Advanced Regression

CS 360 Machine Learning
Week 5, Day 1

February 20, 2024

Contents

1 Stochastic gradient descent

Recall that in stochastic gradient descent we think about starting in a random location and then walk towards what we hope is a minimum of the function. We're trying to minimize a function that is capturing a notion of error. We have some learning rate or learning step (denoted α) that helps us walk along the function towards that minimum. Ideally we do this quickly, but not so quickly that we overshoot. If the learning rate is too small, we can get stuck somewhere that's not the minimum. If the learning rate is too high, we might overshoot the minimum (bounce off the walls of the function) and we might not converge. In reality, there may also be more than one minimum including both local minima and a global minimum. We might even end up stopping on a plateau if the stopping criteria is that the result isn't changing too much. We need to be aware that we might not end up in the global minimum using gradient descent.

The classic stochastic gradient descent algorithm is:

```
set w = 0 vector
while cost J(w) still changing (or max iter reached):
    shuffle data points
    for i = 1 ... n:
        w = w - alpha (derivative of J(w) wrt xi)
    store J(w)
```

Note that we have a negative sign in this algorithm that tells us to go in the opposite direction from the derivative — why do we do that? Recall that we're trying to minimize $J(w)$. In order to move towards the minimum of the function $J(w)$, we go in the opposite direction of its derivative.

Why do we need to shuffle the data points? It's been shown to improve convergence in practice. Going in the same data order each time can bias you towards specific solutions if they're in a specific order (e.g., sorted) and empirically shuffling the order has been shown to improve the convergence speed. It's known as "stochastic" because of this shuffle step. Note that no matter what you still use the full dataset, just in different orders.

There are multiple types of gradient descent:

- **Batch gradient descent** — The derivative is taken over the whole dataset; you consider the full training dataset before making a weight update. This is generally slower and requires a smaller learning rate, but will work. The weights aren't updated frequently enough for it to be fast.
- **Stochastic gradient descent** — Shuffle the data and make a weight update after each training example (as shown above). This updates the weights more frequently (too frequently) than the batch version with potentially larger step sizes.
- **Mini-batch gradient descent** — Update after a "mini-batch" of training examples (e.g., 50 training examples). This subset is chosen randomly (shuffled). An *epoch* is one pass through the entire training dataset based on the chosen mini-batch size, seeing every example exactly once. This version of gradient descent is "just right" in the sense that it updates weights after each mini-batch which is more frequently than batch gradient descent and less often than stochastic gradient descent.

Note that you don't look at the data more or less with any of these approaches, you just update the weights more or less frequently. A comparison between these methods can be visualized considering the gradient descent paths (see the textbook figure from Chapter 4).

2 Logistic Regression

Recall that the model we're considering in general is

$$h_{\vec{w}}(\vec{x}) \;=\; p(y = 1 \mid \vec{x})$$

This is sort of like the posterior probability, but is a model for that. Given our example $\vec{x}$ the goal is to determine the probability of $y = 1$. Again, we're considering a binary classification problem ($y \in \{0, 1\}$) for now.

In a logistic regression model we say that:

$$h_{\vec{w}}(\vec{x}) \;=\; p(y = 1|\vec{x}) \;=\; \frac{1}{1 + e^{-\vec{w}\cdot\vec{x}}}$$

and recall that:

$$\vec{w} \cdot \vec{x} = w_0 + w_1 x_1 + \ldots w_p x_p$$

where w_0 is a "fake" 1. Here, we are considering how to fit a model (i.e., training time), so recall that we know $\vec{x}$ (the example) and it's associated label y in this case and don't know the weights $\vec{w}$.

How do we find $\vec{w}$? We use the likelihood function:

$$L(\vec{w}) = \prod_{i=1}^{n} \underbrace{h_{\vec{w}}(\vec{x}_i)}_{\text{Probability the label (y_i) is 1}}{}^{y_i} \; \overbrace{(1 - h_{\vec{w}}(\vec{x}_i))}^{\text{Probability the label (y_i) is 0}}{}^{1-y_i}$$

Recall the definition of $h_{\vec{w}}(\vec{x}_i)$ from above. Given this, the first part inside the product is the probability the label is 1 for a specific $\vec{x}_i$, and the second part gives the probability of label 0. Note that for each y_i we'll only have one of these terms contributing to the product, since the other will be 0.

Overall we want to maximize the likelihood. This is what a model does! If the true label is 0, we want the probability of predicting 0 to be high. The unknown is $\vec{w}$ – this is part of the training process, and in this case the values of y and $\vec{x}$ are known. If we imagine that this data came from some true model, we want to maximize the chances of getting these values back at a later testing time.

2.1 Maximum likelihood estimation example

Consider a coin flip example with outcomes 0 or 1 meant to mirror the logistic regression outcomes but where we only have y values. Suppose that it's a biased coin where the probability of getting a 1 is p and the probability of 0 is $1 - p$. The number of flips is n. Consider the vector of coin flips:

$$\vec{y} = [0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 0]$$

I want to solve for p. The likelihood of p, denoted $L(p)$ is:

$$L(p) = (1-p)(1-p)pp(1-p)p(1-p)p(1-p)(1-p)$$

based on the specific y vector above. This can be rewritten as:

$$L(p) = p^4(1-p)^6$$

A maximum likelihood in general is focused on explaining the observed data, i.e., on finding p based on these observations. I can describe the above likelihood more generally as:

$$L(p) = \prod_{i=1}^{n} p^{y_i}(1-p)^{1-y_i}$$

Note that this idea is very similar to logistic regression!

Consider the handout. The number of 1s in the data can be denoted as $\bar{y}n$ where $\bar{y}$ denotes the mean. Similarly, the number of 0s can be written as $(1-\bar{y})n$. A few additional hints for the handout:

- $\log(b^a) = a \log b$
- When $f(x) = \log x$ then $f'(x) = \frac{1}{x}$

When we consider how to solve for the likelihood function $L(p)$ we can instead take the log of the function (denoted $\ell(p)$ and referred to as the *log likelihood*). Maximizing $\ell(p)$ will still allow us to go in the right direction in terms of maximizing the original function $L(p)$. Considering the $L(p)$ as defined above, and using the above observations and hints, we get:

$$\ell(p) = \underbrace{n\bar{y}}_{\text{Number of 1s}} \log p + \overbrace{n(1-\bar{y})}^{\text{Number of 0s}} \log(1-p)$$

In order to maximize the (log) likelihood function, we next take the derivative of $\ell(p)$ with respect to p, denoted $\ell'(p)$, and set it equal to 0. Solving for p, this becomes our maximum likelihood estimator $\hat{p}$:

$$\ell'(p) = \frac{n\bar{y}}{p} - \frac{n(1-\bar{y})}{1-p} = 0$$

$$\hat{p} = \bar{y}$$

With the specific example above, this gives $\hat{p} = \bar{y} = \frac{4}{10}$. This example shows on a small scale what's happening for each of these steps: 1) the goal is to maximize the likelihood (which is a product), 2) we turn the product into a sum by taking the log, 3) we take the derivative of the log likelihood, and 4) set it equal to zero and solve for p. Sometimes we can't do all these steps and then we use things like gradient descent to approximate the steps of this process.

2.2 Maximum likelihood estimation for logistic regression

Turning back to our context of logistic regression, recall that our likelihood function is:

$$L(\vec{w}) = \prod_{i=1}^{n} h_{\vec{w}}(\vec{x}_i)^{y_i}(1 - h_{\vec{w}}(\vec{x}_i))^{1-y_i}$$

We will start by taking the negative log likelihood. We want to maximize our likelihood function, so when we take the negative log likelihood we want to *minimize* it. We'll call this negative log likelihood function J:

$$J(\vec{w}) = -\log L(\vec{w})$$
$$= -\sum_{i=1}^{n} [y_i \log h_{\vec{w}}(\vec{x}_i) + (1 - y_i)\log(1 - h_{\vec{w}}(\vec{x}_i))]$$

Consider what this cost function looks like for a single example:

$$J_{\vec{x}}(\vec{w}) = \begin{cases} -\log h_{\vec{w}}(\vec{x}) & \text{if } y = 1 \\ -\log(1 - h_{\vec{w}}(\vec{x})) & \text{if } y = 0 \end{cases}$$

If we consider the graph of this (see slides for figure) with $h(\vec{x})$ on the x-axis and $J(\vec{w})$ on the y-axis, the cost function starts at the origin and increases in an exponential curve wsith an asymptote at $x = 1$ when $y = 0$. When $y = 1$ it's the reverse with an asymptote at $x = 0$ and an intersection with the x-axis at $x = 1$. As we predict more and more incorrect things, it drives up the cost – just as we'd like!

Stochastic Gradient Descent. Now, we'll take the derivative of J, known as the *gradient* and denoted ∇J, with respect to one example $\vec{x}_i$. In the context of the stochastic gradient descent algorithm described earlier, this means we have the following steps:

```
shuffle the data
for i = 1...n
    $\vec{w} = \vec{w} - \alpha \nabla J_{\vec{x}_i}(\vec{w})$
```

Note that we'll update all weights at once based on one example (x_i). What remains is to determine ∇J. Recall that a single example x_i contributes $-\big(y_i \log h_{\vec{w}}(\vec{x}_i) + (1 - y_i)\log(1 - h_{\vec{w}}(\vec{x}_i))\big)$ to the overall sum of $J(\vec{w})$. So we have

$$\nabla J_{\vec{x}_i}(\vec{w}) = -\nabla_{\vec{w}}\Big(y_i \log h_{\vec{w}}(\vec{x}_i) + (1 - y_i)\log(1 - h_{\vec{w}}(\vec{x}_i))\Big)$$

$$= -\left(\frac{y_i}{h_{\vec{w}}(\vec{x}_i)} - \frac{1 - y_i}{1 - h_{\vec{w}}(\vec{x}_i)}\right)\nabla h_{\vec{w}}(\vec{x}_i)$$

In order to complete this calculation we'll need to determine what $\nabla h_{\vec{w}}(\vec{x}_i)$ is. Recall that the logistic function $g(z) = \frac{1}{1+e^{-z}}$. The derivative $g'(z) = g(z)(1 - g(z))$ (this calculation is left as an exercise to the reader). Note that this derivative requires no knowledge other than the function value! This is useful in practice. Using the chain rule and substituting this derivative above gives us:

$$= -\Big(\frac{y_i}{h_{\vec{w}}(\vec{x}_i)} - \frac{1 - y_i}{1 - h_{\vec{w}}(\vec{x}_i)}\Big)\ h_{\vec{w}}(x_i)\big(1 - h_{\vec{w}}(\vec{x}_i)\big)\vec{x}_i$$

After simplifying this, you get:

$$= \Big(-y_i + y_i h_{\vec{w}}(\vec{x}_i) + h_{\vec{w}}(\vec{x}_i) - y_i h_{\vec{w}}(\vec{x}_i)\Big)\vec{x}_i$$

$$= \left(\underset{\text{Predicted label}}{h_{\vec{w}}(\vec{x}_i)} - \overset{\text{True label}}{y_i} \right) \vec{x}_i$$

This is the difference between the true value and the prediction scaled by the specific data point. This is the $\nabla_{\vec{w}} J_{\vec{x}_i}(\vec{w})$ which we can plug into the algorithm weight update. We can think about this as taking one example out of the cost function to consider the cost function with respect to a single example, while the gradient is taken with respect to all of the weights. This is a good thing to remember for implementation; all the weights are updated at the same time. It doesn't make sense in terms of the stability of the model to update one weight at a time, but it's ok to look at one data point at a time.

With the stochastic gradient descent calculation in place, we have almost all the pieces we need to implement classification with logistic regression. The key pieces for stochastic gradient descent are the hypothesis function (the prediction based on the logistic function), the cost function we want to minimize (the negative log likelihood $J(\vec{w})$), and the calculation of the gradient of that cost function ($\nabla J(\vec{w})$ with respect to a single example x_i. Finally, we apply a linear threshold (similar to that for the Naïve Bayes model) to the final output to determine the classification, assumed by default to be 0.5 such that values greater than or equal to 0.5 receive a predicted label of 1 and otherwise the predicted label is 0.

2.3 Multi-class classification with logistic regression

Often, we're considering a classification of categorical items that are non-binary, such as political parties, blood groups, etc. Multi-class logistic regression can help us do this. When we only have two classes, we can always take $1 - p$ to determine the probability of the other case — in multi-class classification we can't do that anymore. We use *softmax* to handle this.

Consider the two-class case where

$$h(\vec{x}) = \frac{1}{1 + e^{-\vec{w}\cdot\vec{x}}} = \frac{e^{\vec{w}\cdot\vec{x}}}{e^{\vec{w}\cdot\vec{x}} + 1}$$

We can think of the first part of the bottom term as the weight on class 1 and the second part as the weight on class 0. We're going to generalize this to K classes. Let $\hat{\vec{y}}$ be my prediction vector:

$$\hat{\vec{y}} = h_{\vec{w}}(\vec{x}) = \begin{bmatrix} p(y = 1 \mid \vec{x}) \\ p(y = 2 \mid \vec{x}) \\ \cdot \\ \cdot \\ \cdot \\ p(y = K|\vec{x}) \end{bmatrix}$$

We can transform this to get our *softmax* version of this prediction vector:

$$= \underset{\text{Normalization term}}{\frac{1}{\sum_{k=1}^{K} e^{\vec{w}^{(k)}\cdot\vec{x}}}} \begin{bmatrix} e^{\vec{w}^{(1)}\cdot\vec{x}} \\ e^{\vec{w}^{(2)}\cdot\vec{x}} \\ \cdot \\ e^{\overset{s_i}{\vec{w}^{(i)}\cdot\vec{x}}} \\ \cdot \\ e^{\vec{w}^{(k)}\cdot\vec{x}} \end{bmatrix}$$

We refer to the right part of this as the score vector and each item's exponent is denoted s_i. By raising e to these s_i values, we'll end up with positive values only, which will help us when we try to sum these to 1. We add a normalization term so that these values sum to 1. (This softmax function is often used as the last layer of a neural network.)

We can now consider a "one-hot" encoding of these vectors by having a 1 in the vector for the specific class value and 0s in all the other places, for example:

$$\vec{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ . \\ . \\ . \\ 1 \\ . \\ . \\ . \\ 0 \end{bmatrix}$$

Note that this is also a proper probability distribution that sums to 1.

Computing the cross entropy of the true values with the predictions (we can think of the binary logistic regression cost function as this) we get:

$$y_i \log h_{\vec{w}}(\vec{x}_i) + (1 - y_i) \log(1 - h_{\vec{w}}(\vec{x}_i))$$

Generalizing this cost function to K classes, we again get cross entropy:

$$J(\vec{w}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log p(y_i = k | \vec{x}_i)$$

One of these terms will be "on" for every example and the other terms will be "off." This incentivizes the probability of the true class to be as high as possible.