Week 5: Logistic Regression & Neural Networks

Instructor: Sergey Levine

1 Summary: Logistic Regression

In the previous lecture, we covered logistic regression. To recap, logistic regression models *and* optimizes the conditional log likelihood:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \log p(y^i|\mathbf{x}^i, \mathbf{w}) = \sum_{i=1}^{N} \left(y^i h(\mathbf{x}^i) \cdot \mathbf{w} - \log[\exp(h(\mathbf{x}^i) \cdot \mathbf{w}) + 1] \right).$$

This likelihood cannot be optimized analytically, so instead we compute a gradient, according to

$$\nabla \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} h(\mathbf{x}^i) \left[y^i - p(y = 1|\mathbf{x}^i, \mathbf{w}) \right].$$

Then, we use an algorithm called gradient ascent to optimize $\mathcal{L}(\mathbf{w})$, by repeatedly performing the following operation:

$$\mathbf{w}^{(j+1)} = \mathbf{w}^{(j)} + \alpha \nabla \mathcal{L}(\mathbf{w}^{(j)}).$$

This performs MLE to determine $\mathbf{w}$. We can also use MAP, where we put a prior on $\mathbf{w}$. For example, a Gaussian prior produces the likelihood

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \left(y^i h(\mathbf{x}^i) \cdot \mathbf{w} - \log[\exp(h(\mathbf{x}^i) \cdot \mathbf{w}) + 1] \right) + \lambda \sum_{k=1}^{|h(\mathbf{x})|} \mathbf{w}_k^2.$$

We can also use logistic regression to classify multiple different classes. If $y \in \{1, \dots, L_y\}$, we can use

$$\log p(y = j|\mathbf{x}, \mathbf{W}) = \frac{\exp(h(\mathbf{x}) \cdot \mathbf{w}_j)}{\sum_{j'=1}^{L_y} \exp(h(\mathbf{x}) \cdot \mathbf{w}_{j'}},$$

where the parameters now correspond to a weight matrix $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_{L_y}]$.

2 Comparison with Naïve Bayes

Logistic regression solves the same kind of problem as naïve Bayes, but uses a different object and a different estimation procedure. The two algorithms have slightly different strengths and weaknesses, and present some interesting tradeoffs. First, in the case where $\mathbf{x} \in \mathbb{R}^K$ (and $h(\mathbf{x}) = \mathbf{x}$)[1] – that is, when all features are real-valued – it can be proven that naïve Bayes and logistic regression are representationally equivalent, meaning that for every logistic regression classifier, we can find a naïve Bayes classifier that always outputs the same answer, and vice-versa. This requires naïve Bayes to use a Gaussian condition distribution for each $p(x_k|y)$, and for all of these Gaussians to have the same variance (but a different mean).

This *representational equivalence* makes it convenient to compare naïve Bayes and logistic regression. We won't derive the representational equivalence in detail: it's quite straightforward to obtain it from Bayes rule, and you may see this on a homework exercise or (hint!) an exam question.

Question. What assumptions does naïve Bayes make that logistic regression does not?

Answer. Naïve Bayes assumes that the features are independent of one another. This is extremely convenient, because it allows us to estimate each conditional $p(x_k|y)$ separately and in closed form. However, this assumption can be quite simplistic.

Question. As the number of features K increases, which model is likely to overfit more, logistic regression or naïve Bayes?

Answer. One way to think about this question is to imagine what happens if a certain feature is replicated with a bit of noise added to it. For example, if we have K features that are all equal in expectation, but corrupted with different Gaussian noise. Naïve Bayes estimates each feature distribution independently, so no matter how many features we have, naïve Bayes will perform about the same. However, logistic regression will get worse and worse as the number of features increases: as the number of features K exceeds the number of datapoints N, it will be easy for logistic regression to fit to the noise in the data. In the extreme case, if we imagine that each feature is on average 0 but randomly $+1$ or -1 with some (equal) but small probability, as $K >> N$, we can find for each sample a feature that is $+1$ for *only that sample*, making it trivially easy to overfit.

Based on this intuition, we can make a few conclusions about the relative of performance of logistic regression vs. naïve Bayes. Indeed, these conclusions can be proven to hold theoretically, though proving this is outside the scope of this class. Naïve Bayes will tend to exhibit more bias, since the independence assumption corresponds to a simpler model. That means that with less data, naïve Bayes will overfit less, but as the amount of data increases, logistic

[1] In this section, I will drop the convention of using $h(\mathbf{x})$ to represent the features and simply denote them with $\mathbf{x}$. As we've learned before, this doesn't change of the math, it's just done here to match the notation in naïve Bayes.

regression will eventually perform better, since it doesn't make the simplifying assumption of feature independence. Correspondingly, naïve Bayes will continue to perform well as the number of features increases, while logistic regression will become more vulnerable to overfitting.

So, in short: naïve Bayes generally needs less data, while logistic regression will generally find a better solution if it doesn't overfit. In the case that the features are truly independent and the amount of data is near infinite, it can be shown that both methods will find the same solution if naïve Bayes uses Gaussian conditionals with input-independent variance.

The other consideration, which is more obvious from inspecting the learning algorithms, is that naïve Bayes classifiers can be learned extremely efficiently and quickly, while logistic regression requires a comparatively more complex gradient ascent procedure. This is typically not a big deal, but naïve Bayes sometimes has a bit more flexibility than logistic regression. For example, it's easy with naïve Bayes to handle missing data: if some record is missing a certain feature x_k, we simply ignore that record when estimating $p(x_k|y)$, and during inference, we can easily ignore features that are unknown. The same is not possible with logistic regression.

3 From Logistic Regression to Neural Networks

Although we can often choose very expressive features $h(\mathbf{x})$ for logistic regression, we sometimes don't know exactly which features are needed to solve a given classification problem. If we just use $h(\mathbf{x}) = \mathbf{x}$, the linear logistic regression classifier can't solve certain otherwise simple problems. For example, imagine that $\mathbf{x} = [x_1, x_2]$, where $x_1, x_2 \in \{0, 1\}$, and $y = x_1$ XOR x_2, such that $y = 0$ if $x_1 = x_2$ and $y = 1$ if $x_1 \neq x_2$. If we have $h(\mathbf{x}) = \mathbf{x}$, no logistic regression classifier can solve this task. To understand why, it's enough to draw a plot with x_1 and x_2 on the axes: the positive and negative examples cannot be separated by a single line. However, we can observe that we can write the XOR function as the following logical expression:

$$y = (x_1 = 1 \text{ and } x_2 = 0) \text{ or } (x_1 = 0 \text{ and } x_2 = 1).$$

Note that the or operator can actually be implemented by a linear classifier, so if we had two features $h_1 = (x_1 = 1 \text{ and } x_2 = 0)$ and $h_2 = (x_1 = 0 \text{ and } x_2 = 1)$, we could set

$$p(y|\mathbf{x}) = \frac{\exp(h_1 + h_2 - 0.5)}{\exp(h_1 + h_2 - 0.5) + 1},$$

which can be implemented using logistic regression (though we need a bias feature). So can we learn to extract h_1 and h_2 automatically? This turns out to be quite easy if we have another "layer" of logistic regression that attempts to output v_1 and v_2 (separately) from x_1 and x_2. For example, we can use:

$$h_1 = \frac{\exp(8x_1 - 4x_2 - 6)}{\exp(8x_1 - 4x_2 - 6) + 1}.$$

Note that $h_1 > 0.5$ when ($x_1 = 1$ and $x_2 = 0$) and less than 0.25 otherwise, which means that $h_1 + h_2 > 0.5$ only when ($x_1 = 1$ and $x_2 = 0$) or ($x_1 = 0$ and $x_2 = 1$). This idea of using intermediate "layers" of logistic regression to automatically construct features results in a model called a neural network.

Each layer in a neural network has weights $\mathbf{W}^{(\ell)} = [\mathbf{w}_1^{(\ell)}, \ldots, \mathbf{w}_{|h^{(\ell)}|}^{(\ell)}]^T$, where each weight matrix has a number of rows equal to the number of "features" (referred to as hidden units or neurons) at that layer. Note that by convention, the weight vectors are rows of this matrix, so that

$$h^{(\ell)} = \sigma(\mathbf{W}^{(\ell)} h^{(\ell-1)}),$$

where $\sigma(z)$ is a *nonlinearity*, which in the case of logistic regression is the logistic function

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

By convention, we automatically add a bias feature to each layer and denote it separately from the weight matrix at $\mathbf{b}^{(\ell)}$, such that

$$h^{(\ell)} = \sigma(\mathbf{W}^{(\ell)} h^{(\ell-1)} + \mathbf{b}^{(\ell)}).$$

Question. How many hidden units does the XOR network have?

Answer. The XOR network has two hidden units, h_1 and h_2.

Question. What are the first layer weights in the XOR network?

Answer. Since there are two hidden units, we need a weight matrix $\mathbf{W}^{(1)}$ with two rows, and a 2D bias vector $\mathbf{b}^{(1)}$. Based on the equations for h_1 and h_2, we have:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 8 & -4 \\ -4 & 8 \end{bmatrix} \qquad \mathbf{b}^{(1)} = \begin{bmatrix} -6 \\ -6 \end{bmatrix}.$$

Question. What are the second layer weights in the XOR network?

Answer. There is only one output, and therefore only one weight vector:

$$\mathbf{W}^{(2)} = [1\ 1] \qquad \mathbf{b}^{(2)} = -0.5.$$

In general, we could have different nonlinearity functions σ on the hidden layers in the network. The sigmoid or logistic function is a popular choice. Other popular choices include the hyperbolic tangent and, more recently, the rectified linear unit, given simply by $\mathrm{ReLU}(z) = \max(z, 0)$. We will discuss this briefly later in the week.

Question. For the neural network presented above, what is the objective and what is the model modeling?

Answer. Just like with logistic regression, we can optimize the neural network with respect to the conditional likelihood

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(y^i | \mathbf{x}^i, \theta),$$

where the parameters θ are given by $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}\}$. We can also use neural networks for multiclass classification, regression, and other estimation problems. Typically, neural networks optimize a conditional objective, though it is also possible to build generative neural networks (that will not be covered in this class).

We'll discuss algorithms for training neural networks next time, along with some discussion of their strengths and weaknesses.