Softmax Regression Explained

Lerner Zhang mail: lerner.adams@hotmail.com

February 26, 2017

Since logistic regression(a.k.a standard or binomial logistic regression) is a special case of the softmax regresson(also termed multilnomial logistic regression) as reduced in this article, you kind reader can interpret it very well without additional explanation. For convenience, the standard binomial logistic regression(and sometimes the simpler standard linear regression) will dominate the basic concepts in the first part, while the later half, from objective function on, stars the softmax regression. When we say logistic regression we mean either the standard one or the multi-class one.

Softmax, expained in a very straightforward way, is one basic approach to categorical classfication problems, which differs from the non-linear SVM classification in a substantial way. It features its maximum likelihood estimation method. Softmax can be seen in many scenarios such as the well known MNIST problem as an independent algorithm, and more complicated neaural networks, for instance RNN as a "normalization function"(acturally it introduces cross entropy to the loss function). In the attention mechanism it also plays the alike role. If we delve into it a little bit we can comprehend why it is also called the maximum entropy classifier.

In order to measure the distance between two distributions the relative entropy is created. And to make the distance as close as possible between the uniform distribution(Laplace's principle of insufficient reasoning) and the distribution we are modeling, given some constrains, we should maixmize the distribution's entropy. Following some constraints(equations or inequalities like the sum should be 1 and probabilites should all be nonnegative and etc) we obtain various of functions to map the natural(canonical) parameter and the mean parameter using the Lagrange multipliers and maximum entropy principle. If the mean of the distribution is given as the constraint with all others the link function is logit function; however if an variance is given the link funciton becomes probit. If you feel confused by these concepts please don't worry for we'll detail it next time. It is these concepts that help me bridge the gap between pracitce(machine learning) and the theory stuff like probability and informtion theory and etc.

Here we go starting on our adventure into the rabbit hole. The remaining of this post is structured as follows: the first three sections elaborate three parts of the generalized linear model(a umbrella term which covers softmax model).

$$g(\mu) = \beta_0 + \beta_1 X$$

β is the coefficient; u is the mean of a conditional response distribution; $g()$ is the link function. And

in the second half, section 4 and 5, we zoom in on the forward process(calculating the lost using the objective function) and the backword process(backpropagation) of the neural network. In section 4 we approach the softmax by going through the maximum likelihood and conditional likelihood. In the last section we show the updating(update the coefficients) process by backpropagation. At the end in the appendix is the implementation of the softmax from scratch.

1 Coefficients

Coefficients(θ below) are average hidden parameters(weights, or constraints) of the dependent variables(or explanatory variables; training data, observations or observed data are the instances). For simplicity,

$$\underbrace{f(x_1, ..., x_n; \theta)}_{link\ function} \approx \underbrace{y_i}_{follows\ the\ response\ distributio}$$

Take for example the standard two class logistic regression(binary) module mentioned later,

$$\underbrace{\log \frac{p}{1-p}}_{logit} = \underbrace{x_1\beta_1 + x_2\beta_2 + ... + x_n\beta_n}_{linear\ predictor}$$

For each one-unit difference in x_1, holding all other variables constant, the logit(the left hand side of the function above) will diff by β_1 on average, indicating the odds, ratio of success to failure, increases or decreases by log transformed β_1.

In generalized linear models here, the purpose of training, or learning(as in machine learing), is to tune the coefficients(and bias or offset) by minimizing the error, just as we need instant feedback(response or instructions) while learning; through the lens of constructivism learning theory, I'd like to intuit the process by associating the training process with Piaget's accommondation and the predicting as assimilation, and the convergence as aquilibration. As new information(here the coming samples or observations) is processed(forward and most importantly backward) the biological changes take place in our brain, as stated in neuralscience.

2 Response Distribution

The response distribution is the distribution of the response variable In gereralized linear model, the response distribution is always a probability distribution from the exponential family like Gaussian or Possion distribution.

$$P(x\eta) = h(x)exp\{\eta(\theta)^T T(x) - A(\eta)\}$$

For our logistic regression the response distribution is the Bernoulli distribution.

$$P(x|u) = u^x(1-u)^{(1-x)}$$

Which can be transformed to the standard exponential distribution form:

$$P(x|u) = exp\{log(u^x(1-u)^{(1-x)})\} = exp\{(log\frac{u}{1-u})x + log(1-u)\}$$

The natural parameter(η) here is the logit function.

How can we map the linear function contained coefficients to the distribution?

3 Link Function

Link functions find their application here, which introduces nonlinearity to the decision boundary. In the standard logistic regression, the link function is in general the logit function; however probit is also common for regression modeling of binary data, since at least they all keep the output between 0 and 1. Detailed exploration on this topic is out of the scope of this article. Acturally the reason why it is called logistic regression is because the link function is the logit-function.

Logit function apllies the same way to both binomial logistic regression, or standard logistic regression, and multinomial logistic regression, or softmax regression. The following math stuff is for explaining the fact that the former is a special case of the later.

The general equation for the probability of the dependent variable Y_i being fallen into the category c(in all K categories), given the sample(observation) X_i, is defined as:

$$Pr(Y_i = c) = \frac{e^{\beta_c * X_i}}{\sum_{k=1}^{K} e^{\beta_k * X_i}}$$

Since for all Y_i, $\sum_{k=1}^{K} Pr(Y_i = k)$ is 1, then there must be a certain $Pr(Y_i = c)$ that is determined by all the rest probabilities(when $k \neq c$). As a result, there are only $K-1$ separately specifiable coefficients(β) for each sample X_i.

According to the characterizations of the aforementioned equation, the probability remains the same if we subtract a constant from each item in the β vector. That is:

$$\frac{e^{(\beta_c + C) * X_i}}{\sum_{k=1}^{K} e^{(\beta_k + C) * X_i}} = \frac{e^{C * X_i} e^{\beta_c X_i}}{e^{C * X_i} \sum_{k=1}^{K} e^{\beta_k * X_i}} = \frac{e^{\beta_c X_i}}{\sum_{k=1}^{K} e^{\beta_k X_i}}$$

Then it is reasonable for us to make $C = -\beta_K$(alternatively any other k), resulting in the Kth item of the new coefficient for each vector β becomes 0(only $K-1$ categories are considered separately specifiable now).

$\beta'_1 = \beta_1 - \beta_K$ ..... $\beta'_{K-1} = \beta_{K-1} - \beta_K$ $\beta'_K = \beta_K - \beta_K = 0$

Hence we can transfer the first general equation into the form of:

$$Pr(Y_i = c) = \frac{e^{\beta'_c * X_i}}{e^{0 * X_i} + \sum_{k=1}^{K-1} e^{\beta'_k * X_i}} = \frac{e^{\beta'_c * X_i}}{1 + \sum_{k=1}^{K-1} e^{\beta'_k * X_i}}$$

When we have two categories, that is when the distribution of the dependent variable is binomial, or $K = 2$, we have(if the Kth category is when $Y_i = 2$):

$$Pr(Y_i = 1) = \frac{e^{\beta'_1 * X_i}}{1 + e^{\beta'_1 * X_i}} = \frac{1}{1 + e^{-\beta'_1 * X_i}}$$

, and

$$Pr(Y_i = 2) = 1 - Pr(Y_i = 1)$$

That is the familiar equations in binomial logistic regression.

In logistic regression the logit function transforms the linear predictor to discrete categorical outcomes. In other generalized linear models, for example Poisson distributions the log function is utilized. It matters a lot to choose the right link function for the right response distribution.

4 Objective Function

Now that we have related the features to the responses which follow a non-normal distribution($\mathbb{R}^n \rightarrow \mathbb{R}^k$) by the link function and others thereof, it's time to know how on earth we estimate the coefficients.

In the coefficient updating process we estimate the coefficients at each step, and the value is called an estimator. To judge if an estimator is good enough to represent the given data set(and to predict), an objective function is set. That is, if one particular estimator of a distribution of estimators makes the objective function maximum, given the same samples, then that estimator is coefficients we seek for. $\hat{\theta} = argmax_\theta L(\theta; x_1, ..., x_n)$. Such objective functions can be either as simple as the number of correct predictions(classification error) and mean squared error(also known as quadratic error) or as complex as matural information and cross entropy.

An objective function is the cost function to be optimized(or to analyses the performance) in for example linear regression and RNN and etc., just like the regret or economic cost in economics which you want to optimize(reduce costs, or in reverse increase profits). The mean squared error (MSE) or summed squared error(SSE) is the most widely used, intuitively which means the Euclidean distance $\sum N(y_i - x_i)^2$, the N can be any nonzero number which usually takes $1/2$ for calculation convenience.

For illustration, we can think of the model as a standard linear regression. θ is $\lambda_1, ..., \lambda_i, ..., \lambda_n$, and the objective function is the sum of the squared mean. Computing λ's is to do the linear regression, which means finding the best fit line(or multidimensional imaginary line) passing through a sequence of data points. the summed distance of all points to the line, in this situation, is the minimum.

Except the aforementioned objective function, there are plenty of others for example logistic loss, hinge loss and etc. They're utilized depending on various requirements. For example, when a distribution follows a normal distribution it is appropriate to use mean squared distribution, but if there are a few outliers absolute-difference the loss function is better.

4.1 Maximum Likelihood Estimation

In logist regression the objective function defaults to maximum likeihood. The likelihood is defined as $\mathcal{L}(\theta; x_1, ..., x_n) = f(x_1, ..., x_n; \theta)$, where f is a probability density function or probability massive function.

According to the Bayesian inference, $f(x_1, ..., x_n; \theta) = \frac{f(\theta; x_1, ..., x_n) * f(x_1, ..., x_n)}{f(\theta)}$ holds, that is $likelihood = \frac{posterior * prior}{evidence}$. Notice that the maximum likelihood estimate treats the ratio of prior to evidence as a constant, which omits the prior beliefs, thus MLE is considered to be a frequentist

technique(rather than Bayesian).

4.2 An example - normal distribution

Let's do the following simple exercise to warn up for the coming logistic regression: Estimate the μ and σ in the probability density of the normal distribution:

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2}2\pi} exp[-\frac{(x-\mu)^2}{2\sigma^2}]$$

If we have a sample of $x_1, ..., x_n$, the likelihood is the product of all $f(x_i;\mu,\sigma^2)$ since x_i's are independent to each other. To estimate μ and σ is to see the likelihood of all probabilities following the Gaussian distribution. We all learned in school that μ should be the mean and σ is the standard deviation of the sample. The larger the sample the μ will be closer to the mean. But let's prove that.

As we know logarithm is monotonic strictly increasing, thus maximizing the likelihood amounts to minimizing the negative log likelihood.

$$\mathcal{L}(\mu,\sigma;X) = \prod_{i=1} f(\mu,\sigma;x_i) \propto \sum_{i=1}^{n}[-log\sigma - log\sqrt[2]{2\pi} = \frac{(x-\mu)^2}{2\sigma^2}] = -nlog\sigma - nlog\sqrt[2]{2\pi} - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2$$

To find the μ and σ that make log maximum likelihood LML given the data,

$$LML =< \bar{\mu}, \bar{\sigma} >= argmax_{<\mu,\sigma^2>}[-nlog\sigma - nlog\sqrt{2}2\pi - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2] \text{where } \bar{\mu} \text{and} \bar{\sigma} \text{are the estimator we seek.}$$

We need to maximize the above by adjusting the μ and σ simultaneously.

Since the partial derivative of LML with respect to(w.r.t) parameter μ is

$$\frac{1}{2\sigma^2}\sum_{i=1}^{n} 2(x_i - u)$$

where σ is a constant, then we set the above equal to 0 to get the maximum. As a result $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$, the mean of the sample.

And apply the same to σ, we can obtain $\sigma = \frac{\sum_{i=1}^{n}(x_i-\bar{\mu})^2}{n}$, that is the standard deviation of the sample.

Then we can say that given the sample x_i we can set μ as the mean of the it and σ as the standard deviation of it to make sure that the model gives the greatest possibility to the sample. That is distribution defined by the model is the best one which can be used to predict the probability of a new sample drawed from that distribution.

4.3 Conditional likelihood and Softmax Regression

Assuming that the likelihood is determined both by the dependent variable and some independent ones, then the joint probability is called conditional likelihood.

For illustration, suppose that there are K elements(x_1 to x_K) determining which class a sentence belongs to($y = 1$, $y = 2$ ...$y = c$.. to $y = K$), then the probability is $Pr(Y = k|X;\theta)$. We have some

data collected beforehand as examples. For instance X_i when $x_1 = 1$, $x_2 = 4.1$, .. for $y_i = 1$(often noted as onehot: (1, 0,...,0)) and X_{i+1} when $x_1 = 0$, $x_2 = 1.2$, .. and etc then $y_i = 3$. The same applies to binomial logistic regression as explained aforementioned.

What we're going to optimize is the product of the conditional probabilities of all the examples(as we did for the normal distribution but conditioanl) provided the examples are independent to each other. It's realized by adjusting the coefficients θ_{ik} in softmax regresson for each class j using the backpropagation algorithm mentioned later.

To work out the joint probability we can take the log of it, resulting in the multiplication becomes addition, since the log function is monotonous and can make the calcultion much more simple and it's said to estimate rare events very well.

$$l_{Y=k}(\theta; Y|X) = \sum_{i=0}^{m} 1\{y^{(i)} = k\} \frac{e^{\theta_k^T X^{(i)}}}{\sum_{l=1}^{K} e^{\theta_k^T x^{(i)}}}$$

where k is the kth class; θ is the coefficient matrix; $1\{y^{(i)} = k\}$ is an indicator functon, and it is 1 when $y^{(i)} = k$ otherwise it is 0; $x^{(i)}$ is the ith observation; m is the total amount of observations in a epoch.

5 Backpropagation

For illustration, let's start by saying that we are going to find the critical points (maxima or minima) of a differentiable function(for exponential family the objective function is always differentiable). The normal method is to determine the points where the derivatives of them are 0. There are a couple of methods to achieve that goal: gradient descent, conjugate gradient and quasi-newton method. Due to many reasons we choose the gradient descent method. As for the reasons please search for the benchmarks of them.

$$b = a - \gamma \bigtriangledown F(a)$$

The gradient descent tries to find the deepest path approaching the extrema.

First we set the learning rate γ which determines the speed of the convergence, and then get the $\delta(J(\theta)$ as illustrated below). And then update each coefficient by the δ. And repeat the foreward and backward process until the update becomes close to zero or the iteration reaches a certain number. Bellow we detail the process to calculate the δ.

The total log conditional likelihood(LCL) is then the sum of the likelihood of all classes $\sum_{k=0}^{K} l_{Y=k}(\theta; Y|X)$.

$$J(\theta) = -\sum_{i=1}^{m} \sum_{k=1}^{K} 1\{y^{(i)} = k\} \log p(y^{(i)} = k \mid x^{(i)}; \theta)$$

Just imagine the result as a 2D matrix with each item(i, k) being a probability, each raw being a coefficient vector for the ith sample(or observation i) and each column being a category(k).

Let's expand p and transform the equation to:

$J(\theta) = -\sum_{i=1}^{m} \sum_{k=1}^{K} 1\{y^{(i)} = k\} \log \frac{e^{\theta_k^T x^{(i)}}}{\sum_{l=1}^{K} e^{\theta_l^T x^{(i)}}}$

And rearranging gives: $J(\theta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\}[\log e^{\theta_k^T x^{(i)}} - \log \sum_{l=1}^{K} e^{\theta_k^T x^{(i)}}]$

Let's take partial derivative of the likelihood w.r.t θ_k Since the negtive log likelihood is a convex function, we can determine its extrema by setting the partical derivative of the likeihood w.r.t every θ to zero.

The partial derivative of J with respect to θ_k is (treat $1\{y^{(i)} = k\}$ as a constant):

$\bigtriangledown_{\theta_k} J(\theta) = -\sum_{i=1}^{m} 1\{y^{(i)} = k\}[x^{(i)} - \underbrace{\frac{1}{\sum_{l=1}^{K} e^{\theta_k^T x^{(i)}}} e^{\theta_k^T x^{(i)}}}_{p(y^{(i)}=k|x^{(i)};\theta)} x^{(i)}]$

Note that for the kth category only one element in $\bigtriangledown_{\theta_k} \sum_{l=1}^{K} e^{\theta_k^T x^{(i)}}$ is nonzero, that is $e^{\theta_k^T x^{(i)}}$.

Then we get p back and have:

$$\bigtriangledown_{\theta_k} J(\theta) = -\sum_{i=1}^{m}[x^{(i)}(1\{y^{(i)} = k\} - p(y^{(i)} = k \mid x^{(i)}; \theta))]$$

Then the updated θ_k is $\theta_k + \bigtriangledown_{\theta_k} J(\theta)$

We do the backpropagation until a mount of iterations or until the $\bigtriangledown_{\theta_k} J(\theta)$ is less than a certain number. By setting m to 2 we can see that of the standrd logistic regression.

The code of the softmax regression is attached in the appendix. Detailed explanations are added as comments in the code.

A The code

```
def array2onehot(X_shape, array, start=1):
    """
    transfer a column to a matrix with each rwo being a onehot
    note that the array index starting from 1 rather than 0
    """
    array += 1 - start if start != 1 else 0
    onehot = np.zeros(X_shape)
    onehot[np.arange(X_shape[0]), array-1] = 1
    return onehot


class SoftmaxRegression(object):
    """
    The softmax regression from scratch with the help only from numpy
    input:
        samples
    outpu:
        coefficients
```

```
    """
    def __init__(self,
                 class_size,
                 learning_rate=0.00001,
                 penalty_rate=0.2,
                 max_iterations=None
                 ):
        # coefficients is a 2D matrix of the shape (len(feature), len(c
        self._learning_rate = learning_rate
        self._penalty_rate = penalty_rate
        self._class_size = class_size

        self._coefficients = None
                                # the coefficients is of the shape of (
        self._batch_size = None

    def stochastic_gradient_descent(self, X, Y, T):
        """how to tell if a network has been converged?"""

        # set the count for calculating the mean delta for each class
        count = np.zeros(self._class_size)
        # initialize the delta matrix
        delta = np.zeros(self._coefficients.shape)
        # iterate all samples in the mini batch
        for i in range(len(X)):
            k = list(T[i]).index(1)
            count[k] += 1
            # for the kth class add the delta
            # delta_{sample_i} =
            # x^{(i)}(1\{y^{(i)}=j\}-p(y^{(i)}=j|x^{(x)};\theta))
            delta[:, k] += X[i] * sum(np.multiply(T[i], Y[i]))
        # except for those classes whose coefficients remain the same
        tmp = self._coefficients
        for i in count:
            if i == 0:
                tmp[:, i] = 0
        # calculate the mean delta for each class in the batch
        delta = np.nan_to_num(delta / count) - 2 * self._penalty_rate *
        # update the coefficient matrix
```

```
        self._coefficients -= delta
        return np.sum(delta**2)

    def get_cost(self, Y, T):
        """
        function:
            work out the cost(cross entropy) of the mini batch
        input:
            X and T, the input samples and the corresponding targets
        output:
            cost = - \[\sum_{i=1}^m\sum_{k=0}^1 1\{y^{(i)}=k\}
            \logP(y^{(i)}=k|x^{(i)}; \theta)\]
        """
        cost = 0
        for i in range(len(Y)):
            cost += np.sum(
                np.multiply(np.log(np.e ** Y[i] / np.sum(np.e ** Y[i]))
            )
        return cost

    def start_training(self, data, epsilon, max_iterations):
        """
        input:
            data, the X's and T as the last item for each row
            epsilon - float, the tolerance at which the algorithm will
            max_iterations - int, tells the program when to terminate
        """
        # init the coeffeicnet matrix, plusing one for the bias
        self._coefficients = np.random.rand(self._class_size, data.shap
        # get the number of steps
        steps = int(np.ceil(len(data) / self._batch_size))
        converged = False
        iterations = 0
        while True:
            for i in range(steps):
                if converged or iterations > max_iterations:
                    return
                # the ith batch
                batch = data[i:i+1]
```

```
                # ones as bias for the input X's
                ones = np.ones(batch.shape[0])
                # add the bias as the first column to the input X as th
                # input
                # x = \[\hat x, 1\]
                X = np.column_stack(ones, batch[:, :-1])
                # run the forward process and get the output as the Y
                Y = self.forward(X)
                # convert the targets to the onehot form
                T = array2onehot(X.shape, batch[:, -1])
                gradient_sum_squares = self.stochastic_gradient_descent
                iterations += 1
                if gradient_sum_squares < epsilon ** 2:
                    converged = True

    def forward(self, X):
        Y = np.multiply(X, self._coefficients.T)
        return Y

    def predict(self, X):
        Y = self.forward(X)
        classes = []
        for i in range(len(Y)):
            normalized_probability = np.e ** Y[i] / np.sum(np.e ** Y[i]
            classes.append(
                list(normalized_probability).index(max(normalized_proba
        return np.array(classes) + 1
```