ML/DL fundamentals

Prerequisites for ML and DL: part 2

0.1 The learning method

Here we deal with the central question of - how are the model parameters (w_i and b) actually learned ? First of all note that logistic regression is a kind of supervised classification method in which we have a correct label y for each observation x. The system ultimately produces $\hat{y}$ which is the estimate for y and we want to choose parameters w and b such that this $\hat{y}$ is as close as possible to y for every observation. Note that this measure of 'closeness' requires a measure of similarity, however we often deal with a measure of dissimilarity, embodied by the **loss function** or **cost function**. The next important thing we need is an optimization algorithm that iteratively updates our weights so that the loss function is minimized.

0.2 Cross entropy loss

We require a loss function that expresses for each observation x, how close the classifier output $\hat{y} = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ is to the correct label y (0 or 1). This could be denoted as:

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from } y \tag{1}$$

We need to design a loss function that prefers the correct class labels of the training examples to be more likely. A process called **conditional maximum likelihood estimation** is used. Basically we select parameters w and b that **maximize the log probability of the occurrence of true y labels in the training data** given an observation x. So the resulting loss function would be the **negative log likelihood loss** which is known as the **cross entropy loss**.

0.2.1 Derivation

Let us say that we are given an observation x and we want to learn weights that maximize the probability of the correct label $p(y|x)$. Note that since there are only two discrete outcomes (1 or 0) this is essentially a **Bernoulli distribution**. The probability that our classifier produces $p(y|x)$ could be written as:

$$p(y|x) = \hat{y}^y(1-\hat{y})^{1-y} \tag{2}$$

Where we note that if $y = 1$ then the probability would simply be $\hat{y}$ and if $y = 0$ then probability would simply be $(1-\hat{y})$. Now take $\log$ on both sides since we note that whatever maximizes probability will also maximize the log of probability.

$$\log[p(y|x)] = \log[\hat{y}^y(1-\hat{y}^{1-y})] = y\log\hat{y} + (1-y)\log(1-\hat{y}) \tag{3}$$

Note that this is essentially an objective function that should ideally be maximized, but in loss functions we are usually dealing with the problem of **minimizing the error** so if we put a minus sign we will get our loss function or the **cross entropy loss**:

$$L_{CE}(\hat{y}, y) = -\log[p(y|x)] = -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{4}$$

Now rewrite the above equation by plugging in the definition of $\hat{y} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x} + b)$. We get the following:

$$L_{CE}(w, b) = -y\log[\sigma(\boldsymbol{w}\cdot\boldsymbol{x} + b) + (1-y)\log(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x} + b))] \tag{5}$$

Note that we want the loss function to assume a small value if the model estimate is very close to the correct value and big if the model estimate if far from the true value.

0.3 Gradient descent

Now that we have laid down our objective function in terms of the cross entropy loss, we now want to find the optimal weights that minimize that function. The notations change here a bit - in the sense that we essentially parameterize the loss function L by its weights, which are in general referred to by θ where $\theta = (w, b)$.

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{m}\sum_{i=1}^{m} L_{CE}(y^{(i)}, x^{(i)}; \boldsymbol{\theta}) \tag{6}$$

Note that the $1/m$ and summation only suggest that we are wanting to minimize the average loss for m training observations. Now gradient descent is a method that finds the minimum of a function by figuring out in which direction (in the space of parameters θ) does the function's slope rises most steeply, and then moves in the opposite direction towards the minima. Now actually the gradient is defined for a **vector of observations** but we first look at the case of a single scalar weight w that we want to find from the minimum.
We first give a random initialization of w at w_1. Then the algorithm should tell us whether to move to the left or right. We note that since the slope at w_1 is negative, we move in the opposite direction (to the right hand side of w_1). After this we indeed find ourselves at a lower position, wherein the loss function is smaller. To summarize, the gradient descent algorithm finds the **gradient** of the loss function at the current point and then moves in the opposite direction. Now if we scale this concept, the gradient of a **function of many variables** is a **vector pointing in the direction of greatest increase in the function**. The gradient is essentially the multivariate generalization of the slope.

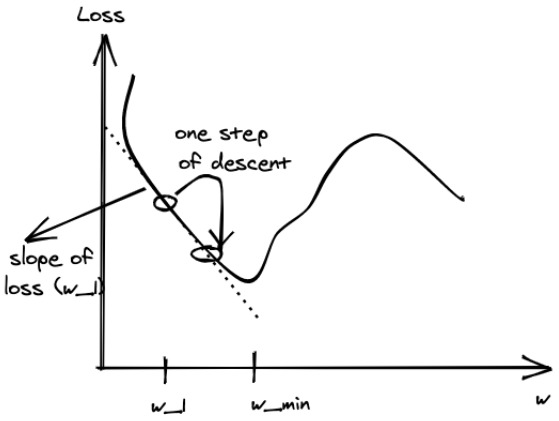

Figure 1: Sigmoid function

0.3.1 Learning rates

So we have now decided which direction to move in, but how much to move ? Basically, the magnitude to move in a certain direction in gradient descent is the value of the slope $\frac{d}{dw}(f(x;w))$ weighted by the **learning rate** η. A higher or a faster learning rate would imply a larger movement at each step. So for a single variable case here is how the values of weights are iteratively updated:

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x;w) \tag{7}$$

Now extending this concept to the N dimensional space, we ideally won't just move left or right, but in the N dimensional space. We do this with the help of the **gradient**, which expresses the directional components of the sharpest slope along each of the N dimensions. For example, in case of 2 weight dimensions, the gradient would be a vector with two **orthogonal components**, with each component laying down the direction in which the multidimensional function moves with respect to partial derivative of each weight taken individually. In practical scenarios, we are dealing with many weights, since we have many features x_i in our dataset and there would naturally be a weight w_i associated with each such feature. We must then ask the critical question - *how much change would result in the loss function if we make a small change in* w_i *?* Note that in each dimension w_i, the slope (along that dimension) is the **partial derivative** of the loss function with resepct to that dimension. **The gradient is a vector of these partials**. This could be represented as:

$$\nabla_\theta L(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \end{bmatrix} \tag{8}$$

Finally we would update the weight vector as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(f(x;\boldsymbol{\theta}),\boldsymbol{y}) \tag{9}$$

0.3.2 Gradient for Logistic regression

Note that in order to update the weight parameters, we need the gradient for the loss function. Recall that in Logistic regression, the loss function is given by:

$$L_{CE}(\boldsymbol{w}, b) = -[y \log(\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)) + (1 - y) \log(1 - \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b))] \tag{10}$$

Note that the partial derivative along on weight dimension would be given by:

$$\frac{\partial L_{CE}(\boldsymbol{w}, b)}{\partial w_j} = [\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)]x_j \tag{11}$$

0.4 Mini-batch learning

Note that **stochastic gradient descent** has the word stochastic in it since it randomly chooses a single observation at a time, moving the weights with each such observation toward the desired minima. This is generally considered a bit inefficient since it is often better to compute the gradient over **batches of training instances** rather than a single instance each time. Now in **batch training** we just compute the gradient over the entire dataset at once, however this method involves significant computing power.

Instead, we utilize the method of **mini-batch training** wherein with each gradient computation, we train on a subset of m training samples. Since we are now dealing with mini batches instead of single observations, it would be nice to defined the loss function in this mini batch version. We still refer to $x^{(i)}$ and $y^{(i)}$ as the i^{th} observation training features and label respectively. Assuming that the training examples are independent, we have:

$$\log p(training\ labels) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \tag{12}$$

$$= \sum_{i=1}^{m} \log p(y^{i}|x^{(i)}) \tag{13}$$

$$= -\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)}) \tag{14}$$

Further note that the **cost function** is traditionally said to be the average loss. So the cost function for the mini batch of m samples would be the average loss of these m samples:

$$Cost(\boldsymbol{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)}) \tag{15}$$

$$= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(\sigma(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b)) + (1 - y^{(i)} \log(1 - \sigma(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b))) \tag{16}$$

Consequently, the mini batch gradient would be the average of the individual gradients:

$$\frac{\partial Cost(\boldsymbol{w}, b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}[\sigma(\boldsymbol{w}\cdot\boldsymbol{x}^{(i)} + b) - y^{(i)}]x_j^{(i)} \tag{17}$$

0.5 Regularization

There is an inherent problem when it comes to learning weights that makes model predictions perfectly in sync with the actual training values. For example there might be a feature (variable) that perfectly predicts the outcome since it might occur in only one class, so in such a situation it would have a very large weight value. So when the weights try to perfectly fit the details of the training set, they kind of model the noisy accidental correlation as well - this is precisely what is known as **overfitting**. Basically a good model must be able to generalize well from the training data to previously unseen data. Recall that we earlier looked at the **mini batch cost function** (that maximizes log probability) - this was essentially our objective function. Now we will add another **regularization term** $R(\theta)$ to this objective function for a batch of m samples.

$$\hat{\boldsymbol{\theta}} = \arg\max_{\theta}\sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)}) - \alpha R(\boldsymbol{\theta}) \tag{18}$$

This regularizatio term imposed a penalty on large weights. There are traditionally two ways to compute this $R(\theta)$ term - **L2 and L1 regularization**. The L2 method basically computes the **Euclidean distance** of $\boldsymbol{\theta}$ from the origin. If this vector consists of n weights then we would have:

$$R(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2 = \sum_{i=1}^{n}\theta_i^2 \tag{19}$$

This would result in the following new objective function:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\theta}\sum_{i=1}^{m}P(y^{(i)}|x^{(i)}) - \alpha\sum_{j=1}^{n}\theta_j^2 \tag{20}$$

On the other hand, L1 regularization is often known as **Manhattan distance** and is given by:

$$R(\boldsymbol{\theta}) = |\boldsymbol{\theta}|_1 = \sum_{i=1}^{n}|\theta_i| \tag{21}$$

The resulting L1 regularized objective function would thus be:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\theta}P(y^{(i)}|x^{(i)}) - \alpha\sum_{j=1}^{n}|\theta_j| \tag{22}$$

We further note that L1 is popularly known as **Lasso regression** and the L2 method is popularly known as the **Ridge regression**.

0.6 Multinomial Logistic Regression

When we have to classify observations in more than 2 classes then we use the **multinomial logit** approach. In this, we want to know the probability of label y being in each potential class $c \in C$, that is $p(y = c|x)$. Here, a generalization of the sigmoid, known as the **softmax** function is used to estimate the probability $p(y = c|x)$. Note that the softmax function takes a vector $z = [z_1, \cdots, z_k]$ as its input and maps them to a **probability distribution** sucht that each value is in the range $(0, 1)$ and all values sum to 1. Now for a vector z of k dimensions, we write the softmax as:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{23}$$

The above denotes the probability of a single z_i. So, the distribution would come from a vector of such softmax computed values for all z values.

$$softmax(\boldsymbol{z}) = \left[\frac{e^{z_1}}{\sum_{j=1}^{k} e^{z_j}}, \cdots, \frac{e^{z_k}}{\sum_{j=1}^{k} e^{z_j}} \right] \tag{24}$$

The denominator term is common for all the values and is essentially used to normalize the values into probabilities. Now just as in the previous case of the sigmoid, the input to our softmax function would also be the dot product of weighted features plus the bias term. **NOTE** - there would now be a different set of weights and bias terms for each class.

$$p(y = c|x) = \frac{e^{\boldsymbol{w}_c \cdot \boldsymbol{x} + b_c}}{\sum_{j=1}^{k} e^{\boldsymbol{w}_j \cdot \boldsymbol{x} + b_j}} \tag{25}$$

Again, the values given out by the softmax would be strictly between 0 and 1 and would be higher for a certain class if a feature shows a high occurrence in that class. For the rest of the classes the associated probability would be estimated to be low.

References

[1] Daniel Jurafsky, James H Martin - Speech and Language Processing