Softmax and robust regressions

In the previous note we motivated logistic regression as a modification of linear regression that works better for binary labels. The main trade-off was that logistic regression can no longer use a standard linear least squares solver, but instead runs an iterative optimizer to convergence.

In this note we give another two examples where we modify the least squares cost. In each case we write down alternative models of our target outputs, find a suitable cost function, and then differentiate it so that we can fit the parameters with a gradient-based method.

1 Softmax regression

Softmax[1] regression is a generalization of logistic regression to cases with more than two labels. Some textbooks will simply call this generalization "logistic regression" as well.

If there are K possible classes, we will model labels with a length-K one-hot encoding:

$$\mathbf{y} = [0\;\;0\;\;\ldots\;\;0\;\;1\;\;0\;\;\ldots\;\;0], \tag{1}$$

where if data-point n has label c, then $y_k^{(n)} = \delta_{kc}$, where δ_{kc} is a Kronecker delta.

We could attempt to predict the elements of $\mathbf{y}$ one at a time, with K separate one-vs-rest classifiers. We would need a method to combine the outputs of these classifiers. Instead we'll fit a system that is explicit about how to predict the whole vector.

We will make a system with parameters W that outputs a vector $\mathbf{f}$. We know that the target vector $\mathbf{y}$ contains a single one. We interpret the vector output $\mathbf{f}$ as a probability distribution over which bit in the target vector $\mathbf{y}$ is turned on, giving the model:

$$P(y_k\!=\!1\,|\,\mathbf{x}, W) = f_k(\mathbf{x}; W). \tag{2}$$

We can start by using a separate regression vector $\mathbf{w}^{(k)}$ for each class, and create a positive score, by exponentiating a linear combination of features:

$$s_k = e^{(\mathbf{w}^{(k)})^\top \mathbf{x}} \tag{3}$$

Moving the features $\mathbf{x}$ in the direction $\mathbf{w}^{(k)}$ increases the score for class k. In particular, if $w_d^{(k)}$ is positive, then large values of x_d give class k a large score.

However, probabilities have to add up to one, so our output vector is normalized:

$$f_k = \frac{s_k}{\sum_{k'} s_{k'}} = \frac{e^{(\mathbf{w}^{(k)})^\top \mathbf{x}}}{\sum_{k'} e^{(\mathbf{w}^{(k')})^\top \mathbf{x}}}. \tag{4}$$

Here k' is a dummy index used to sum over all of the classes. Our vector-valued function $\mathbf{f}$ is parameterized by $W = \{\mathbf{w}^{(k)}\}_{k=1}^K$. The classes now *compete* with each other. If $w_d^{(k)}$ is large, then large x_d doesn't necessarily favour class k. The normalization means that the probability of class k could be small if $w_d^{(j)}$ is larger for another class j.

If W is a $K\!\times\!D$ matrix, we might write the above function as $\mathbf{f}(\mathbf{x}; W) = \mathrm{softmax}(W\mathbf{x})$.

[1] I believe the term *softmax* was coined by John S. Bridle (1990). Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In: F.Fogleman Soulie and J.Herault (eds.), Neurocomputing: Algorithms, Architectures and Applications, Berlin: Springer–Verlag, pp. 227–236.

1.1 Fitting the model

We want our function evaluated at a training input, $\mathbf{f}(\mathbf{x}^{(n)})$, to predict the corresponding target $\mathbf{y}^{(n)}$. At best, they are equal, in which case we confidently predict the correct label. However, if the labels are inherently noisy, so that the same input could be labelled differently, making perfect predictions every time is impossible.

Matching targets $\mathbf{y}$ and function values $\mathbf{f}$ by least squares would be *consistent*: if a setting of the parameters can make $\mathbf{f}$ match the probabilities of the labels under the real process that is generating the data, then we will fit those parameters given infinite data. However, "maximum likelihood" estimation is also consistent, and for well-specified models has faster asymptotic convergence. Compared to least squares, maximum likelihood heavily penalizes confident but wrong function values, which may or may not be seen as desirable. (Consider the maximum loss that can be obtained on a single example under the two cost functions.)

We first find the gradients of the model's log-probability of a single observation: a label c when the input features are $\mathbf{x}$:

$$\log P(y_c\!=\!1\,|\,\mathbf{x}, W) = \log f_c = (\mathbf{w}^{(c)})^\top\mathbf{x} - \log\sum_{k'} e^{(\mathbf{w}^{(k')})^\top\mathbf{x}} \tag{5}$$

$$\nabla_{\mathbf{w}^{(k)}} \log f_c = \delta_{kc}\mathbf{x} - \frac{1}{\sum_{k'} e^{(\mathbf{w}^{(k')})^\top\mathbf{x}}}\,\mathbf{x}\,e^{(\mathbf{w}^{(k)})^\top\mathbf{x}} \tag{6}$$

$$= (y_k - f_k)\,\mathbf{x}. \tag{7}$$

In the last line we have substituted $y_k\!=\!\delta_{kc}$ and the definition of f_k.[2]

To apply stochastic gradient ascent[3] on the log-likelihood we would take a training example $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$ and move the weight vector for each class parallel to the input $\mathbf{x}^{(n)}$. The size and sign of the move depends on the disparity between the binary output $y_k^{(n)}$ and the prediction f_k.

Alternatively we could use an optimizer that uses a batch of examples, requiring the gradient

$$\nabla_{\mathbf{w}^{(k)}} \sum_n \log f_{c^{(n)}}^{(n)} = \sum_n (y_k^{(n)} - f_k(\mathbf{x}^{(n)}))\,\mathbf{x}^{(n)}, \tag{8}$$

or minus that value for the gradient of the batch's negative log probability.

1.2 Redundant parameters and binary logistic regression

For the special case of two classes, the softmax classifier gives:

$$P(y\!=\!1\,|\,\mathbf{x}, W) = \frac{e^{(\mathbf{w}^{(1)})^\top\mathbf{x}}}{e^{(\mathbf{w}^{(1)})^\top\mathbf{x}} + e^{(\mathbf{w}^{(2)})^\top\mathbf{x}}} \tag{9}$$

$$= \frac{1}{1 + e^{(\mathbf{w}^{(2)} - \mathbf{w}^{(1)})^\top\mathbf{x}}} = \sigma\big((\mathbf{w}^{(1)} - \mathbf{w}^{(2)})^\top\mathbf{x}\big). \tag{10}$$

The prediction only depends on the vector $(\mathbf{w}^{(1)} - \mathbf{w}^{(2)})$. In logistic regression we normally fit that single vector directly as "$\mathbf{w}$".

We can fit two vectors, as in the softmax classifier formulations. The parameters are not well specified: adding a constant vector $\mathbf{a}$ to both class's weight vectors gives the same

[2] It may take some time to digest all of the notation here. We index into the vector of function outputs using the observed label c. We didn't use k for this index so we can use k to index the weight vector we are computing gradients for. We need the gradients for $k\!\neq\!c$ as well as $k\!=\!c$. The dummy index k' is used because we need to consider all of the weight vectors $\{\mathbf{w}^{(k')}\}_{k'=1}^{K}$, when computing the gradient with respect to a particular weight vector $\mathbf{w}^{(k)}$.

[3] Optimization texts normally talk about gradient *descent* and minimizing functions. Gradient descent on the negative log-likelihood amounts to the same thing. We take a negative step in the direction of the gradient of the negative likelihood, and the two negatives cancel.

predictions. However, if we add a regularization term to our cost function, the cost function for this model will be minimized by a unique setting of the weights.

Alternatively we could set the weight vector for one of the classes to zero, and fit the rest of the parameters. For example, recovering logistic regression by setting $\mathbf{w}^{(2)} = \mathbf{0}$.

2 A Robust Model

Real data is often imperfect. The labels attached to training examples, may be incorrect. The logistic/softmax models can model noisy labels by reducing the magnitude of the weight vector. However, a corrupted example with an extreme input $\mathbf{x}$ could force the weight vector in some direction to be small, even if many other examples, with less extreme $\mathbf{x}$, indicate otherwise. This problem occurs most readily for the negative log-likelihood loss, because a single example can have arbitrarily large loss.

There are various solutions to this issue. We could limit the magnitude of the feature vector: some practitioners only use binary features, or scale their features to have length one. We could try to identify outliers and discard them. We could intervene in the optimization procedure, and limit the size of the update from any individual example, or group of examples.

In the rest of this section we take a probabilistic modelling approach. We take the existing model of the labels, and modify it to reflect the fact we think corruptions could happen. We can then derive a new expression for the log probability and proceed as before. The lesson isn't necessarily to use this particular robust classifier — I haven't seen it widely used in practice for generic machine learning tasks. The lesson is that writing down a probabilistic model for your data can suggest ways to improve it. After modifying the model we immediately get a new negative log-likelihood cost function, we can then mechanically obtain gradients and get a new learning algorithm quickly.

We will assume that a binary choice $m \in \{0, 1\}$ was made for each observation, about whether to corrupt the label:

$$P(m) = \text{Bernoulli}(m;\, 1-\epsilon) = \begin{cases} 1-\epsilon & m = 1 \\ \epsilon & m = 0. \end{cases} \tag{11}$$

With probability $(1-\epsilon)$ the model sets $m=1$ and generates a label using the normal logistic regression process (to keep notation simple, I'll assume binary classification). Otherwise, with probability ϵ, the model sets $m=0$ and picks the label uniformly at random:

$$P(y=1 \,|\, \mathbf{x}, \mathbf{w}, m) = \begin{cases} \sigma(\mathbf{w}^\top \mathbf{x}) & m = 1 \\ \frac{1}{2} & m = 0. \end{cases} \tag{12}$$

If we knew the indicator variable m for each example, we'd use it as a mask: we'd ignore the irrelevant points where $m=0$, and fit the remaining points as usual. However, because we don't observe the indicator variable m, we need to write an expression for the probability of just the label we observe. To include the unknown choice m on the right-hand side, we need to "marginalize" (sum) it out using the sum rule:

$$\begin{aligned} P(y=1 \,|\, \mathbf{x}, \mathbf{w}) &= \sum_{m \in \{0,1\}} P(y=1, m \,|\, \mathbf{x}, \mathbf{w}) && (13) \\ &= \sum_{m \in \{0,1\}} P(y=1 \,|\, \mathbf{x}, \mathbf{w}, m)\, P(m) && (14) \\ &= (1-\epsilon)\sigma(\mathbf{w}^\top \mathbf{x}) + \epsilon \frac{1}{2}. && (15) \end{aligned}$$

In the second line we use the product rule. In general the last term would be $P(m \,|\, \mathbf{x}, \mathbf{w})$, but in this model we decided to make the choice independent of the input position and the weights.

Now we need the gradients of the log probability. As stated in the previous note, it's always possible to get these at reasonable cost. I rearranged them by hand to see if they were interpretable. As in the previous logistic regression note, I use $\sigma_n = \sigma(z^{(n)}\mathbf{w}^\top\mathbf{x}^{(n)})$ as the probability of getting the nth label correct under standard logistic regression, where $z^{(n)} = (2y-1)$ is a $\{-1, +1\}$ version of the label. I obtained the expression:

$$\frac{\partial \log P(z^{(n)} \,|\, \mathbf{x}^{(n)}, \mathbf{w})}{\partial \mathbf{w}} = \frac{1}{1 + \frac{1}{2}\frac{\epsilon}{1-\epsilon}\frac{1}{\sigma_n}} \nabla_{\mathbf{w}} \log \sigma_n, \tag{16}$$

where $\nabla_{\mathbf{w}} \log \sigma_n = (1-\sigma_n) z^{(n)} \mathbf{x}^{(n)}$ was the gradient of the log-probability for the original logistic regression model. As a check, we recover the original model when $\epsilon = 0$. If we implemented this equation we could also check it with finite differences (see previous note).

In the original model, the gradient is large for an extreme $\mathbf{x}$ where the label is poorly predicted. In the robust model, the contribution from the gradient is small if the probability of being correct, σ_n, is much smaller than the probability of a corruption, ϵ. Really extreme outliers will be nearly ignored.

We could set ϵ by hand, for example to 0.01. We could try a grid of settings. Or we could optimize ϵ along with $\mathbf{w}$ using gradient methods. Most gradient-based optimizers need the parameters to be unconstrained, whereas ϵ is constrained to $[0, 1]$. We can state $\epsilon = \sigma(b)$, and optimize $b = \text{logit}(\epsilon) = \log \frac{\epsilon}{1-\epsilon}$ instead.

3 Some general principles

Transforming the range of numbers that a quantity can take is frequently useful in machine learning. So far in the course we have used exp() to force a number to be positive, and log() to make a positive number unconstrained. We used the logistic sigmoid $\sigma()$ to make numbers lie in $[0, 1]$, and now we've used its inverse, the logit(), to make values in $[0, 1]$ become unconstrained.

Where easily possible, we transform data to have a sensible distribution for a model. Sometimes we have to transform the model output to better match the data. Then if we apply an appropriate differentiable loss function, we can match the model to data with gradient methods. When we have identified a probabilistic model, the loss function can be negative log-probability. We might then see obvious ways that the model could be wrong, which may help us to derive improvements.

3.1 Convexity

We have a recipe for writing down a cost function. Identify a probabilistic model of the data, and minimize the negative log probability. But will we be able to minimize this cost function?

If the cost function is *convex* the numerical optimization literature provides methods where we can guarantee finding parameters that minimize the cost as much as possible. A function is convex if a straight line drawn between any two points on the cost function surface never goes below the surface. Algebraically:

$$C(\alpha\mathbf{w} + (1-\alpha)\mathbf{w}') \;\le\; \alpha C(\mathbf{w}) + (1-\alpha)C(\mathbf{w}'), \quad \text{for any } \mathbf{w}, \mathbf{w}', 0 \le \alpha \le 1. \tag{17}$$

If the inequality above is strict ($<$ rather than $\le$), the function is strictly convex and has one unique minimum. Otherwise there could be a connected convex set of parameters, where the cost is equal, but as small as possible.

From the definition, it's easy to show that the sum of any convex functions is convex. So if the loss function for an individual example is convex, then our loss function will be too.

For logistic regression, the loss, $-\log\sigma((2y-1)\mathbf{w}^\top\mathbf{x})$, is a convex function of $\mathbf{w}$. We can guarantee we'll find the maximum likelihood solution to within a small numerical tolerance.

For the robust regression model, $-\log(\epsilon/2+(1-\epsilon)\sigma(z\mathbf{w}^\top\mathbf{x}))$ is not convex for $\epsilon > 0$. Simply plot an example of changing one weight to see a counter example. So, we can't provide the same guarantees about being able to find the minimum of this loss. We can however try to reduce the loss function using gradient methods, and in practice we're likely to find parameters with reasonable loss, that generalize usefully to new examples. We just can't guarantee that a better optimizer wouldn't have found better parameters.

3.2 Why log probability?

You might have been wondering why we use the *log* probability to score our models, rather than evaluating and differentiating the probability itself.

One reason is that the log probability of a training set is a sum (rather than product) over examples. That usually makes the maths more convenient. Also we can approximate the sum with a subset to perform stochastic gradient descent.

We also use log probabilities to avoid numerical underflow. Most numerical software (including Matlab and NumPy) works with IEEE floating point. Even with "double" precision, the probability of a sequence of 1,000 coin tosses, 0.5^{1000}, underflows to zero. In contrast, we can compute $1000\log 0.5$ with no difficulty.

Finally, the log probability is more appropriate for numerical optimizers. As an example, if we fit the mean of a Gaussian's negative log PDF by gradient methods, we have a quadratic cost function, which is convex and the ideal cost function for most gradient-based optimization methods. The PDF itself is not convex and has values in a narrow numerical range near zero in the tails. The log-likelihoods for non-Gaussian models, such as logistic regression, are also sometimes approximately quadratic when there are many datapoints. Even if the model is not convex (like the robust model), optimizing the log-probability (which might look nearly convex, at least near an optimum) usually works better.

4 Check your understanding

Can you show that any softmax classifier with a weight vector for each class has an equivalent set of parameters with the weights for the final class set to zero, $\mathbf{w}^{(K)}=\mathbf{0}$?

How would you perform stochastic gradient descent on the models in this note if you wanted to apply L2 regularization of the weights?

A complete description of an algorithm includes any initialization conditions. How would you initialize the weights? Does it matter? What might happen numerically if you initialize with weights that have extremely large magnitude?

For keen students: We could write down a model that states that the standard logistic regression model with no label noise is correct for our data, but that the inputs $\mathbf{x}$ are noisy. Why did I not go that route? That is, if you write down a model that says $\mathbf{x}$ could be corrupted by noise, do you get stuck? If so, where and why?

5 For keen students

The readings for the previous note already mentioned that by reading about "generalized linear models" you could see more examples of probabilistic models for the outputs, beyond those we've seen.

Those who already know something about neural networks (or those returning to these notes later) might also think about adding probabilistic models for different types of output to neural networks:

- Multi-modal real-valued distributions with “Mixture Density Networks” (Bishop, 1994).
- Count data, such as in sales forecasting, using a negative-binomial model on the output layer. “DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks” (Flunkert et al., 2017).