Supervised Learning I
Linear Regression and Logistic Regression

Jeff Tang

Outline

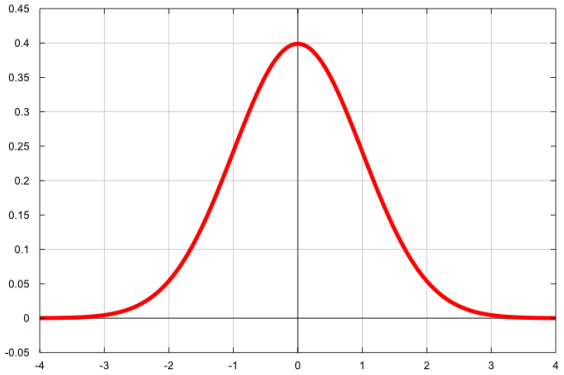
- Introduction
- Prediction of output by linear function mapping: univariate and multivariate inputs
- How Linear Regression works
- MSE, Cost function and Gradient Descent optimization
- Overfitting and Regularization
- Parameters vs. Hyperparameters
- How Logistic Regression works
- Cross Entropy Loss function, Cost function and Gradient Descent optimization
- Practical applications

Introduction

Both *linear regression* and *logistic regression* are parametric, supervised learning algorithms making use of linear models.

Linear Regression

- It is a regression method used to predict an output which is a real number (i.e. continuous), e.g. prediction of housing prices.
- 2 assumptions:
  - The input data (*explanatory variables*) and output data (*response variables*) have a linear relationship.
  - The input and output data have a **normal (Gaussian) distribution**.



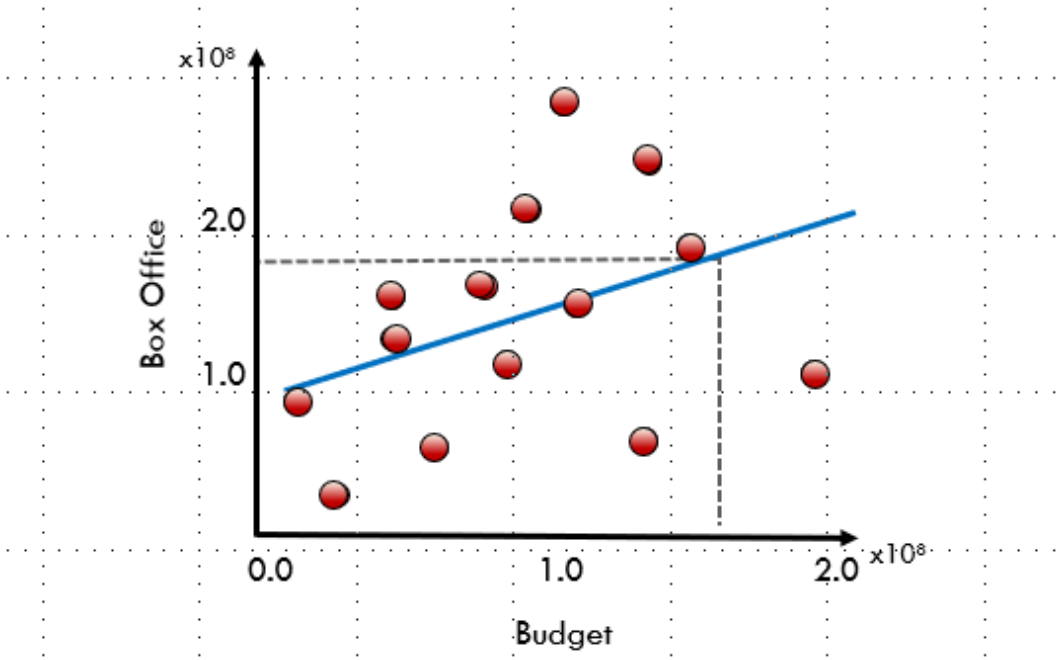
Logistic Regression

- It is a classification method used to predict an output which is a categorical value (i.e. discrete), e.g. prediction of diabetes in patients.
- It does not make the key assumptions of linear regression regarding linearity and normality. However, it assumes a linear relationship between the log odds of the dependent variables and the independent variables.

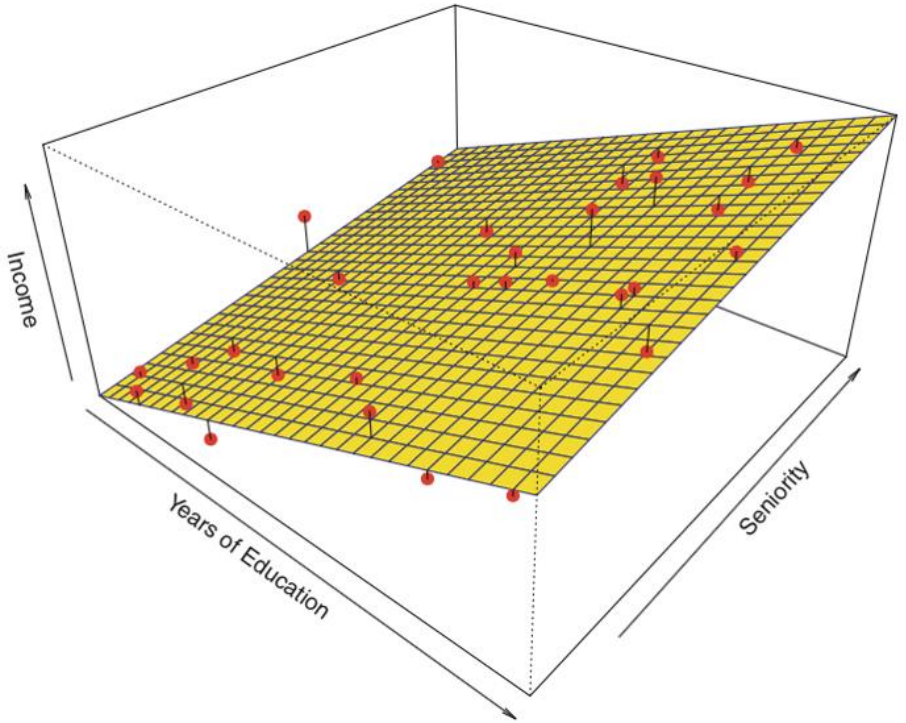
Predicting from linear regression

2 kinds of input variables:

Univariate (1 feature input)



Multivariate (2 or more inputs)



How linear regression works

It works by fitting the data (e.g. multivariate) to a linear model in the following steps:

1. Initialize the model by assigning random coefficients to the function (*hypothesis*):

   $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots\ldots + \theta_n x_n$

   - where x = feature vector and θ = coefficient (weight) vector
   - x_0 is taken as 1 and the constant term, θ_0, is called the *bias*.
   - Graphically, the function can be represented by a *hyperplane*, e.g. it is a line when n = 1 and it is a plane when n = 2.

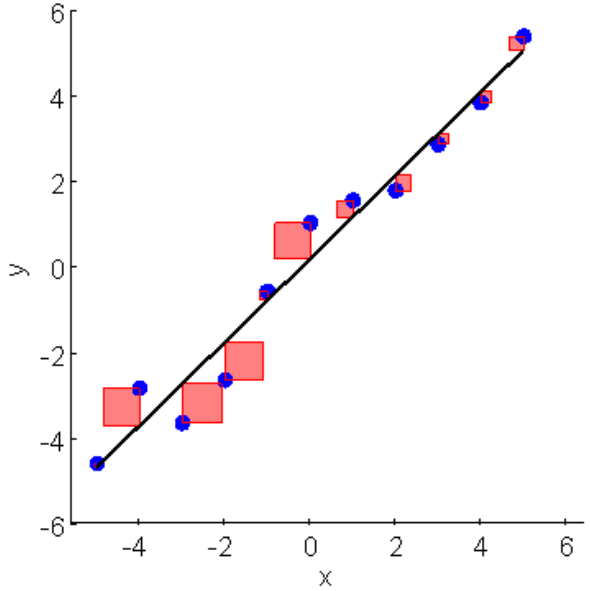
   $x = (x_0, x_1, x_2, \ldots, x_n)$
   $\theta = (\theta_0, \theta_1, \theta_2, \ldots, \theta_n)$

2. Input the sample with labelled data (i.e. *targets*) to the model for processing to give outputs.
3. Calculate the **MSE (Mean Squared Error)** of the predicted outputs from the targets.
4. Update the coefficients iteratively by using **Gradient Descent** as an optimization algorithm to minimize the output errors.
5. Stop until convergence or an acceptable error limit is reached.

Mean Squared Error (MSE) and Cost Function

- **MSE**: the average of the sum of squares of errors of the predicted outputs of a batch of *m* samples in a training set with targets y_i.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$$



- **Cost (Loss) function**: a measure of the MSE

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$$

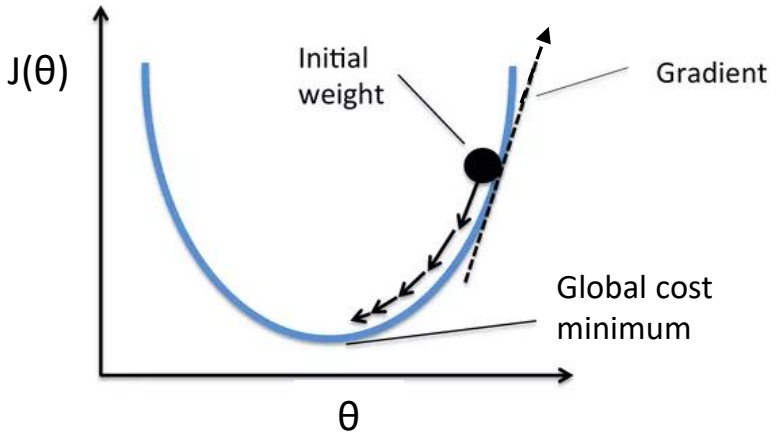
  - ½ is added for convenience to use the **update rule** derived from gradient descent later.

- By minimizing the cost function, the optimal set of weights can be found.

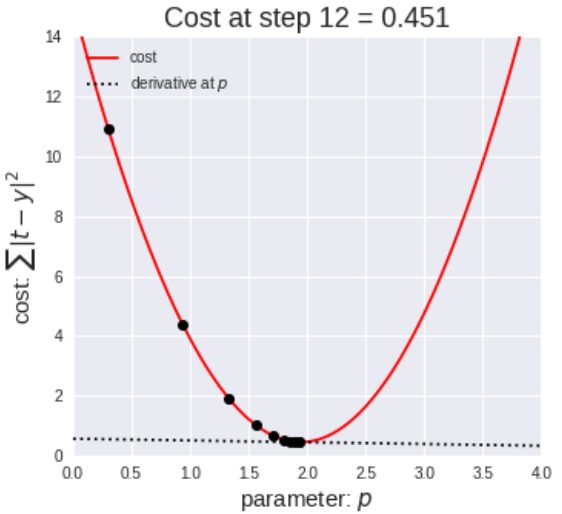
Gradient Descent

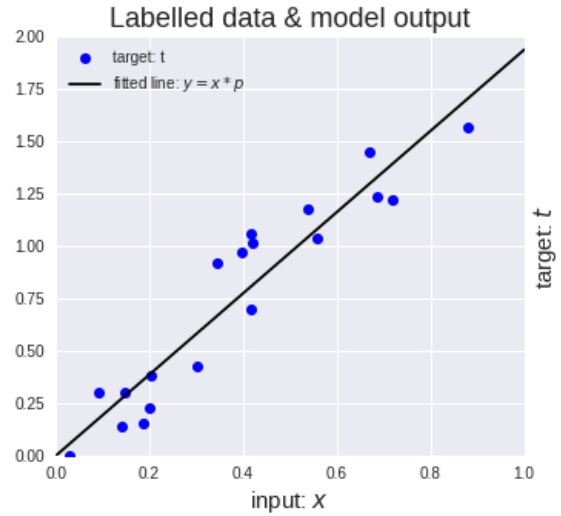
- The *gradient vector* (i.e. the slope) is denoted by the partial derivatives of the cost function with respect to the weights each of which points in the direction of greatest increase of the cost function with the respective weight.

$$\nabla f(x,y) = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix}$$



- Gradient descent iteratively updates the weights by calculating the -ve partial derivatives of the cost function (pointing to the direction moving towards the minimum) and applying the **update rule** for the weights in each iteration.





Gradient Descent and Update rule

To get the update rule:

- Linear function (hypothesis): $h_\theta(x_i) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots \theta_n x_n$
- Cost function: $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$
- Partial derivative of cost function: $\frac{\delta}{\delta\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
- Update rule: $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
  - where α is the *learning rate* which is a scaling factor.
  - θ_j (for j = 0, 1, …., n) are updated to θ_j : simultaneously.
- E.g. for n = 2: $\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \quad (since\ x_0 = 1)$

$$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}$$

Types of gradient descent

- There are 3 types of gradient descent distinguished by the number of training instances that are used for updating the model parameters in each training iteration. One cycle through the entire training dataset is called an *epoch*.
  - **Batch gradient descent (BGD)**: It uses all of the training instances in each iteration (i.e. an epoch).
  - **Stochastic gradient descent (SGD)**: It uses only a single training instance selected randomly in each iteration (i.e. *online training*).
  - **Mini-batch gradient descent (MBGD)**: It uses a small batch of training instances in each iteration.
- Both BGD and SGD can be regarded as special cases of MBGD.
- MBGD is often preferred when the number of training instances is large as it converges more quickly than BGD.
- SGD has a noisy gradient signal (due to frequent updates) but it is faster in learning and is applied in deep learning.

Training example using SGD

Dataset	
x	y
1	1
2	3
4	3
3	2
5	5

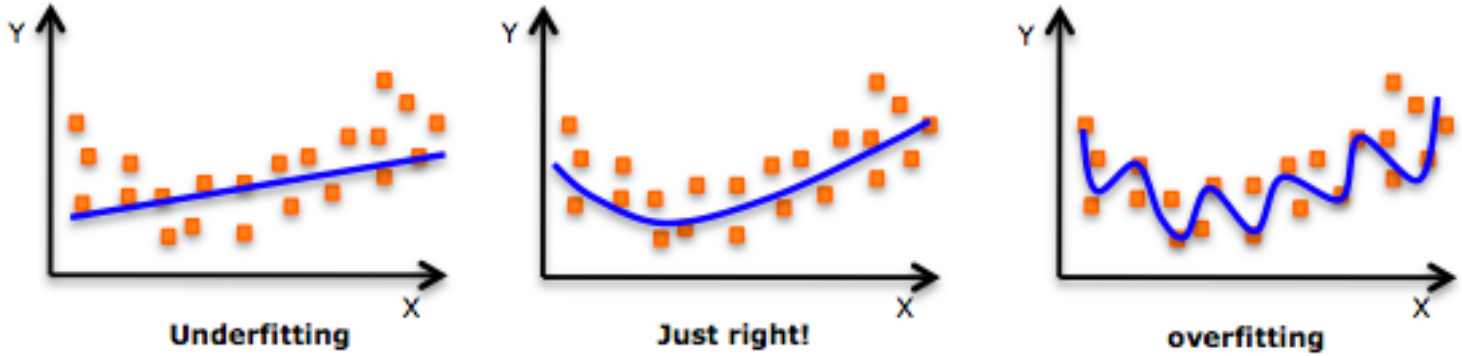
$0.01 - (0.01)(0.03 - 3)$

$0.01 - (0.01)(0.03 - \mathbf{3})(\mathbf{2})$

iteration	x	y	w0	w1	Prediction	learning rate	error	error^2	w0+1	w1+1
1	1	1	0	0	0	0.01	-1	1	0.01	0.01
2	2	3	0.01	0.01	0.03	0.01	-2.97	8.8209	0.0397	0.0694
3	4	3	0.0397	0.0694	0.3173	0.01	-2.6827	7.19687929	0.066527	0.176708
4	3	2	0.066527	0.176708	0.596651	0.01	-1.403349	1.96938842	0.08056049	0.21880847
5	5	5	0.08056049	0.21880847	1.17460284	0.01	-3.8253972	14.6336634	0.11881446	0.41007833
6	1	1	0.11881446	0.41007833	0.52889279	0.01	-0.4711072	0.221942	0.12352553	0.4147894
7	2	3	0.12352553	0.4147894	0.95310433	0.01	-2.0468957	4.18978187	0.14399449	0.45572731
8	4	3	0.14399449	0.45572731	1.96690374	0.01	-1.0330963	1.06728787	0.15432545	0.49705116
9	3	2	0.15432545	0.49705116	1.64547894	0.01	-0.3545211	0.12568518	0.15787066	0.5076868
10	5	5	0.15787066	0.5076868	2.69630464	0.01	-2.3036954	5.30701231	0.18090762	0.62287156
11	1	1	0.18090762	0.62287156	0.80377918	0.01	-0.1962208	0.03850261	0.18286983	0.62483377
12	2	3	0.18286983	0.62483377	1.43253737	0.01	-1.5674626	2.4569391	0.19854445	0.65618302
13	4	3	0.19854445	0.65618302	2.82327655	0.01	-0.1767235	0.03123118	0.20031169	0.66325196
14	3	2	0.20031169	0.66325196	2.19006757	0.01	0.19006757	0.03612568	0.19841101	0.65754994
15	5	5	0.19841101	0.65754994	3.48616069	0.01	-1.5138393	2.29170947	0.2135494	0.7332419
16	1	1	0.2135494	0.7332419	0.9467913	0.01	-0.0532087	0.00283117	0.21408149	0.73377399
17	2	3	0.21408149	0.73377399	1.68162947	0.01	-1.3183705	1.73810087	0.2272652	0.7601414
18	4	3	0.2272652	0.7601414	3.26783079	0.01	0.26783079	0.07173333	0.22458689	0.74942817
19	3	2	0.22458689	0.74942817	2.47287139	0.01	0.47287139	0.22360735	0.21985817	0.73524203
20	5	5	0.21985817	0.73524203	3.8960683	0.01	-1.1039317	1.2186652	0.23089749	0.79043861

Overfitting and Regularization

- **Overfitting** happens when the model learns signal as well as noise in the training data and wouldn't perform well for new data on which the model wasn't trained before, i.e. it can not generalize the data. The model is more complex and is characterized by large parameters.



- **Regularization** basically adds the penalty as model complexity increases. In **Ridge regression (L2 Regularization)**, depending on the *regularization coefficient* (λ), all the weights (except the bias) will be penalized/shrink so that the model generalizes the data and won't overfit.

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

$$\min_\theta J(\theta)$$

(Ridge regression/L2 regularization)

Overfitting and Regularization (cont'd)

- An alternative approach is **LASSO regression (L1 regularization):**

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n}|\theta_j|\right]$$

$$\min_\theta J(\theta)$$

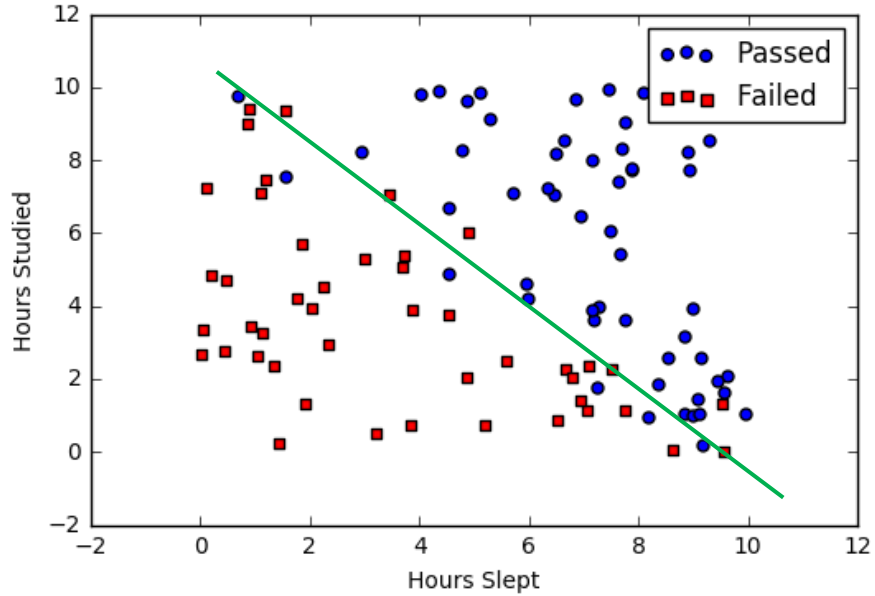
  - **LASSO** stands for **Least Absolute Shrinkage Selector Operator**.
  - The difference of L1 from L2 is that the regularization term is just the sum of weights instead of the sum of square of weights in L2.
  - Those less important weights can become zero (i.e. *sparsity*), which also makes LASSO useful as a *feature selection* technique.

Parameters vs. hyperparameters

- **Model parameters:**
  - They are the properties of the training data that are learnt during training, e.g. weight coefficients.
  - They differ for each experiment and depend on the type of data and task at hand.
- **Model hyperparameters:**
  - They are common for similar models and cannot be learnt during training but are set before training, e.g. learning rate, regulation coefficient, weight initialization scheme, batch size and the number of iterations to be taken.
  - The learning rate, α, must be set to an appropriate value. Its size determines how fast we will move towards the optimal weights. If α is too large, we might skip the optimal solution. If it is too small, too many iterations is needed to reach convergence.
  - In both L1 and L2 regularizations, the model will become overfitting if the regularization coefficient, λ, is too small and underfitting if it is too large.
  - The initial random weights should be small, e.g. all set to be zero.

How Logistic Regression works

- In binary logistic regression, a linear equation with optimized coefficients is to be found to give a *decision boundary* (in green) which can be used to classify the featured data into 2 classes as most accurate as possible, e.g. to predict a student to get a pass or failure with 2 features: hours slept and hours studied.

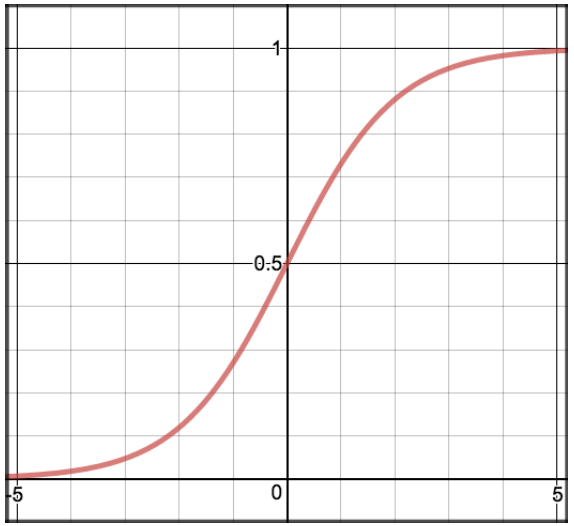


- Steps taken (for binary classification):
  1. Compute weighted outputs using a linear model equation (similar to linear regression).
  2. Transform its output using the **Sigmoid function** to return a probability value which can then be used for mapping to the discrete classes.
  3. Calculate the **Cross Entropy** of the predicted output and use it as the cost function.
  4. Update the coefficients iteratively by using gradient descent as an optimization algorithm to minimize the cost function.
  5. Stop until convergence or an acceptable error limit is reached.

Sigmoid function and Threshold value

- In order to map predicted values to probabilities, **Sigmoid function** (σ) is used to map any real value into another value between 0 and 1.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



  - where z = the weighted output from the linear model

(i.e. $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots\ldots + \theta_n x_n$)

- Threshold value (= 0.5): with probability p as the sigmoid output:

$$p \geq 0.5, class = 1$$
$$p < 0.5, class = 0$$

  - E.g. If the model returns 0.4 (<0.5), it believes there is only a 40% chance of being class 1, we would categorize this observation as class 2.

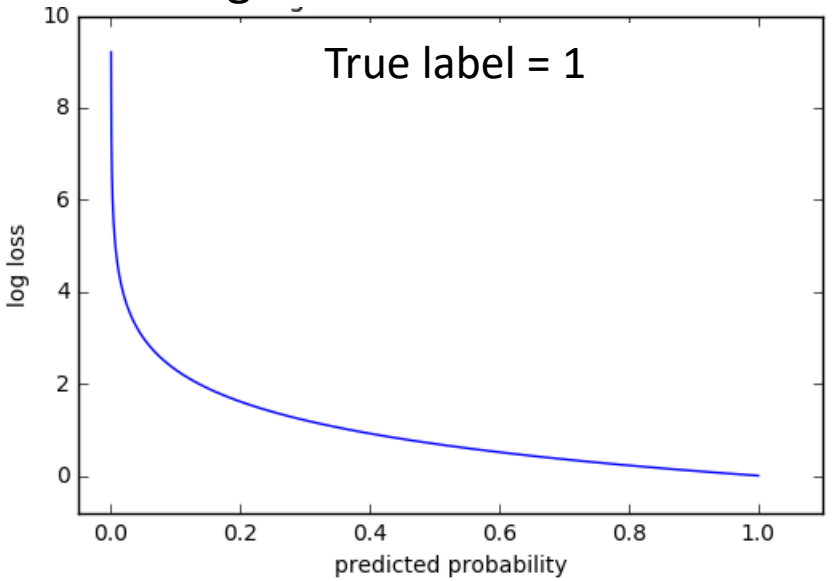
Cross entropy and Cost function

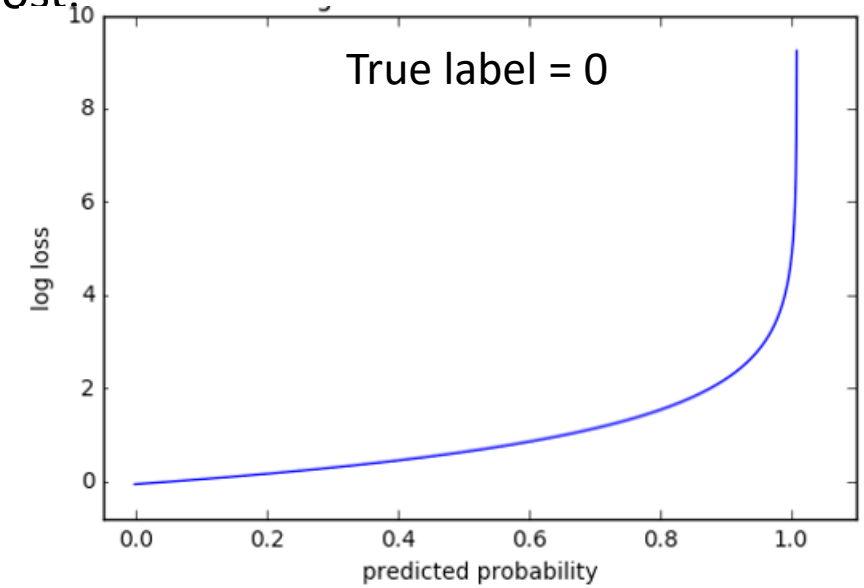
- **Cross entropy** is used instead of MSE as the cost function (J_{CE}) (*log loss*):

$$J_{CE}(y,t) = \begin{cases} -\log y & \text{if } t = 1 \\ -\log 1 - y & \text{if } t = 0 \end{cases}$$

where *y*=predicted probability *t*=target (true label)

- For *t*=1: if *y*=0 then J=+∞ and if *y*=1 then J=0
- For *t*=0: if *y*=0 then J=0 and if *y*=1 then J=+∞
- These smooth *monotonic* functions (always increasing or always decreasing) make it easy to calculate the gradient and hence minimize the cost.

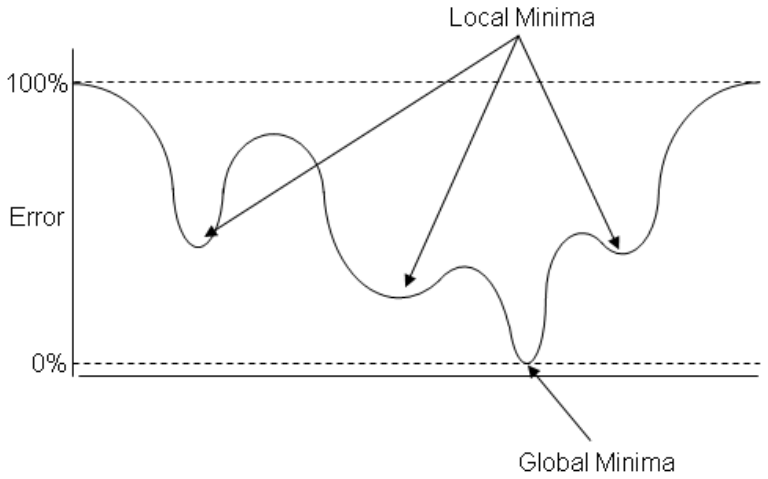




Why not MSE as cost function

- Small difference in distinguishing bad predictions
  - The problem with squared error loss in the classification setting is that it doesn't distinguish bad predictions from extremely bad predictions. If target of t = 1, then a prediction of y = 0.01 has roughly the same squared-error loss as a prediction of y = 0.00001.
  - Using cross entropy, we have $J_{CE}(0.01, 1) = 4.6$ and $J_{CE}(0.00001, 1) = 11.5$. So cross entropy treats the latter as much worse than the former.

- Non-convexity: Generally, there are multiple local minima which make it difficult for the cost function to converge to a global minimum via gradient descent.



Gradient Descent and Update rule

- The partial derivative of the sigmoid function is given by:

$$\frac{\delta}{\delta\theta_j}J_{CE}(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta\left(x^{(i)}\right) - y^{(i)})x_j^{(i)}$$

  - Note that this is the same as the MSE partial derivative, the only difference is the hypothesis function.

- Update rule (same as that for linear regression):

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta\left(x^{(i)}\right) - y^{(i)})x_j^{(i)}$$

Worked example

Learning Rate
0.3

0.7275 – (0.3)(0.99998469 – 0)(8.62)

Iteration	Bias	X1	X2	Y	B0	B1	B2	Prediction	B0(t+1)	B1(t+1)	B2(t+1)
1	1	4.85	9.63	1	0	0	0	0.5	0.15	0.7275	1.4445
1.1	1	8.62	3.23	0	0.15	0.7275	1.4445	0.99998469	-0.1499954	-1.8584604	0.47551484
1.2	1	5.43	8.23	1	-0.1499954	-1.8584604	0.47551484	0.00178251	0.14946984	-0.2323641	2.94011382
1.3	1	9.21	6.34	0	0.14946984	-0.2323641	2.94011382	0.99999994	-0.1505301	-2.9953639	1.03811393
1.4	1	4.85	9.63	1	-0.1505301	-2.9953639	1.03811393	0.00918453	0.1467145	-1.5537274	3.90057983
1.5	1	8.62	3.23	0	0.1467145	-1.5537274	3.90057983	0.3435434	0.04365148	-2.4421307	3.56768628
1.6	1	5.43	8.23	1	0.04365148	-2.4421307	3.56768628	0.9999999	0.04365151	-2.4421305	3.56768652
1.7	1	9.21	6.34	0	0.04365151	-2.4421305	3.56768652	0.54258706	-0.1191246	-3.9412986	2.53568593
1.8	1	4.85	9.63	1	-0.1191246	-3.9412986	2.53568593	0.994427	-0.1174527	-3.9331898	2.55178633
1.9	1	8.62	3.23	0	-0.1174527	-3.9331898	2.55178633	6.3709E-12	-0.1174527	-3.9331898	2.55178633
2	1	5.43	8.23	1	-0.1174527	-3.9331898	2.55178633	0.38379479	0.06740885	-2.9293916	4.07319699
2.1	1	9.21	6.34	0	0.06740885	-2.9293916	4.07319699	0.2519539	-0.0081773	-3.6255402	3.59398068
2.2	1	4.85	9.63	1	-0.0081773	-3.6255402	3.59398068	0.99999996	-0.0081773	-3.6255401	3.59398079
2.3	1	8.62	3.23	0	-0.0081773	-3.6255401	3.59398079	2.9197E-09	-0.0081773	-3.6255401	3.59398079
2.4	1	5.43	8.23	1	-0.0081773	-3.6255401	3.59398079	0.999949	-0.008162	-3.625457	3.59410671
2.5	1	9.21	6.34	0	-0.008162	-3.625457	3.59410671	2.4619E-05	-0.0081694	-3.6255251	3.59405989
2.6	1	4.85	9.63	1	-0.0081694	-3.6255251	3.59405989	0.99999996	-0.0081694	-3.625525	3.59406001
2.7	1	8.62	3.23	0	-0.0081694	-3.625525	3.59406001	2.9209E-09	-0.0081694	-3.625525	3.59406
2.8	1	5.43	8.23	1	-0.0081694	-3.625525	3.59406	0.99994904	-0.0081541	-3.625442	3.59418583
2.9	1	9.21	6.34	0	-0.0081541	-3.625442	3.59418583	2.4635E-05	-0.0081615	-3.6255101	3.59413898
3	1	4.85	9.63	1	-0.0081615	-3.6255101	3.59413898	0.99999996	-0.0081615	-3.62551	3.5941391
3.1	1	8.62	3.23	0	-0.0081615	-3.62551	3.5941391	2.922E-09	-0.0081615	-3.62551	3.59413909
3.2	1	5.43	8.23	1	-0.0081615	-3.62551	3.59413909	0.99994907	-0.0081462	-3.625427	3.59426483
3.3	1	9.21	6.34	0	-0.0081462	-3.625427	3.59426483	2.465E-05	-0.0081536	-3.6254952	3.59421795
3.4	1	4.85	9.63	1	-0.0081536	-3.6254952	3.59421795	0.99999996	-0.0081536	-3.6254951	3.59421806
3.5	1	8.62	3.23	0	-0.0081536	-3.6254951	3.59421806	2.9232E-09	-0.0081536	-3.6254951	3.59421806
3.6	1	5.43	8.23	1	-0.0081536	-3.6254951	3.59421806	0.99994911	-0.0081383	-3.6254122	3.5943437

Regularization in logistic regression

- Similar to linear regression, a function is added to penalize large values of the parameters. Most often the function is $\lambda\sum\theta j^2$, which is some constant λ (the *regularization strength*) times the sum of the squared parameter values θj^2. Rather than specifying λ, we specify $C=1/\lambda$ in scikit-learn implementation (for coding use), i.e. C is the inverse of λ.
- The value of λ controls the trade-off between allowing the model to increase its complexity as much as it wants with trying to keep it simple. For example, if λ is very low or 0, the model will have enough power to increase its complexity (overfitting occurs) by assigning big values to the weights. If, on the other hand, we increase the value of λ, the model will tend to underfit, as the model will become too simple.
- C will work the other way around. For small values of C, we increase the regularization strength which will create simple models that underfit the data. For big values of C, we lower the power of regularization which implies that the model is allowed to increase its complexity, and therefore, overfit the data.

Practical Applications

- **Linear regression:**
  - Financial forecasting
  - Software cost prediction, effort prediction and quality assurance.
  - Restructuring of the budget : organization or country.
  - Crime data mining : predicting the crime rate of a State based on drug usage, number of gangs, human trafficking and killings.
- **Logistic regression:**
  - Image segmentation and categorization.
  - Geographic image processing
  - Handwriting recognition
  - Healthcare : analyzing a group of millions of people for myocardial infarction within a period of 10 years.

URL references

- https://machinelearningmastery.com/linear-regression-for-machine-learning/
- http://ucanalytics.com/blogs/intuitive-machine-learning-gradient-descent-simplified/
- https://www.youtube.com/watch?v=hiOQDsdOZ7I
- https://www.analyticsvidhya.com/blog/2015/02/avoid-over-fitting-regularization/
- https://jamesmccaffrey.wordpress.com/2016/10/28/cross-entropy-error-and-logistic-regression/
- http://www.cs.toronto.edu/~guerzhoy/321/lec/W04/onehot.pdf
- https://www.pyimagesearch.com/2016/09/12/softmax-classifiers-explained/
- https://houxianxu.github.io/2015/04/23/logistic-softmax-regression/