CS 5785 – Applied Machine Learning – Lec. 4

Prof. Serge Belongie, Cornell Tech
Scribe: Mor Cohen, Edward W., Angel W., Chloe E., Harsha V., Yubin X.

September 12, 2018

1   The Newton-Raphson Method

Logistic regression models are usually fit by maximum likelihood. As described at the end of last lecture, we will make use of its derivatives to maximize the log likelihood $\ell(\beta)$. This sets the stage for this optimization problem and brings us to a classic problem: root-finding, and a classic algorithm: Newton's method (also known as the Newton–Raphson method, named after Isaac Newton and Joseph Raphson).

Fig. 1 is an example of Newton's method from Wikipedia. To find the root of a function (shown in blue), we start with an initial guess x_1 and fit a tangent line using its first derivative if it exists. The tangent line is an approximation of the function at that point. Then we get new point x_2, which is assumed to be the root. Because the function is complex and we can see it is off the correct root. Then we use x_2 as the next guess. We repeat the process until a sufficiently accurate value is reached. Note that this is only guaranteed to provide us a local optimum.

Fig. 2 shows an instance of a Newton-type methods that uses both the first and second derivatives for a 1D function to find a local maximum of function $f(x)$. We start with a guess at x_k. Instead of fitting a tangent line to the function to generate the next guess, we fit a local quadratic approximation, i.e., a function that has the same 1st and 2nd derivatives as the target function. We can easily solve for the maximum of this new function and obtain the next guess for the maximum, $x_k + d_k$. We repeat this process until convergence.

In our case we wish to solve the related problem of finding the maximum of a function. We will do this using both the first and second derivatives. In higher dimensions, as with our log likelihood $\ell(\beta)$ where $\beta \in \mathbb{R}^{p+1}$, the counterparts to the 1st and 2nd derivatives are the gradient and the Hessian, respectively.

$$\ell(\beta) = \sum_{i=1}^{N} \left\{ y_i \beta^\top x_i - log(1 + e^{\beta^\top x_i}) \right\} \tag{1}$$

The gradient of the log likelihood is given by

$$\frac{\partial \ell(\beta)}{\partial \beta} = \sum_{i=1}^{N} x_i (y_i - p(x_i; \beta)) \tag{2}$$

Note that the gradient is a length $p+1$ vector. We can see from this expression that the gradient can be expressed as a sum of feature vectors $x_i \in \mathbb{R}^{p+1}$,

(a) Start with a guess x_1, find the tangent line using the first derivative, extrapolate to get next guess x_2.

(b) Repeat this process to get x_3, which as we can see overshoots in this case.

(c) Keep iterating

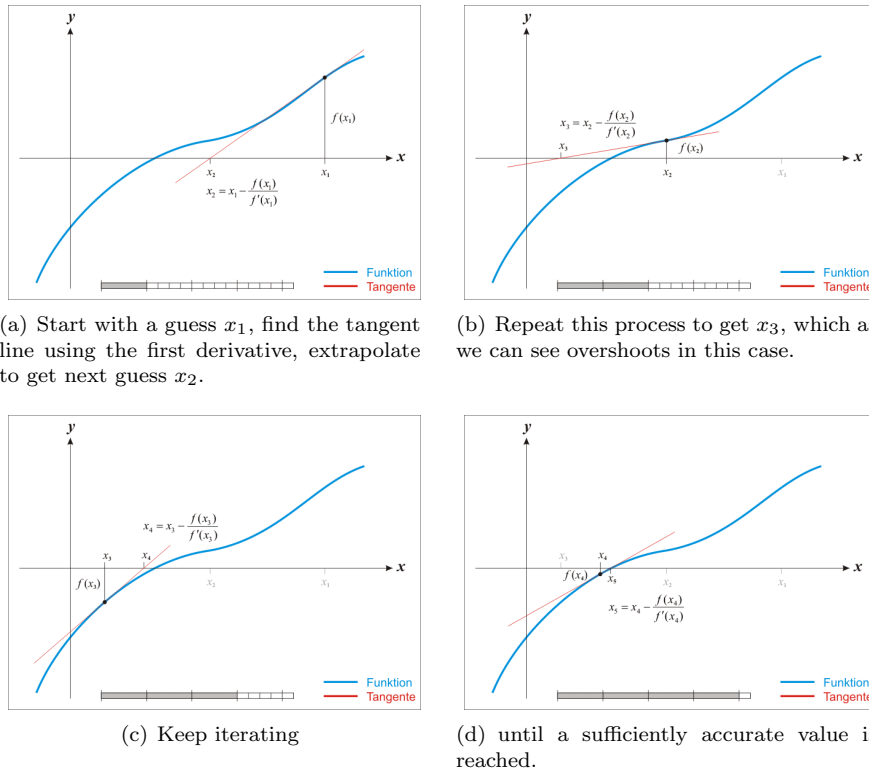(d) until a sufficiently accurate value is reached.
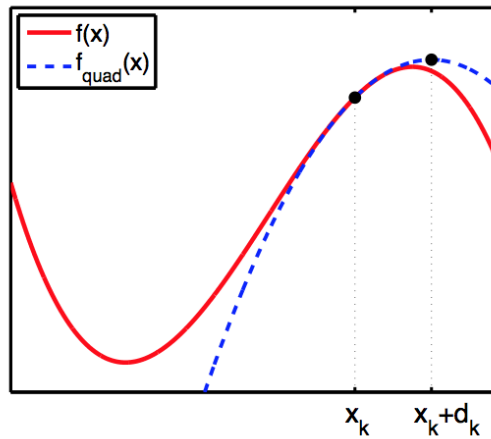
Figure 1: Illustration of Newton's method for finding a root. (Wikipedia)

Figure 2: Finding a local maximum of a function $f(x)$ using a quadratic approximation. We start with a guess at x_k. Using both the 1st and 2nd order derivatives we fit a quadratic approximation to the function at that point. That function $f_{\text{quad}}(x)$, is equivalent to the 2nd order Taylor series approximation of $f(x)$ around x_k. We can easily solve for the maximum of this function and obtain the next guess for the maximum, $x_k + d_k$. We repeat this process until convergence. (Murphy)
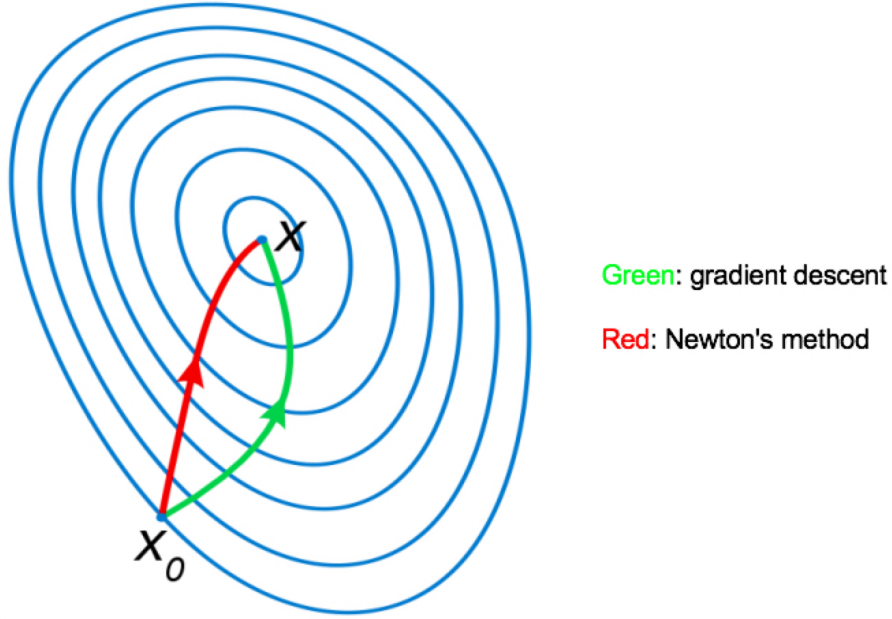
Figure 3: A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information to take a more direct route. (Wikipedia)

$i = 1, \ldots, N$, weighted by $y_i - p(x_i; \beta)$. We can interpret that weight as a signed error, since $p(x_i; \beta)$, the posterior probability, aims to approximate y_i.

The Hessian of $\ell(\beta)$, which is a $(p+1) \times (p+1)$ matrix, is given by

$$\frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^\top} = -\sum_{i=1}^{N} x_i x_i^\top p(x_i; \beta) \left(1 - p(x_i; \beta)\right)$$

This matrix is a sum of terms of the form $x_i x_i^\top$ weighted by $p(x_i; \beta)(1 - p(x_i; \beta))$ and it characterizes the curvature of the function we're trying to minimize. Each $x_i x_i^\top$ term is the *outer product* of x_i with itself, resulting in a symmetric matrix of size $(p+1) \times (p+1)$. We will see soon that this is closely related to how we compute the *covariance matrix* for $\{x_i\}_1^N$. The scalar weight $p(x_i; \beta)(1 - p(x_i; \beta))$ has a small value when $p(x_i; \beta)$ is close to y_i. If $p(x_i; \beta)$ is close to 1 then $1 - p(x_i; \beta)$ is close to 0 and vice versa, which means that if the approximation is good at least one multiplier in the expression $p(x_i; \beta)(1 - p(x_i; \beta))$ is close to 0. As a result, the entire expression has a very low value far from the decision boundary. If $p(x_i; \beta)$ is a bad approximation of y_i there is a higher chance that the classification is incorrect and as such $p(x_i; \beta)$ is closer to 0.5, $1 - p(x_i; \beta)$ is also close to 0.5 making the entire expression close to its maximum: 0.25. If we look back at the Hessian we can now see that the sum emphasizes x_is for which we are less certain of their classification making the contribution to the Hessian larger when the choice of β results in a poor decision boundary.

Fig. 3 is a comparison of gradient descent (green) and Newton's method (red) for minimizing a function with small step sizes (the name of method varies de-

3

pending on the reference) but the green curve is a classic first-order method and the red curve uses first- and second-order information. The first-order tells you something about the slope, and it will eventually get you to the local optimum by taking step in the direction of gradient. But if you take the curvature into account, which is indicated by the second-order derivative, you can get a more direct route.

1.1 Iterative Reweighted Least Squares (IRLS)

In IRLS we start by picking a random choice for β and apply a "Newton update" to get a better β. We continue taking Newton steps until we reach convergence. A Newton update is:

$$\beta^{new} = \beta^{old} - \left(\frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^\top}\right)^{-1} \frac{\partial \ell(\beta)}{\partial \beta}$$

in which the derivatives are evaluated at $\beta = \beta^{old}$. If we put this in matrix notation we get:

$$\frac{\partial \ell(\beta)}{\partial \beta} = \mathbf{X}^\top(\mathbf{y} - \mathbf{p})$$

$$\frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^\top} = -\mathbf{X}^\top \mathbf{W} \mathbf{X}$$

Where:

- $\mathbf{y}$ is a vector of all the y_is.

- $\mathbf{X} \in \mathbb{R}^{N \times (p+1)}$ is the matrix of all the feature vectors x_is.

- $\mathbf{p}$ is a vector of the $p(x_i; \beta)$s.

- $\mathbf{W} \in \mathbb{R}^{N \times N}$ is a diagonal weighting matrix containing the values of $p(x_i; \beta)(1 - p(x_i; \beta))$ on the diagonal.

When we write the Newton update using this notation we get:

$$\beta^{new} = \beta + (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top(\mathbf{y} - \mathbf{p})$$

$$\Downarrow$$

(Multiply β by $(\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1}(\mathbf{X}^\top \mathbf{W} \mathbf{X})$ which is the same as not changing it)

$$\beta^{new} = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1}(\mathbf{X}^\top \mathbf{W} \mathbf{X})\beta + (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top(\mathbf{y} - \mathbf{p})$$

$$\Downarrow$$

(Factor out $(\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W}$)

$$\beta^{new} = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W}(\mathbf{X}\beta + \mathbf{W}^{-1}(\mathbf{y} - \mathbf{p}))$$

$$\Downarrow$$

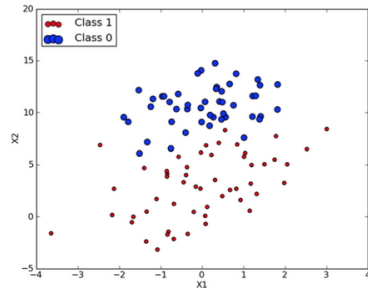$$\beta^{new} = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W} \mathbf{z}$$
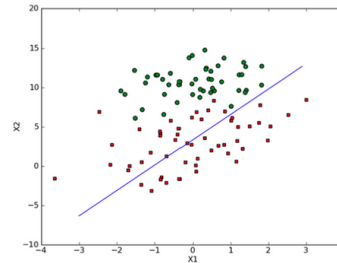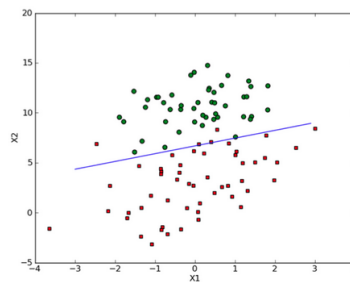
Figure 4: Logistic regression example (Harrington)

in which $\mathbf{z} = \mathbf{X}\beta + \mathbf{W}^{-1}(\mathbf{y} - \mathbf{p})$. We call $\mathbf{z}$ the *adjusted response.* In every IRLS iteration we solve this equation with a new set of $\mathbf{p}$, $\mathbf{W}$ and $\mathbf{z}$. This iteration solves the following weighted least squares problem over and over:

$$\beta^{new} \leftarrow \arg\min_{\beta} (\mathbf{z} - \mathbf{X}\beta)^{\top} \mathbf{W} (\mathbf{z} - \mathbf{X}\beta)$$

(Recall W is diagonal.) We iterate this step until convergence. This is named *Iteratively Reweighted Least Squares* since in every iteration it solves a weighted least squares problem.

One remaining question is, how do we initialize β? Starting with $\beta = 0$ is often OK. Another option is to use multiple randomized restarts to reduce the chances of getting stuck at a local maximum.

- Pros: It's fast

- Cons: For each iteration need to invert a large matrix.

- Cons: Need to include all observations in this matrix.

2 Online Learning

2.1 Stochastic Gradient Descent

IRLS is called a "batch" method since it has all the necessary data in advance, but that is not always possible or available. For example, the data could be
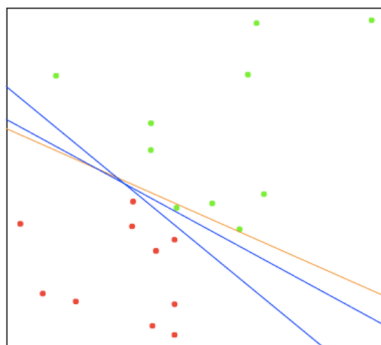
FIGURE 4.14. *A toy example with two classes separable by a hyperplane. The orange line is the least squares solution, which misclassifies one of the training points. Also shown are two blue separating hyperplanes found by the* perceptron learning algorithm *with different random starts.*

too big to fit into memory. Online learning updates β every step based on individual streamed instances, as opposed to IRLS that uses all the data in every step. *Stochastic Gradient Descent* (SGD) is an example of an online learning or streaming method that uses the following update step with *step size* or *learning rate* α:

$$\beta^{new} \leftarrow \beta^{old} - \alpha x_i(y_i - p(x_i; \beta))$$

Compared to the exact expression for the gradient in Eqn. (2), SGD updates β per data point rather than summing the gradient contributions over the entire data set. If the data point was classified correctly the boundary doesn't change, if not we use the update rule above to update it.

Fig. 4 is an example of SGD. We have two-class problem. The line presents the decision boundary. Every time it picks up a sample and checks that it is classified correctly or not. If correctly, the response is to do nothing. If incorrectly, it changes the boundary slightly to increase the likelihood. At the end, even the decision boundary is close to the example, it is considered as done.

How do we choose the step size α? This is left as an exercise for the reader.

Perceptron Algorithm

> Technically that's perceptron learning; in this expression the update is proportional (including sign) to the discrepancy between the ground truth and the estimated posterior probability.

This algorithm is closely related to Rosenblatt's *Perceptron Learning Algorithm* (HTF §4.5.1), which is hard thresholded and doesn't maintain probability estimates. See an example here: `http://www.youtube.com/watch?v=vGwemZhPlsA` ("Perceptron Learning Rule"). The LMS algorithm of Widrow and Hoff is also an example of SGD. SGD are also used in training neural networks.

2.2 Thoughts on Separating Hyperplanes

As shown in HTF Fig. 4.14, given two classes that are linearly separable, there exists an infinite family of separating lines/planes/hyperplanes between the classes. The figure shows solutions found via perceptron learning with different initializations. Which solution is best? Our intuition suggests we should choose the line with the furthest perpendicular distance from the closest data points from each class, since this might prevent overfitting. How do we translate that into an algorithm? We will tackle this question later in the semester when we discuss *support vector machines*.

2.3 Kernels: a Preview

Note that so far we have just been using the input variables X in their raw form (apart from adding the 1 to the front). A whole new frontier of algorithms awaits us if we supplement x by squares and cross products of its variables like $x_1^2, x_2^2, x_1 x_2$, etc. This hints at the idea of a *kernel*, a mapping through which we can augment the representation by adding dimensionality. This is particularly useful when dealing with points that are not linearly separable, as in the XOR problem. We will discuss kernels later in the semester.

3 Selected Q&A

3.1 When do you use the first order approximation and when do you use the second order approximation?

In practice, in very large big data machine learning contexts, people overwhelmingly use the first order approximation. Even though we know that the second order information is very very helpful and it infers the design of the algorithm, it has certain practical drawback that makes it harder to deploy. Every iteration, you have to invert a large matrix and it is not really make things fast.

3.2 How do we know if we are stuck in local optimization and how to avoid it?

You never know and could not solve it yet. It is a big problem in the filed. Now you can only have gigantic clusters doing random restarts and get the best local optima.