Lecture 24: MATH 342W: Introduction to Data Science and Machine Learning

Sergio E. Garcia Tapia*

May 9th, 2025 (last updated May 13, 2025)

1 Bagging

Remember that in bootstrapped aggregation (bagging), we started with a data set $\mathbb{D}$ and constructed M bootstrap samples $\mathbb{D}_1, \ldots, \mathbb{D}_M$. Then we used the M samples to construct M models $g_1, \ldots, g_M$ in parallel. Finally, we shipped as our final model their average:

$$g_{\text{bag}} := \frac{1}{M} \sum_{m=1}^{M} g_m$$

The g_m's have *low bias* (low mispecification error) and *high variance* (high estimation error) because they were chosen to overfit. They are called **strong learners**. For example, if using trees with $N_0 = 1$, they are called **deep trees**. Bootstrapping the $i = 1, \ldots, M$ data sets minimizes the covariance terms $\text{Cov}[g_i, g_j]$, which is the largest component of $\text{MSE}[g_{\text{bag}}]$. In summary, we begin with *low bias* (strong learners), and as M increases, the variance decreases.

2 Boosting

There are some similarities and differences between bagging and *boosting*. In the former, we compute an *average*; in the latter, we sum:

$$g_{\text{boost}} := \sum_{m=1}^{M} g_m$$

In bagging, the g_m's are low bias and high variance; in *boosting*, the g_m's have *high bias* and *low variance* (less accurate, less fit, more sure of our features). Thus when boosting, we say that the g_m's are **weak learners**, and instead we use *shallow trees* (N_0 is large), which are nevertheless flexible enough to make the sum complex. Moreover, the covariance terms $\text{Cov}[g_i, g_j]$ are small since the g_i are weak in different ways, so the variance term $\text{Var}[g_{\text{boost}}]$ becomes small.

*Based on lectures of Dr. Adam Kapelner at Queens College. See also the course GitHub page.

Example. In the Boston housing data set, one model may split only on the number of rooms, whereas another model may split only on the distance to Boston city center.

Therefore, we begin with *low variance* and as we increase M, the bias decreases.

3 Gradient Descent

There are many variations of boosting, but we will study **gradient boosting** (2004). To build this up, we will revisit calculus.

Gradient Descent for Function of One Variable

Consider a function

$$f(x) = 2 + (x-3)^2$$

We seek to minimize it. Let's pretend we know about derivatives, but we do not how to use them to find local minima. The derivative of f is $f'(x) = 2(x-3)$. We will use an iterative procedure to approximate $x_* = 3$, the point where f attains its local minimum value.

We begin with an input, say $x_0 = 1$, and evaluate f' at that point to get

$$f'(x_0) = 2(x_0 - 3) = -4$$

Draw a tangent line to the graph of f at $(x_0, f(x_0)) = (1, 6)$. We happen to know that x_* is to the right of x_0. We can move to the right by going in the direction of $-f'(x_0)$ (since that quantity will be positive), along the tangent line of f at $(1, 6)$, but we want to be careful not to overshoot by too much. To control how much we move by, define a hyparameter η called the **step size** or the **learning rate**. We will let $\eta = 0.1$, and we will move by $-\eta f'(x_0)$. Thus we obtain a new input x_1:

$$x_1 := x_0 + (-\eta f'(x_0)) = 1 + (-(0.1) \cdot (-4)) = 1.4$$

If we repeat this procedure, this time starting at x_1, we can obtain another approximation to x_* by using the tangent line to f at x_1:

$$x_2 := x_1 + (-\eta f'(x_1)) = 1.4 + (-(0.1)(-3.2)) = 1.72$$

See Figure 1. The idea is that if we continue this way, we eventually arrive at x_* (assuming certain conditions on the function and the starting point). This is called **gradient descent**.

Gradient Descent for Function of Multiple Variables

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be a scalar-valued function defined by

$$f(\boldsymbol{x}) = 2 + (x_1 - 3)^2 + (x_2 - 2)^4$$

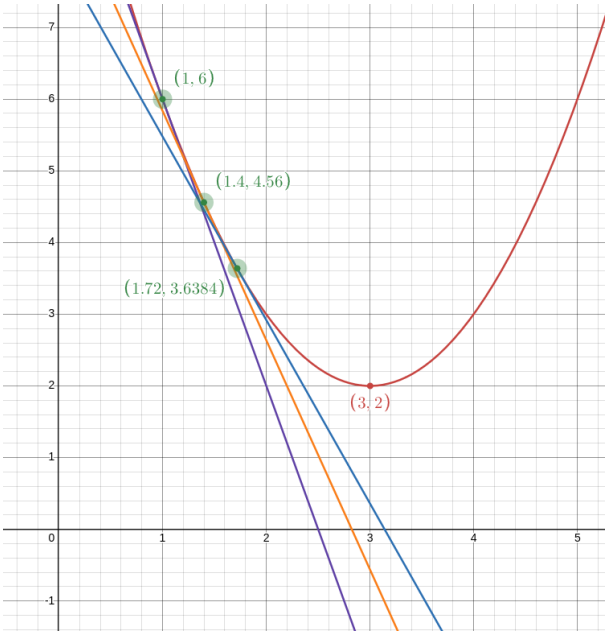

Figure 1: Applying gradient descent to approximate the local minimum of $f(x) = 2 + (x-3)^2$, starting at $x_0 = 1$.

where $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\top$. We want to find the minimum at $\boldsymbol{x}_* = (3, 2)$. Analogous to the previous example, we compute the derivative, which in this case is the **gradient**:

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2(x_1 - 3) \\ 4(x_2 - 2)^3 \end{bmatrix}$$

Let's use $\boldsymbol{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\top$ as the starting point, and let $\eta = 0.1$. Then similar to before, we can approximate $\boldsymbol{x}_*$ by moving in the direction of $-\nabla f(\boldsymbol{x}_0)$. This works because the gradient of a scalar-valued function gives the direction of fastest rate of increase of a function. Thus, by moving in the direction of $-\nabla f(\boldsymbol{x}_0)$, we move in the direction of greatest decrease. We compute a first approximation to $\boldsymbol{x}_*$ as follows:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + (-\eta \cdot \nabla f(\boldsymbol{x}_0)) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \cdot \begin{bmatrix} -4 \\ -4 \end{bmatrix} = \begin{bmatrix} 1.4 \\ 1.4 \end{bmatrix}$$

$$\vdots = \vdots$$

4 Gradient Boosting

The **gradient boosting algorithm** is based on the idea of gradient descent. The reason it took so long to arrive at this algorithm is because it is more complicated. Instead of starting with a point, we start with a *function*; each iteration we get not a new point but a new *function*. We need two things to make this work:

(**1**) A **loss function (error metric, objective function)**. We have seen many such functions throughout our study. Call this $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

(**2**) Fit a model to the negative gradient of $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$, that is to $-\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}})$, as the response.

After many iterations, we see the loss decreases. Let's compare the notations for gradient boosting and gradient descent side-by-side:

Gradient Descent	**Gradient Boosting**
Begin at x_0.	Begin at $G_0 := g_0$, the default (i.e., a the null model).
Direction to move: $-f'(x_0)$.	Move in the direction that improves loss. That is, run the algorithm on data X, but fit on loss of $\boldsymbol{y}$: $$\mathcal{A}(\langle X, -\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}_0)\rangle, \mathcal{H})$$
Move by amount $-\eta f'(x_0)$.	Move by amount $$g_1 := \eta\mathcal{A}(\langle X, -\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}_0)\rangle, \mathcal{H})$$
Compute a better approximation $$x_1 := x_0 - \eta f'(x_0)$$	Compute a better approximation $$G_1 := g_0 + g_1$$
Compute a second approximation $$x_2 := x_1 - \eta f'(x_1)$$	Compute a second approximation $$G_2 := \underbrace{g_0 + g_1}_{G_1} + \eta\mathcal{A}(\langle X, -\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}_1)\rangle, \mathcal{H})$$
Compute mth approximation $$x_m := x_{m-1} - \eta f'(x_{m-1})$$	Compute mth approximation $$G_m := \underbrace{g_0 + g_1 + \cdots + g_{m-1}}_{G_{m-1}} + \eta\mathcal{A}(\langle X, -\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}_{m-1})\rangle, \mathcal{H})$$

After M steps, we set $g_{\text{boost}} = G_M$. The algorithm is the *weak learner* that we talked about. Each term of G_M gets better each time. In order to employ this algorithm, we need to know the loss function that we are trying to fit.

Numeric Response Space, $\mathcal{Y} = \mathbb{R}$

Suppose we are predicting in a context where the response space is $\mathcal{Y} = \mathbb{R}$. For the loss function, we can use

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = SSE := \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We compute the gradient by treating L as a function of $\hat{\boldsymbol{y}}$ (the output of the unknown prediction function), and we negate it:

$$-\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\begin{bmatrix} \frac{\partial L}{\partial \hat{y}_1} \\ \vdots \\ \frac{\partial L}{\partial \hat{y}_n} \end{bmatrix} = -\begin{bmatrix} -(y_1 - \hat{y}_1) \\ \vdots \\ -(y_n - \hat{y}_n) \end{bmatrix} = 2(\boldsymbol{y} - \hat{\boldsymbol{y}}) = 2\boldsymbol{e}$$

where $\boldsymbol{e}$ is the residual vector. Thus, the algorithm begins by computing

$$g_1 := \eta \cdot \mathcal{A}(\langle X, 2\boldsymbol{e} \rangle, \mathcal{H})$$

Probability Estimation, $\mathcal{Y} = \{0, 1\}$

Suppose we are predicting in a context where the response is space is $\mathcal{Y} = \{0, 1\}$, and we intend to use probability estimation. Our modeling target is $P(Y = 1 \mid \boldsymbol{x})$. Assume we use the logistic link function, which is

$$\phi(u) = \frac{e^u}{1 + e^u}$$

We arrived at the following quantity for the probability of the data set $\mathbb{D}$ that we wished to maximize:

$$P(\boldsymbol{y}, \hat{\boldsymbol{p}}) = \prod_{i=1}^{n} \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1 - y_i}$$

where we have

$$\hat{p}_i = \phi(\hat{y}_i) = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} \iff \hat{y}_i = \ln\left(\frac{\hat{p}_i}{1 - \hat{p}_i}\right)$$

It turns out that it is easier to work with log-odds, so we will manipulate the equation for $P(\boldsymbol{y}, \hat{\boldsymbol{p}})$ to express it as a function of $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$:

$$\begin{aligned} P(\boldsymbol{y}, \hat{\boldsymbol{p}}) &= \prod_{i=1}^{n} \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1 - y_i} \\ &= \prod_{i=1}^{n} \left(\frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}}\right)^{y_i} \cdot \left(\frac{1}{1 + e^{\hat{y}_i}}\right)^{1 - y_i} && \text{(Substitute expression for } \hat{p}_i\text{)} \\ &= \prod_{i=1}^{n} \frac{e^{y_i \hat{y}_i}}{1 + e^{\hat{y}_i}} \\ &= P(\boldsymbol{y}, \hat{\boldsymbol{y}}) \end{aligned}$$

Now recall that since the logistic link function is monotonically increasing, maximizing probability is equivalent to maximizing log-probability. We define the loss function $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ as

$$\begin{aligned} L(\boldsymbol{y}, \hat{\boldsymbol{y}}) &= \ln(P(\boldsymbol{y}, \hat{\boldsymbol{y}})) \\ &= \ln\left(\prod_{i=1}^{n} \frac{e^{y_i \hat{y}_i}}{1 + e^{\hat{y}_i}}\right) \end{aligned}$$

$$= \sum_{i=1}^{n} \ln\left(\frac{e^{y_i \hat{y}_i}}{1 + e^{\hat{y}_i}}\right) \qquad \text{(Product property of logarithms)}$$

$$= \sum_{i=1}^{n} (y_i \hat{y}_i - \ln(1 + e^{\hat{y}_i})) \qquad \text{(Quotient property of logarithms)}$$

Then we compute the negative gradient as before:

$$-\nabla L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\begin{bmatrix} \frac{\partial L}{\partial \hat{y}_1} \\ \vdots \\ \frac{\partial L}{\partial \hat{y}_n} \end{bmatrix} = -\begin{bmatrix} y_1 - \frac{e^{\hat{y}_1}}{1+e^{\hat{y}_1}} \\ \vdots \\ y_n - \frac{e^{\hat{y}_n}}{1+e^{\hat{y}_n}} \end{bmatrix} = \boldsymbol{y} - \hat{\boldsymbol{p}} = \boldsymbol{e}$$

5 Types of Machine Learning

Machine learning can be applied in a variety of ways. The following is a non-exhaustive survey of them.

Machine Teaching

Begin with $\mathbb{D}$, and compute g. Then, inspect $\boldsymbol{e}$ to see which e_i's are large. Assume you can generate $\langle \boldsymbol{x}_*, y_* \rangle$, where $\boldsymbol{x}_* \approx \boldsymbol{x}_i$, and then retrain. The idea is to target specific places where residuals are poor.

Supervised Learning

Use $\mathbb{D} = \langle X, \boldsymbol{y} \rangle$ to create g for prediction. This was our focus this semester.

Unsupervised Learning

Use $\mathbb{D} = X$ to *understand* the covariate space. That is, we are not trying to *predict*; we are trying to *understand*. There are some variations on this idea:

- **Clustering**: Cluster units into groups $G_1, G_2, \ldots, G_K$, such that $G_1, \ldots, G_K$ form a partition of X (i.e., $\cup_{i=1}^{K} G_i = X$ and $G_i \cap G_j = \varnothing$ for $i \neq j$).
- **Anomaly detection**: Is a given $\boldsymbol{x}_*$ "strange", considering the historical data X?
- **Dimension reduction**: Can you explain or approximate with much less than p measurements? The main methods in this category are **principal component analysis (PCA)** and **factor analysis**.