line on right side as well
    subplot(1,2,2)
    xx_rescaled = (xx + 2)*500/5+1;
    yy_rescaled = 500 - (yy + 2)*500/5+1;
    %plot(xx_rescaled, yy_rescaled, 'LineWidth',2)
    plot(xx_rescaled, yy_rescaled, 'LineWidth',2)
    hold off
    %axis([-2, 3, -2, 3])
    axis square

    drawnow()

    %pause(0.1)
    %===================================
end
```

Expected output



Logistic regression = One separating line

The take-home message from the above example is:

Logistic regression draws one linear frontier (a "hyperplane") in the classification space.

In the following episode we will see how we can combine multiple neurons (multiple hyperplanes) into forming any classification boundary, however complicated.

Matlab function doing the job for us

Linear regression can be fitted in Matlab using the function `fitglm()` :

In [12]:
```
X = X(:,1:2);    % Keep only the original two components, not the one
s we added extra
mdl = fitglm(X, Y, 'Distribution', 'binomial') % X are the inputs, Y
is the target vector, mdl is a model object
```

```
mdl = 


Generalized linear regression model:
    logit(y) ~ 1 + x1 + x2
    Distribution = Binomial

Estimated Coefficients:
                   Estimate      SE        tStat       pValue  
                   ________    _______    _______    __________

    (Intercept)    -4.7378     0.50621    -9.3595     8.015e-21
    x1              4.8554     0.53635     9.0526    1.3957e-19
    x2              4.6079     0.53354     8.6366    5.7933e-18


600 observations, 597 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 637, p-value = 5.55e-139
```

Let's make sime predictions with our model:

In [19]:
```
% Predict
mdl.predict([0,0])
mdl.predict([0,0.5])
mdl.predict([3,-1.5])
```

```
ans =

    0.0087


ans =

    0.0806


ans =

    0.9486
```

To which class do the previous inputs belong?

3. Practical work

We use the same data as in linear regression, but instead we try to predict if one of the two possibilities: the quality score is <=5 (class 0) or the quality score is > 5 (class 1).

As a reminder, the data comes from here: [https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009 (https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009)](https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009), and contains 11 numerical chemical measurements for some different brands of red wines, together with a quality score indicated by buyers (quality goes from 3 to 8).

Inputs:

- 1 - fixed acidity
- 2 - volatile acidity
- 3 - citric acid
- 4 - residual sugar
- 5 - chlorides
- 6 - free sulfur dioxide
- 7 - total sulfur dioxide
- 8 - density
- 9 - pH
- 10 - sulphates
- 11 - alcohol

Outputs:

- 12 - quality

Let's load the data first

```
In [47]: Data = readmatrix('winequality-red.csv');
         X = Data(:,1:11);        % 11 columns for the inputs
         N = size(Data,1);        % The number of wines in the set (1599)

         Y = Data(:,12) > 5;      % make 1 column for the output: 1 if score >
         5, 0 if score <= 5
```

Extend the X matrix so we can treat the bias b as just another weight.

```
In [41]: X = [X ones(N,1)];
```

Let's initialize the weights to some random values

```
In [42]: W = randn(12, 1)    % a column vector
```

```
W =

    2.7694
   -1.3499
    3.0349
    0.7254
   -0.0631
    0.7147
   -0.2050
   -0.1241
    1.4897
    1.4090
    1.4172
    0.6715
```

Task 1: Compute and show the cost function with the above weights

```
In [ ]: %======================
        % Your code here
        %======================
```

Task 2: Implement optimization with Gradient Descent

You can implement a visualizaiton just like in the example provided, by copying and adapting the code. You cannot plot all the 11 dimensions of the input data, so pick only two of them to plot.

In [ ]:
```
%======================
% To fill in
%======================

W = randn(12, 1);           % initialize parameters randomly

number_of_epochs = 1000;    % set number of iterations

for iter = 1:number_of_epochs

    % Compute predictions:
    Y-pred = ...

    % Compute cost:
    J(iter) = 1/N * ...

    % Compute derivatives according to the given formula
    dW = ...

    % Update the weights
    mu = 0.0001;             % try multiple values here
    W = W - mu * dW;

    % Store the weights history
    W_hist(:,i) = W;
end

% Plot the error and the evolution of the weights
plot(J)
figure
plot(W_hist)
```

Task 3: Compute the solution with the Matlab function `fitglm()`

In [43]:
```
%======================
% Your code here
%======================
```

4. Final questions

1. In our example, the parameters W keep updating forever, making the gray transition area smaller and smaller, but the actual frontier does not change much. Why does this happen? How can we prevent it?
2. What happens if the two classes are **unbalanced** (many more inputs in one class compared to the other)?
3. Suggest some good termination conditions for Gradient Descent (i.e. when should we stop the iterations)?