Gradient Descent

Ryan Miller

Grinnell College
Statistics

Introduction

We've seen a variety of modeling algorithms that can be divided into two paradigms:

1. Data-dependent models, such as decision trees or KNN, where the model structure is directly shaped by the training data without any fixed functional form
2. Parametric models, such as linear/logistic regression, where the model has a pre-defined functional form, and the parameters of that form (weights) are learned from the data

Data-dependent models typically rely upon their own specialized optimization algorithms, but parametric models can be optimized by **gradient descent**.

Cost Functions

- When we talk about a model "learning" from the training data, what we really mean is that it alters its parameters to achieve a lower error rate, which is measured by a **cost function**
  - Cost functions quantify how well a model's predictions align with actual outcomes
- The two cost functions we'll focus on are:
  - **Squared-error**
    - Cost $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
    - We'll use this for numeric outcomes
  - **Cross-entropy** (log-loss)
    - Cost $= -\sum_{i=1}^{n}\left(y_i \cdot log(\hat{y}_i) + (1-y_i)\cdot log(1-\hat{y}_i)\right)$
    - We'll use this for categorical outcomes

Linear Regression and Squared-error

Recall that linear regression represents the outcome as a linear combination of predictive features:

$$Y = w_o + w_1 X_1 + w_2 X_2 + \ldots + w_p X_p + error$$

This model can be expressed using matrix notation:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{error}$$

For a given set of weight estimates, $\widehat{\mathbf{w}}$, we can define the cost:

$$Cost = \frac{1}{n}(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$$

The cost quantifies the effectiveness of $\widehat{\mathbf{w}}$

Linear Regression and Squared-error

As you might be expecting, we generally want the *best possible* set of weights, or the set that minimizes the cost function. We can use calculus to perform this minimization, but first let's rearrange a few terms:

$$\begin{aligned} Cost &= \frac{1}{n}(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) \\ &= \frac{1}{n}\mathbf{y}^T\mathbf{y} + \frac{1}{n}(-2\mathbf{y}^T\mathbf{X}\hat{\mathbf{w}} + (\mathbf{X}\hat{\mathbf{w}})^T\mathbf{X}\hat{\mathbf{w}}) \\ &= \frac{1}{n}\mathbf{y}^T\mathbf{y} + \frac{-2}{n}\mathbf{y}^T\mathbf{X}\hat{\mathbf{w}} + \frac{1}{n}\hat{\mathbf{w}}^T\mathbf{X}^T\mathbf{X}\hat{\mathbf{w}} \end{aligned}$$

For linear regression, we can differentiate with respect to $\widehat{\mathbf{w}}$, set the resulting expression equal to zero, then solve for a closed form solution of: $\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

Gradient Descent

- The derivative is the slope of a function at a particular location, so we can use the gradient to gradually move towards the minimum of any (convex) cost function
  - The **gradient descent algorithm** works to minimize the cost function using sequential updates:

$$\hat{\mathbf{w}}^{(j)} = \hat{\mathbf{w}}^{(j-1)} - \alpha \frac{\partial \text{Cost}}{\partial \mathbf{w}}\left(\hat{\mathbf{w}}^{(j-1)}\right)$$

- α is a tuning parameter that controls the *learning rate*, or how quickly to update the weight vector at each iteration
  - A small α requires many iterations for the algorithm to converge (reach the minimum)
  - A large α can overshoot the minimum, which can also cause convergence issues

Gradient Descent for Linear Regression

Recall our rearranged cost function is:

$$Cost = \frac{1}{n}\mathbf{y}^T\mathbf{y} + \frac{1}{n}(-2\mathbf{y}^T\mathbf{X}\hat{\mathbf{w}} + \hat{\mathbf{w}}^T\mathbf{X}^T\mathbf{X}\hat{\mathbf{w}})$$

Thus, after applying some matrix calculus shortcuts the gradient is:

$$Gradient = \frac{-2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$$

So, our gradient descent update scheme is:

$$\hat{\mathbf{w}}^{(j)} = \hat{\mathbf{w}}^{(j-1)} - \alpha \cdot \left(\frac{-2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}^{(j-1)})\right)$$

Simple Example

As an illustration, consider a very simple special case of linear regression involving no bias term (intercept) and a single weight parameter where data are generated via:

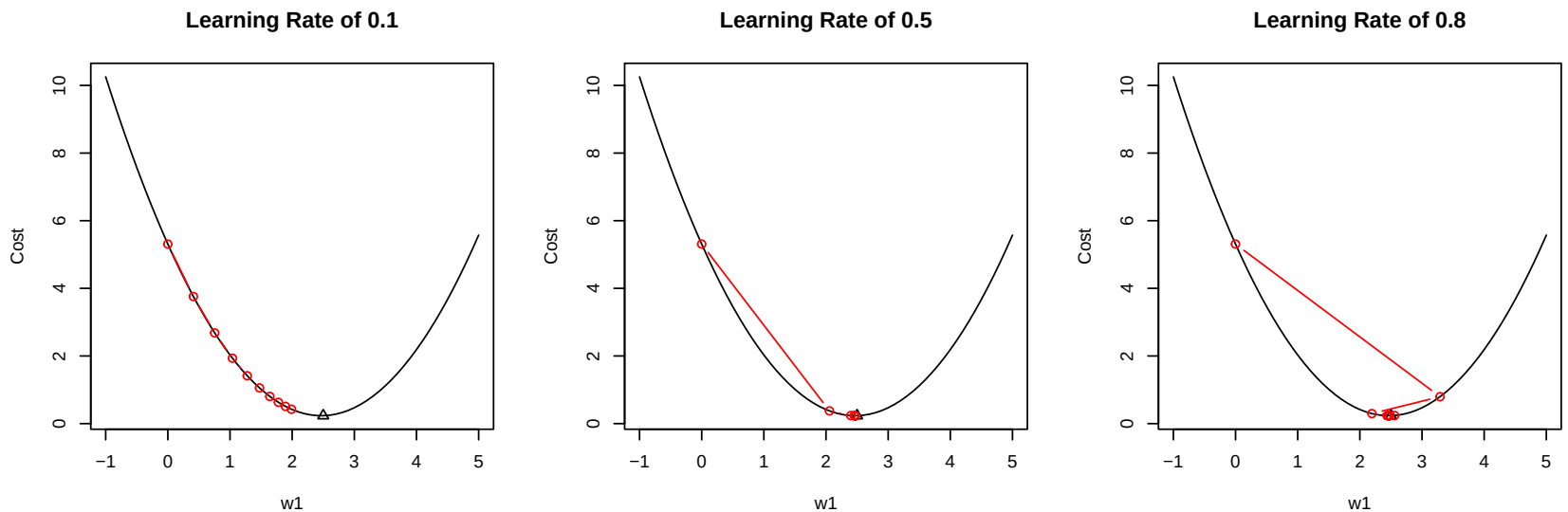
$$Y = 2.5X_1 + error$$

For this model, the squared error cost function is:

$$Cost = \frac{1}{n}(\mathbf{y} - \mathbf{x}_1^T \hat{w}_1)^T(\mathbf{y} - \mathbf{x}_1^T \hat{w}_1)$$

Simple Example (cont.)

The graphs below illustrate 10 iterations of gradient descent for our simple example (starting at $w_1^{(0)} = 0$):



Gradient descent typically stops when the next update uses weight changes that are below a small, predefined threshold.

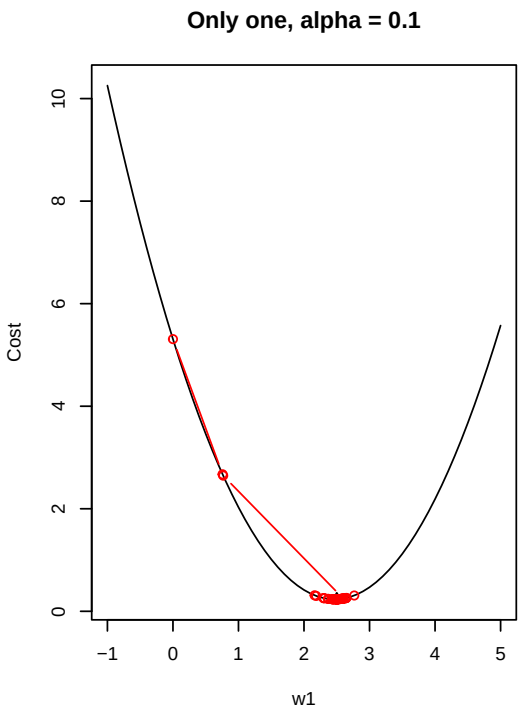
Stochastic Gradient Descent

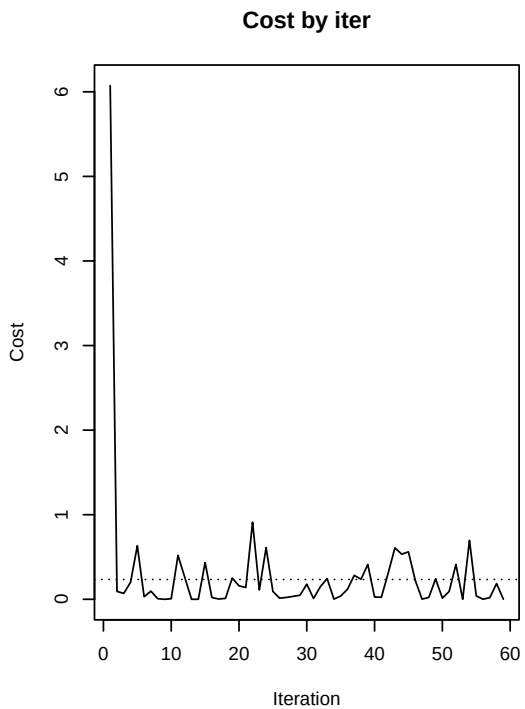
- In our simple example, computing the gradient at each iteration required two vector-product calculations: $\mathbf{y}^T\mathbf{x}_1$ and $\mathbf{x}_1^T\mathbf{x}_1$
  - Fortunately, these can both be computed ahead of time (rather than at each iteration) which makes algorithm very computationally efficient
- For other models, the parameter vector is involved in vector-product calculations within the gradient, so these vector-product calculations must be redone at each iteration
  - In big-data settings, this computational challenge (among others) has led to the popularity of **stochastic gradient descent**

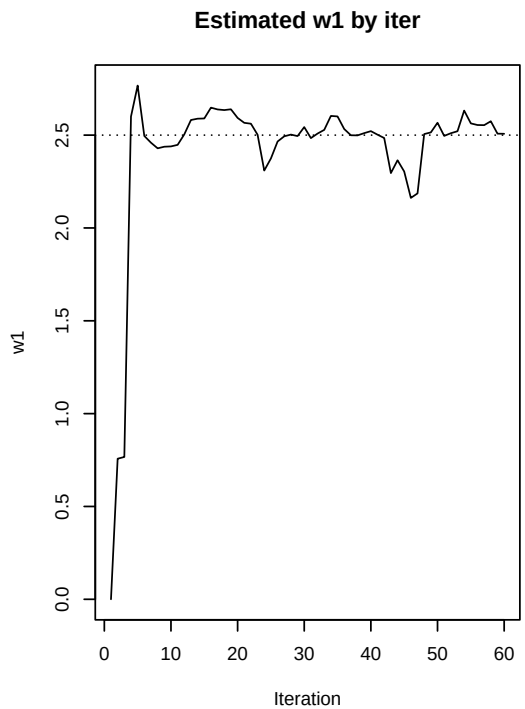
Stochastic Gradient Descent

- In Stochastic Gradient Descent (SGD) the gradient is computed and the model parameters are updated using a single training sample at a time
  - Samples are used in a random order, and a complete pass through all of the training data is called an **epoch**
- The inherent randomness introduced in SGD leads to noisy convergence that tends to move stochastically about the minimum of the cost function (hence the name)
  - However, this randomness has added benefit of allowing the optimization to avoid getting trapped at local minima

Stochastic Gradient Descent



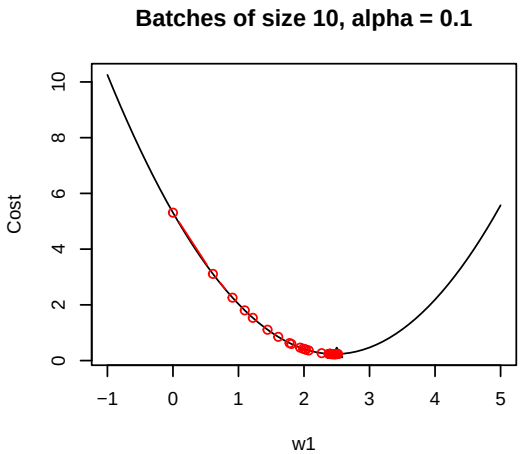


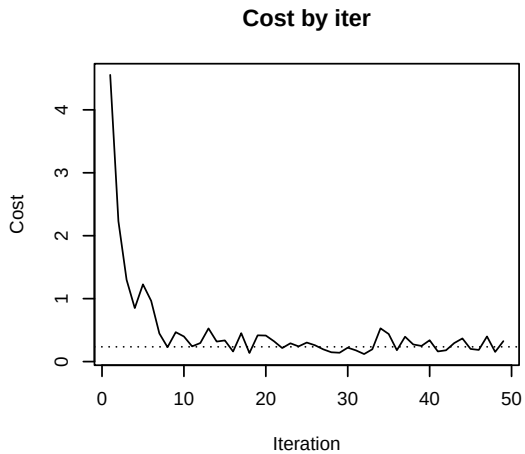


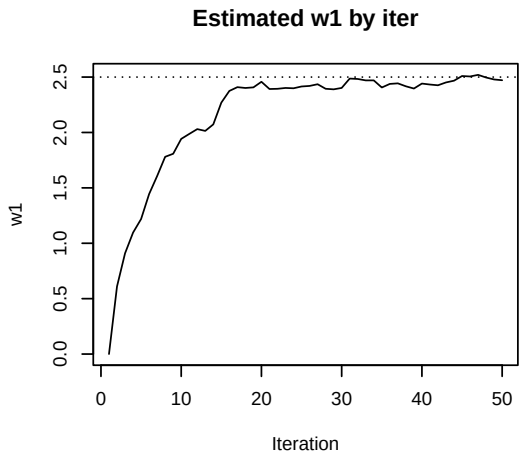
Mini-Batch Gradient Descent

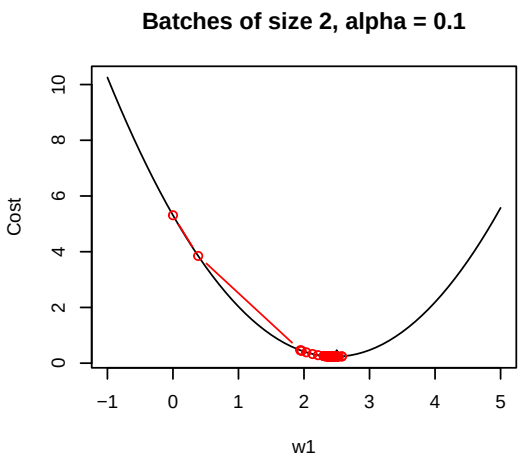
- **Mini-batch gradient descent** divides the data into batches and performs each update using a batch of data rather than the entire thing (or only one observation)
  - This is less noisy and more easily parallelizable than stochastic gradient descent, while still introducing enough randomness to help overcome local minima

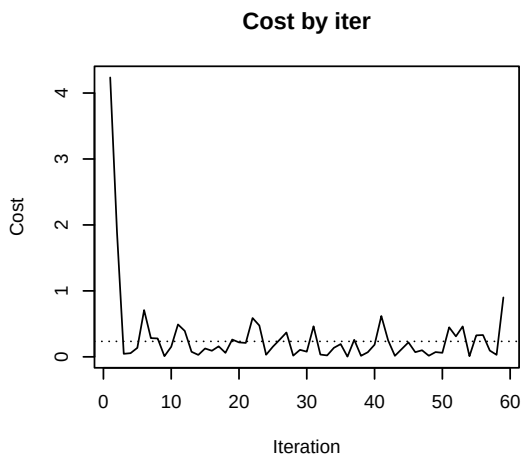
Mini-Batch Gradient Descent

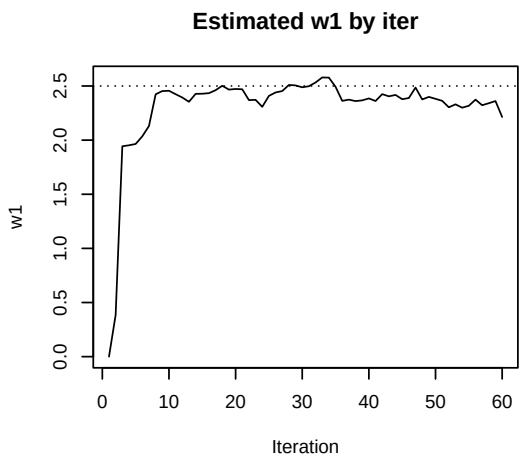












Gradient Descent Variations

Method	**Each Update**	**Convergence**
Gradient Descent	All data	Stable and slow
SGD	One obs.	Very noisy and fast
Mini-batch	Small subset	Moderately noisy and fast

Key Takeaways

- Machine learning models learn by adjusting weights to minimize a *cost function*.
- The gradient points toward the direction of greatest increase in the cost function, so moving in the opposite direction decreases the cost.
- *Gradient descent* repeatedly updates the weights using the gradient until further improvements become negligible.
- Modern machine learning models are often trained using *stochastic* or *mini-batch gradient descent*, which trade a noisier gradient estimate for much greater computational efficiency.