01 - Linear Regression

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$$\nabla_{\mathbf{w}} J = \frac{2}{m}\mathbf{X}^T \cdot (\hat{\mathbf{y}} - \mathbf{y})$$
$$\nabla_{\mathbf{b}} J = \frac{2}{m}(\hat{\mathbf{y}} - \mathbf{y})$$

$$\mathbf{w} = \mathbf{w} - \eta\nabla_w J$$
$$b = b - \eta\nabla_b J$$

02 - Logistic Regression

Cross-entropy can be used to define a loss function in machine learning and optimization. The true probability p_i is the true label, and the given distribution q_i is the predicted value of the current model.

More specifically, consider logistic regression, which (among other things) can be used to classify observations into two possible classes (often simply labelled 0 and 1 . The output of the model for a given observation, given a vector of input features x, can be interpreted as a probability, which serves as the basis for classifying the observation. The probability is modeled using the logistic function $g(z) = 1/(1+e^{-z})$ where z is some function of the input vector$x$, commonly just a linear function. The vector of weights $\mathbf{w}$ is optimized through some appropriate algorithm such as gradient descent.

Having set up our notation, $p \in \{y, 1-y\}$ and $q \in \{\hat{y}, 1-\hat{y}\}$, we can use cross-entropy to get a measure of dissimilarity between p and q:

Logistic regression typically optimizes the log loss for all the observations on which it is trained, which is the same as optimizing the average cross-entropy in the sample. For example, suppose we have N samples with each sample indexed by $n = 1, \ldots, N$. The *average* of the loss function is then given by:

where

$$\hat{y}_n \equiv g(\mathbf{w}\cdot\mathbf{x}_n) = 1/(1+e^{-\mathbf{w}\cdot\mathbf{x}_n})$$

with $g(z)$ the logistic function as before.

The logistic loss is sometimes called cross-entropy loss. It is also known as log loss (In this case, the binary label is often denoted by {-1,+1}).(2)

Remark: The gradient of the cross-entropy loss for logistic regression is the same as the gradient of the squared error loss for Linear regression. That is, define

$$X^T = \begin{pmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ & & \dots & \\ 1 & x_{n1} & \dots & x_{np} \end{pmatrix} \in \mathbb{R}^{n \times (p+1)}$$

$\hat{y}_i = \hat{f}\left(x_{i1}, \dots, x_{ip}\right) = \frac{1}{1+exp(-\beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})}$

$L(\overrightarrow{\beta}) = -\sum_{i=1}^{N} [y^i \log \hat{y}^i + (1-y^i) \log(1-\hat{y}^i)$

Then we have the result

$$\frac{\partial}{\partial \overrightarrow{\beta}} L(\overrightarrow{\beta}) = X(\hat{Y} - Y)$$

The proof is as follows. For any $\hat{y}^i$, we have

$\frac{\partial}{\partial \beta_0} \ln \frac{1}{1+e^{-\beta_0+k_0}} = \frac{e^{-\beta_0+k_0}}{1+e^{-\beta_0+k_0}}$

$\frac{\partial}{\partial \beta_0} \ln\left(1 - \frac{1}{1+e^{-\beta_0+k_0}}\right) = \frac{-1}{1+e^{-\beta_0+k_0}}$

$$\begin{aligned} \frac{\partial}{\partial \beta_0} L(\overrightarrow{\beta}) &= -\sum_{i=1}^{N} \left[\frac{y^i \cdot e^{-\beta_0+k_0}}{1+e^{-\beta_0+k_0}} - (1-y^i) \frac{1}{1+e^{-\beta_0+k_0}} \right] \\ &= -\sum_{i=1}^{N} [y^i - \hat{y}^i] = \sum_{i=1}^{N} (\hat{y}^i - y^i) \end{aligned}$$

$\frac{\partial}{\partial \beta_1} \ln \frac{1}{1+e^{-\beta_1 x_{i1}+k_1}} = \frac{x_{i1} e^{k_1}}{e^{\beta_1 x_{i1}} + e^{k_1}}$

$\frac{\partial}{\partial \beta_1} \ln\left[1 - \frac{1}{1+e^{-\beta_1 x_{i1}+k_1}}\right] = \frac{-x_{i1} e^{\beta_1 x_{i1}}}{e^{\beta_1 x_{i1}} + e^{k_1}}$

$\frac{\partial}{\partial \beta_1} L(\overrightarrow{\beta}) = -\sum_{i=1}^{N} x_{i1}(y^i - \hat{y}^i) = \sum_{i=1}^{N} x_{i1}(\hat{y}^i - y^i)$

In a similar way, we eventually obtain the desired result.

$\hat{\mathbf{y}} = \sigma(\mathbf{a}) = \frac{1}{1+\exp(-\mathbf{a})}$

$J(\mathbf{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(\hat{y}^{(i)}\right) + \left(1-y^{(i)}\right) \left(1 - \log\left(\hat{y}^{(i)}\right)\right) \right]$

Derive the gradients of objective function:

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

$\underset{\text{linearity}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\partial}{\partial \theta_j} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \frac{\partial}{\partial \theta_j} \log\left(1 - h_\theta\left(x^{(i)}\right)\right) \right]$

$\underset{\text{chain rule}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\frac{\partial}{\partial \theta_j} h_\theta\left(x^{(i)}\right)}{h_b\left(x^{(i)}\right)} + \left(1 - y^{(i)}\right) \frac{\frac{\partial}{\partial \theta_j}\left(1 - h_\theta\left(x^{(i)}\right)\right)}{1 - h_\theta\left(x^{(i)}\right)} \right]$

$= \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\frac{\partial}{\partial \theta_j} \sigma\left(\theta^\top x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} + \left(1 - y^{(i)}\right) \frac{\frac{\partial}{\partial o_j}\left(1 - \sigma\left(\theta^\top x^{(i)}\right)\right)}{1 - h_\theta\left(x^{(i)}\right)} \right]$

$= \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{h_\theta\left(x^{(i)}\right)\left(1 - h_\theta\left(x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} - \left(1 - y^{(i)}\right) \frac{h_\theta\left(x^{(i)}\right)\left(1 - h_\theta\left(x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{1 - h_\theta\left(x^{(i)}\right)} \right]$

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

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

Thus, $\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left[\hat{y}^{(i)} - y^{(i)} \right] x_j^{(i)}$

$$\mathbf{w} = \mathbf{w} - \eta \nabla_w J$$
$$b = b - \eta \nabla_b J$$

03 - Perceptron

$$\mathbf{a} = \mathbf{X} \cdot \mathbf{w} + b$$

$$\hat{y}^{(i)} = 1 if a^{(i)} \geq 0, \text{ else } 0$$

$$\Delta \mathbf{w} = \eta \mathbf{X}^T \cdot (\hat{\mathbf{y}} - \mathbf{y})$$
$$\Delta b = \eta (\hat{\mathbf{y}} - \mathbf{y})$$

$$\mathbf{w} = \mathbf{w} + \Delta \mathbf{w}$$
$$b = b + \Delta b$$

From Hands-on DL, Perceptron learning rule (weight update) $w_{i,j}^{(\text{ next step })} = w_{i,j} + \eta \left(y_j - \hat{y}_j\right) x_i$

wi, j is the connection weight between the ith input neuron and the jth output neuron.

xi is the ith input value of the current training instance.

yˆj is the output of the jth output neuron for the current training instance.

yj is the target output of the jth output neuron for the current training instance.

η is the learning rate.

04 - KNN

Classification: a) unweighted: output the most common classification among the k-nearest neighbors b) weighted: sum up the weights of the k-nearest neighbors for each classification value, output classification with highest weight

Regression: a) unweighted: output the average of the values of the k-nearest neighbors b) weighted: for all classification values, sum up classification value $*$ weight and divide the result trough the sum of all weights

05 - K-Means

Given a set of data points x_1, ..., x_n and a positive number k, find the clusters C_1, ... C_k that minimize

$$J = \sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} \|x_i - \mu_j\|_2$$

where:

- $z_{ij} \in \{0, 1\}$ defines whether of not datapoint x_i belongs to cluster C_j
- μ_j denotes the cluster center of cluster C_j
- $\| \|_2$ denotes the Euclidean distance

06 - Simple NN

The training set consists of $m = 750$ examples. Therefore, we will have the following matrix shapes:

- Training set shape: $X = (750, 2)$
- Targets shape: $Y = (750, 1)$
- W_h shape: $(n_{features}, n_{hidden}) = (2, 6)$
- b_h shape (bias vector): $(1, n_{hidden}) = (1, 6)$
- W_o shape: $(n_{hidden}, n_{outputs}) = (6, 1)$
- b_o shape (bias vector): $(1, n_{outputs}) = (1, 1)$

Use the logistic regression lost function

$$J(\mathbf{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \left(1 - \log\left(\hat{y}^{(i)}\right)\right) \right]$$

1. Initialize the parameters (i.e. the weights and biases)
2. Repeat until convergence: 2.1. Propagate the current input batch forward through the network. To do so, compute the activations and outputs of all hidden and output units. 2.2

Compute the partial derivatives of the loss function with respect to each parameter 2.3 Update the parameters

Forward Pass

The hidden neurons will have tanh as their activation function

$\tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{\exp(x) - exp(-x)}{\exp(x) + exp(-x)}$

$\tanh'(x) = 1 - tanh^2(x)$

The output neurons will have th sigmoid activation function

$\sigma(x) = \frac{1}{1+\exp(-x)}$

$\sigma'(x) = 1 - (1 + \sigma(x))$

The activations and outputs can then be computed as follows

$\boldsymbol{A}_h = \boldsymbol{X} \cdot \boldsymbol{W}_h + \boldsymbol{b}_h, \text{shape: } (750, 6)$

$\boldsymbol{O}_h = \sigma(\boldsymbol{A}_h), \text{shape: } (750, 6)$

$\boldsymbol{A}_o = \boldsymbol{O}_h \cdot \boldsymbol{W}_o + b_o, \text{shape: } (750, 1)$

$\boldsymbol{O}_o = \sigma(\boldsymbol{A}_o), \text{shape: } (750, 1)$

Backward Pass

To compute the weight updates we need the partial derivatives of the loss function with respect to each unit.

For the output neurons, the gradients are given by (matrix notation):

$\frac{\partial L}{\partial \boldsymbol{A}_o} = d\boldsymbol{A}_o = (\boldsymbol{O}_o - \boldsymbol{Y})$

$\frac{\partial L}{\partial \boldsymbol{W}_o} = \frac{1}{m}(\boldsymbol{O}_h^T \cdot d\boldsymbol{A}_o)$

$\frac{\partial L}{\partial \boldsymbol{b}_o} = \frac{1}{m} \sum d\boldsymbol{A}_o$

For the weight matrix between input and hidden layer we have:

$\frac{\partial L}{\partial \boldsymbol{A}_h} = d\boldsymbol{A}_h = (\boldsymbol{W}_o^T \cdot d\boldsymbol{A}_o) * (1 - \tanh^2(\boldsymbol{A}_h))$

$\frac{\partial L}{\partial \boldsymbol{W}_h} = \frac{1}{m}(\boldsymbol{X}^T \cdot d\boldsymbol{A}_h)$

$\frac{\partial L}{\partial \boldsymbol{b}_h} = \frac{1}{m} \sum d\boldsymbol{A}_h$

Weight Update

$\boldsymbol{W}_h = \boldsymbol{W}_h - \eta * \frac{\partial L}{\partial \boldsymbol{W}_h}$

$\boldsymbol{b}_h = \boldsymbol{b}_h - \eta * \frac{\partial L}{\partial \boldsymbol{b}_h}$

$\boldsymbol{W}_o = \boldsymbol{W}_o - \eta * \frac{\partial L}{\partial \boldsymbol{W}_o}$

$\boldsymbol{b}_o = \boldsymbol{b}_o - \eta * \frac{\partial L}{\partial \boldsymbol{b}_o}$

07 - Softmax regression

For class $k$and input vector $\mathbf{x}^{(i)}$ we have:

$score_k(\boldsymbol{x}^{(i)}) = \boldsymbol{w}_k^T \cdot \boldsymbol{x}^{(i)} + b_k$

$\hat{p}_k(\boldsymbol{x}^{(i)}) = \frac{\exp(score_k(\boldsymbol{x}^{(i)}))}{\sum_{j=1}^{K} \exp(score_j(\boldsymbol{x}^{(i)}))}$

we can perform this step for all classes and training examples at once using vectorization. The class predicted by the model for $\mathbf{x}^{(i)}$ is then simply the class with the highest probability.

$J(\boldsymbol{W}, b) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \log(\hat{p}_k^{(i)}) \right]$

So $y_k^{(i)}$ is 1 is the target class for $\mathbf{x}^{(i)}$ is k, otherwise $y_k^{(i)}$ is 0.

$\nabla_{\boldsymbol{w}_k} J(\boldsymbol{W}, b) = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}^{(i)} \left[\hat{p}_k^{(i)} - y_k^{(i)} \right]$

$\boldsymbol{w}_k = \boldsymbol{w}_k - \eta \nabla_{\boldsymbol{w}_k}$

$b_k = b_k - \eta \nabla_{b_k} J$

08 - Decision Tree for Classification

Given a set of training examples and their labels, the algorithm repeatedly splits the training examples D into two subsets D_{left}, D_{right} using some feature f and feature threshold $t_f$such that samples with the same label are grouped together. At each node, the algorithm selects the split $\theta = (f, t_f)$ that produces the purest subsets, weighted by their size. Purity/impurity is measured using the *Gini impurity*.

So at each step, the algorithm selects the parameters θ that minimize the following cost function:

$J(D, \theta) = \frac{n_{left}}{n_{total}} G_{left} + \frac{n_{right}}{n_{total}} G_{right}$

- D: remaining training examples
- n_{total} : number of remaining training examples
- $\theta = (f, t_f)$: feature and feature threshold
- n_{left} / n_{right}: number of samples in the left/right subset
- G_{left} / G_{right}: Gini impurity of the left/right subset

This step is repeated recursively until the *maximum allowable depth* is reached or the current number of samples n_{total} drops below some minimum number.

Given K different classification values $k \in \{1, ..., K\}$ the Gini impurity of node m is computed as follows:

$G_m = 1 - \sum_{k=1}^{K} (p_{m,k})^2$

where $p_{m,\ k}$ is the fraction of training examples with class k among all training examples in node m.

The Gini impurity can be used to evaluate how good a potential split is. A split divides a given set of training examples into two groups. Gini measures how "mixed" the resulting groups are. A perfect separation (i.e. each group contains only samples of the same class) corresponds to a Gini impurity of 0. If the resulting groups contain equally many samples of each class, the Gini impurity will reach its highest value of 0.5

Without regularization, decision trees are likely to overfit the training examples. This can be prevented using techniques like *pruning* or by providing a maximum allowed tree depth and/or a minimum number of samples required to split a node further.

09 - Decision Tree for Regression

Given a set of training examples and their labels, the algorithm repeatedly splits the training examples D into two subsets D_{left}, D_{right} using some feature f and feature threshold t_f such that samples with the same label are grouped together. At each node, the algorithm selects the split $\theta = (f, t_f)$ that produces the smallest *mean squared error* (MSE) (alternatively, we could use the mean absolute error).

So at each step, the algorithm selects the parameters θ that minimize the following cost function:

$J(D, \theta) = \frac{n_{left}}{n_{total}} MSE_{left} + \frac{n_{right}}{n_{total}} MSE_{right}$

- D: remaining training examples

- n_{total} : number of remaining training examples
- $\theta = (f, t_f)$: feature and feature threshold
- n_{left} / n_{right}: number of samples in the left/right subset
- MSE_{left} / MSE_{right} : MSE of the left/right subset

This step is repeated recursively until the *maximum allowable depth* is reached or the current number of samples n_{total} drops below some minimum number.

When performing regression (i.e. the target values are continuous) we can evaluate a split using its MSE. The MSE of node m is computed as follows:

$$\hat{y}_m = \frac{1}{n_m} \sum_{i \in D_m} y_i$$

$$MSE_m = \frac{1}{n_m} \sum_{i \in D_m} (y_i - \hat{y}_m)^2$$

- D_m: training examples in node m
- n_m : total number of training examples in node m
- y_i: target value of $i-$ th example

Without regularization, decision trees are likely to overfit the training examples. This can be prevented using techniques like pruning or by providing a maximum allowed tree depth and/or a minimum number of samples required to split a node further.