# TP4 Data Mining: Softmax Classifier - Gradient Descent - L2 Regularizer

Monday 29th April, 2019

Kieran Schubert

# Introduction

Sotfmax classification is a generalization of binary logistic regression for multiple classes K (K > 2). Our training data is such that $(x^1, y^1), ..., (x^m, y^m)$ where the labels y can take K different values (corresponding to the K different classes). The Softmax classifier estimates the probability that a test value $x_0$ belongs to class j $p(y = j|x_0)$ for $j = 1, ...K$ by taking a function of a linear combination of the data x and weights w and outputing k class probabilities $\in$ [0,1], which sum to 1. As this is a linear method, we have that the predicted class label probabilities are a linear combination of the data x and weights w, which are then normalized to be in the range [0,1]. The softmax function is used for this purpose: $p(C_k|x) = \frac{exp(a_k)}{\sum_k exp(a_k)}$, where $a_k(x) = w_k^T x$ is a linear function of the data x and weights w. We see that a set of weights has to be learned for each class k. We will learn these weights through a constrained optimization problem named $L_2$ regularization using three variations of Gradient Descent (online, full batch and mini-batch), and the theoretical and practical results will be presented in the following section using the Iris dataset.

# Theory

**(1. a)** As in binary logistic regression, we can use the negative log-likelihood function as the cost function. In the case of multinomial logistic regression, we obtain a cost function:

$$J(w) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y_i = j\} log \frac{exp(w_j^T x_i)}{\sum_{l=1}^{K} exp(w_l^T x_i)}\right] = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y_i = j\}\left[w_j^T x_i - log(\sum_{l=1}^{K} exp(w_l^T x_i))\right]$$

This cost function sums over all m training examples and for all K classes. We also see that for each class j, we have a vector of weights $w$ where $p$ is the number of features of the dataset, and $k$ is the number of classes:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1k} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2k} \\ \dots & \dots & \dots & \dots & \dots \\ w_{p1} & w_{p2} & w_{p3} & \dots & w_{pk} \end{bmatrix}$$

**(1. b)** A classical way to minimize this cost function is to compute the gradient with respect to the weights, that is we find the weights which minimize the gradient of the cost function. The gradient will be a (p x k) matrix, since we must differentiate with respect to each feature $x_p$. We first start by rewriting the cost function by applying the quotient logarithm rule:

$$\frac{\delta J(w)}{\delta w} = \nabla J(w) = -\frac{\delta J(w)}{\delta w} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y_i = j\}\left[w_j^T x_i - log(\sum_{l=1}^{K} exp(w_l^T) x_i)\right]$$

The first part of the cost function is constant with respect to w, so the differentiation occurs in the square parenthesis. For the first term:

$$\frac{\delta J(w)}{\delta w} w_j^T x_i = 1\{y_i = j\} x_i$$

since this is a linear combination of w and x.

We are dealing with vector differentiation, so the derivative is a vector with each term being the derivative with respect to a specific $w_{kp}$ (one element of each row of the w matrix, see above). The derivative is thus 0 when the class doesn't correspond, and this is why the indicator variable appears. We use the chain rule for the second term:

$$\frac{\delta J(w)}{\delta w} log \sum_{l=1}^{K} exp(w_l^T x_i) = \frac{\sum_{l=1}^{K} exp(w_l^T x_i)\{y_i=j\}x_i}{\sum_{l=1}^{K} exp(w_l^T x_i)} = \sum_{l=1}^{K} \left[ \frac{exp(w_l^T x_i)}{\sum_{l=1}^{K} exp(w_l^T x_i)} \{y_i = j\} x_i \right] = \sum_{l=1}^{K} [p(y = l|x_i)\{y_i = j\}x_i = p(y = j|x_i)x_i]$$

using the fact that $p(C_k|x) = \frac{exp(w_l^T x_i)}{\sum_{l=1}^{K} exp(w_l^T x_i)}$ as defined in the introduction.

Putting it all together we obtain:

$$\frac{\delta J(w)}{\delta w} = \nabla J(w) = -\frac{1}{m} \sum_{i=1}^{m} [x_i(1(y_i = j) - p(y_i = j|x_i))]$$

which is a (p x k) matrix. For the implementation, it is useful to visualize it in the following way:

$$\begin{bmatrix} \frac{\delta J(w)}{\delta w_{11}} & \frac{\delta J(w)}{\delta w_{12}} & \frac{\delta J(w)}{\delta w_{13}} & \cdots & \frac{\delta J(w)}{\delta w_{1k}} \\ \frac{\delta J(w)}{\delta w_{21}} & \frac{\delta J(w)}{\delta w_{22}} & \frac{\delta J(w)}{\delta w_{23}} & \cdots & \frac{\delta J(w)}{\delta w_{2k}} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\delta J(w)}{\delta w_{p1}} & \frac{\delta J(w)}{\delta w_{p2}} & \frac{\delta J(w)}{\delta w_{p3}} & \cdots & \frac{\delta J(w)}{\delta w_{pk}} \end{bmatrix}$$

**(1. c)** As mentionned previously, an arbitrary number of weights has to be estimated for each class. This can cause two issues when this number is large. Firstly, the exponentiation of $w_j^T x_i$ can lead to numerical instability if these values are too large. Secondly, overfitting becomes an issue with too many parameters. This is why an $L_2$ regularizer is added, which penalizes for large weight values, pushing the weights to 0 (but not exactly 0). The regularized cost function is:

$$J(w) = -\frac{1}{m} \left[ \sum_{i=1}^{m} \sum_{j=1}^{K} 1\{y_i = j\} log \frac{exp(w_j^T x_i)}{\sum_{l=1}^{K} exp(w_l^T x_i)} \right] + \frac{\lambda}{2} \sum_{i=1}^{K} \sum_{j=0}^{p} w_{ij}^2$$

which is the same as before, with the added regularization term.

Lambda is usually chosen in $R^+$ with large values of $\lambda$ increasing the penalization. When $\lambda = 0$ we obtain the previous softmax cost function. Here we penalize the weights of each feature and each class (sum over the K classes as the weights are different for each class, and we regularize each weight including the bias up to the $p^{th}$ weight).

**(1. d)** The gradient of the cost function with a $L_2$ regularizer is:

$$\frac{\delta J(w)}{\delta w} = \nabla J(w) = -\frac{1}{m} \sum_{i=1}^{m} [x_i(1(y_i = j) - p(y_i = j|x_i))] + \lambda \sum_{i=1}^{K} \sum_{j=0}^{p} w_{ij}$$

The first part stays the same, and the differentiation of the second term cancels the denominator under $\lambda$.

# Implementation

**(2. a)** See softmax() function in script.

**(2. b)** See softmax_loss() function in script.

**(2. c)** See softmax_loss_gradient() function in script.

**(3. a)** See train() function in script.

**(3. b)** See train() function in script.

# Results

**(4. a)** As outputed in the jupyter notebook, we see that the learned parameters influence the accuracy of the prediction. Results show that the best **training accuracies** were obtained as [0.96 0.98 0.97] for online SGD with LR = 0.1, $\lambda = 0.1$, for mini-batch (n = 20) with LR = 0.1, $\lambda = 0$ and for full batch with LR = 0.0001, $\lambda = 0.1$. Although accuracies are very close, online SGD has the lowest accuracy of the three methods. This makes sense, as online SGD only uses one sample of data at a time to compute the gradient. Less information is used and the prediction is less accurate. However, this method is useful when the dataset is updated one sample at a time (live dataset updates). On the other hand, full batch SGD performs the best as it uses a sample of the size of the training dataset. However, this method may be slow when working with very large datasets. The best compromise if found with mini-batch SGD (n=20) with results on the training set close to the full batch method and a computation time which is lower.

Going to the **test accuracies**, we see that they are lower than on the training set at [0.82 0.96 0.18] respectively. This is to be expected, as we trained our model on the training data to fit our parameters, and then made predictions on the test set. Online SGD performs surprisingly well, and full batch SGD performs the best (virtually no loss in accuracy from the training to the test set). Surprisingly, mini-batch SGD performs very poorly on the test set. This result should be taken with care, as the loss history sometimes computed "NaNs" which then cause the optimization procedure to fail (as a result, the stopping criterion was implemented but commented out as the NaN values caused Gradient Descent to stop even though the optimal weights were not yet attained).

Based on these results, we see that the optimal learning rate (LR) is close to 0 as well as the $\lambda$ regularizer. A large LR may result in "overshooting" the optimum when applying gradient descent. Indeed, the LR is the "step size" the optimization algorithm takes in the opposite direction of the gradient when in a minimization setup. A large LR can thus result in too large steps, and a small LR in slow convergence. The $\lambda$ parameter is useful in high dimensional setups. Since the softmax classifier is a parametric model with linear predictor functions, a dataset with many features will result in many weight parameters and can cause overfitting. $L_2$ regularization penalizes for large values of weights (see equation in section 1. c). The larger the lambda, the lower the values of the weights and the less "complex" the model is. On the other hand, a low value of $\lambda$ penalizes less the weights and a $\lambda = 0$ results in basic softmax regression (one can note that $L_2$ regularization cannot force the weights to 0, but they can be close to 0). We do not have many features in the Iris setting (p = 4), so there is no need to penalize much the weights and the optimal lambdas make sense.

Finally, we see that the best classification relies on a small LR, a $\lambda$ close to 0 and if possible with full batch SDG updates. However, in most cases with large datasets one could rely on mini-batch updates to speed the process up, even if the results shown here do not support this claim.