MS&E 226: Fundamentals of Data Science

Lecture 6: An optimization view of learning algorithms

Ramesh Johari

Learning algorithms: An optimization view

Examples of learning algorithms

Recall that a *learning algorithm* is a procedure that takes as input training data $(\mathbf{X}^{\text{train}}, \mathbf{Y}^{\text{train}})$, and produces as output a fitted model $\hat{f}$.

Examples we've seen include OLS linear regression; ridge regression; and lasso.

Where do learning algorithms come from?

OLS as an optimization problem

Recall that OLS solves an optimization problem:

> *"Find the coefficient vector $\hat{\boldsymbol{\beta}}$ that minimizes the sum of squared errors (SSE) on the training data ($\mathbf{X}^{train}, \mathbf{Y}^{train}$)."*

Similarly for ridge regression and lasso. (What are the corresponding optimization problems?)

An optimization view

In general, for both regression and classification problems, many learning algorithms work as follows (in fact the vast majority!):

- Define a family of *models* in terms of a collection of *parameters* (e.g., for OLS, linear models with *coefficients* on the features)
- Define an objective to optimize (often written in terms of a *training loss* to minimize, e.g., sum of squared errors for OLS)
- Solve the resulting optimization problem for the optimal parameters (i.e., OLS coefficients)

In this lecture we'll discuss some other examples of this approach.

Maximum likelihood estimation

Overview

The basic idea behind this optimization approach:

1. *Distributional assumption*: “Pretend” that the data came from a probability distribution with a known structure, but unknown parameters.

Overview

The basic idea behind this optimization approach:

1. *Distributional assumption*: "Pretend" that the data came from a probability distribution with a known structure, but unknown parameters.
2. *Likelihood computation*: For each choice of parameters, compute the chance of seeing the training data, given a particular value of the parameters.

Overview

The basic idea behind this optimization approach:

1. *Distributional assumption*: "Pretend" that the data came from a probability distribution with a known structure, but unknown parameters.
2. *Likelihood computation*: For each choice of parameters, compute the chance of seeing the training data, given a particular value of the parameters.
3. *Optimization*: Pick the parameter values that maximize the likelihood.

Overview

The basic idea behind this optimization approach:

1. *Distributional assumption*: “Pretend” that the data came from a probability distribution with a known structure, but unknown parameters.
2. *Likelihood computation*: For each choice of parameters, compute the chance of seeing the training data, given a particular value of the parameters.
3. *Optimization*: Pick the parameter values that maximize the likelihood.

Learning algorithms that work this way are called *maximum likelihood estimators* (MLE).

OLS is a maximum likelihood estimator

It turns out that OLS is a maximum likelihood estimator.

Training data: n observations Y_i, $i = 1, \dots, n$, with associated p-dimensional covariate vectors $\mathbf{X}_i = (X_{i1}, \dots, X_{ip})$, $i = 1, \dots n$.

We'll work through each of the three steps in turn.

OLS step 1: Distributional assumption

We "pretend" (assume) the Y_i came from a *linear normal population model* with *i.i.d. errors*:

$$Y_i = \sum_{i=j}^{p} \beta_j X_{ij} + \epsilon_i,$$

where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, and the ϵ_i are i.i.d.

Note that we treat $\mathbf{X}$ as *given*; i.e., we focus on the distribution of $\mathbf{Y}$ given $\mathbf{X}$, rather also modeling $\mathbf{X}$ as random.

The *parameters* are $\beta_1, \ldots, \beta_p$, as well as σ^2.

OLS step 2: Likelihood computation

Likelihood is probability density (pdf) of seeing $\mathbf{Y}$, given parameters and $\mathbf{X}$:

$$f(\mathbf{Y}|\boldsymbol{\beta}, \sigma^2, \mathbf{X}) = \prod_{i=1}^{n} \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) \exp\left(-\frac{\left(Y_i - \sum_{j=1}^{p} \beta_j X_{ij}\right)^2}{2\sigma^2} \right).$$

(Recall that observations in the sample are assumed to be *independent* of each other.)

In general: Parametric likelihood

In general, suppose there is some process that generates $\mathbf{Y}$.

Suppose each choice of a parameter vector $\boldsymbol{\theta}$ gives rise to a conditional pmf (or pdf) $f(\mathbf{Y}|\boldsymbol{\theta})$.[1]

This is the probability (or density) of seeing $\mathbf{Y}$, given parameters $\boldsymbol{\theta}$.

We call this the *likelihood* of $\mathbf{Y}$ given $\boldsymbol{\theta}$.

[1] As noted on the previous slide, in the case of regression, we also treat $\mathbf{X}$ as given and look at $f(\mathbf{Y}|\boldsymbol{\theta}, \mathbf{X})$.

Log likelihood

It is often easier to work with *logarithm* of likelihood:

- Converts a *product* into a *sum*, which is convenient for optimization
- Multiplying many small probabilities together can cause numerical instability

We call this the *log likelihood function* (LLF).

LLF for linear normal population model):

$$\log f(\mathbf{Y}|\boldsymbol{\beta}, \sigma^2, \mathbf{X}) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(Y_i - \sum_{j=1}^{p}\beta_j X_{ij}\right)^2 .$$

Maximum likelihood estimation

The *maximum likelihood estimate* (MLE) is the parameter value that maximizes the likelihood.

It is found by solving the following optimization problem:

$$\begin{array}{rl} \text{maximize} & f(\mathbf{Y}|\boldsymbol{\theta}) \quad (\text{or } \log f(\mathbf{Y}|\boldsymbol{\theta})) \\ \text{over} & \text{feasible choices of } \boldsymbol{\theta}. \end{array}$$

OLS step 3: Maximizing likelihood (known σ^2)

Suppose that $\boldsymbol{\sigma}^2$ is known, so the optimization is only over coefficients $\boldsymbol{\beta}$.

Returning to LLF, our problem is equivalent to choosing $\hat{\boldsymbol{\beta}}$ to minimize:

$$\text{minimize} \quad \sum_{i=1}^{n} \left(Y_i - \sum_{j=1}^{p} \hat{\beta}_j X_{ij} \right)^2 .$$

OLS step 3: Maximizing likelihood (known σ^2)

Suppose that σ^2 is known, so the optimization is only over coefficients $\boldsymbol{\beta}$.

Returning to LLF, our problem is equivalent to choosing $\hat{\boldsymbol{\beta}}$ to minimize:

$$\text{minimize} \quad \sum_{i=1}^{n} \left(Y_i - \sum_{j=1}^{p} \hat{\beta}_j X_{ij} \right)^2 .$$

In other words: the OLS solution is *the MLE estimate of the coefficients!*

OLS step 3: Maximizing likelihood (unknown σ^2)

What happens if σ^2 is *unknown*?

The MLE for $\boldsymbol{\beta}$ remains unchanged (the OLS solution), and the MLE estimate for σ^2 is:

$$\hat{\sigma}^2_{\mathrm{MLE}} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \sum_{j=1}^{p}\hat{\beta}_j X_{ij}\right)^2 = \frac{1}{n}\sum_{i=1}^{n} r_i^2.$$

This is intuitive: the sum of squared residuals is an estimate of the variance of the error.

“Interpretability”

Maximum likelihood estimation is a powerful technique for building “interpretable” predictive models:

One interpretation of MLE is to view the parameters as determining a “reasonable” approximation to the true data-generating distribution.

The extent to which this view is “reasonable” depends on whether or not you believe the distributional assumptions of your particular MLE in your context.

E.g.: Do you in fact believe that the data you obtained came from a linear normal population model?

An MLE for classification:
Logistic regression

Logistic regression

At its core, logistic regression is a learning algorithm for *binary classification* that works as follows:

Input: Sample data $\mathbf{X}$ and $\mathbf{Y}$.

Output: A fitted model $\hat{f}(\cdot)$, where we interpret $\hat{f}(\vec{X})$ as *an estimate of the probability that the corresponding outcome Y is equal to 1.*

To convert this to a classifier, we choose a *threshold* t, and return 1 if $\hat{f}(\vec{X}) > t)$ (resp., return 0 if $\hat{f}(\vec{X} < t)$). (Ties can be broken in any way you like.)

Logistic regression step 1: Distributional assumption

Suppose we have training data $(\mathbf{X}, \mathbf{Y})$, with binary $Y_i \in \{0, 1\}$.

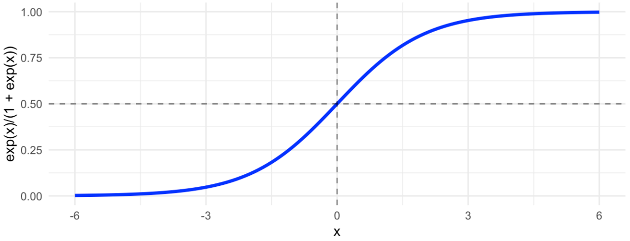
We "pretend" the Y_i came from a *logistic* population model with *coefficients* $\boldsymbol{\beta}$:

$$\mathbb{P}(Y = 1|\vec{X}) = \frac{\exp(\vec{X}\boldsymbol{\beta})}{1 + \exp(\vec{X}\boldsymbol{\beta})} = 1 - \mathbb{P}(Y = 0|\vec{X}).$$

Note that $\vec{X}\boldsymbol{\beta} = \sum_{j=1}^{p} \beta_j X_j$.

Logistic curve (Sigmoid function)

The function g given by $g(x) = e^z/(1 + e^z)$ is called the *logistic curve* or *sigmoid* function.



In terms of g, we can write the population model as:[2]

$$\mathbb{P}(Y = 1|\vec{X}) = g(\vec{X}\boldsymbol{\beta}).$$

[2]This is one example of a *generalized linear model* (GLM); for a GLM, g^{-1} is called the *link function*.

Logistic curve (Sigmoid function)

Note that from this curve we see some important characteristics of logistic regression:

- The logistic curve is *increasing*. Therefore, in logistic regression, larger values of covariates that have *positive* coefficients will tend to increase the probability that $Y = 1$.

Logistic curve (Sigmoid function)

Note that from this curve we see some important characteristics of logistic regression:

- The logistic curve is *increasing*. Therefore, in logistic regression, larger values of covariates that have *positive* coefficients will tend to increase the probability that $Y = 1$.
- When $z > 0$, then $g(z) > 1/2$; when $z < 0$, then $g(z) < 1/2$. Therefore, when $\vec{X}\boldsymbol{\beta} > 0$, Y is more likely to be one than zero; and conversely, when $\vec{X}\boldsymbol{\beta} < 0$, Y is more likely to be zero than one.

See the appendix for some motivations for the logistic population model.

Logistic regression step 2: Likelihood computation

The (conditional) likelihood for $\mathbf{Y}$ given $\mathbf{X}$ and parameters $\boldsymbol{\beta}$ is:

$$\mathbb{P}(\mathbf{Y}|\boldsymbol{\beta}, \mathbf{X}) = \prod_{i=1}^{n} g(\mathbf{X}_i\boldsymbol{\beta})^{Y_i}(1 - g(\mathbf{X}_i\boldsymbol{\beta}))^{1-Y_i}$$

where $g(z) = \exp(z)/(1 + \exp(z))$, and $\mathbf{X}_i$ is the i'th feature vector in the training sample.

The negation of the LLF for logistic regression is also referred to as *log loss* or *cross-entropy loss* (see problem set); maximizing likelihood is equivalent to minimizing log loss.

Logistic regression step 3: Maximum likelihood

Let $\hat{\boldsymbol{\beta}}_{\text{MLE}}$ be the resulting MLE solution; these are the *logistic regression coefficients*.

Unfortunately, in contrast to our previous examples, maximum likelihood estimation does not have a closed form solution in the case of logistic regression.

However, there are reasonably efficient iterative methods for algorithmically computing the MLE solution.[3]

One example is an algorithm inspired by weighted least squares, called *iteratively reweighted least squares*. This algorithm iteratively updates the weights in weighted least squares to converge to the logistic regression MLE solution.

[3] The MLE optimization problem turns out to be *convex*, which is what enables such a procedure.

Logistic regression in R

To run logistic regression in R, we use the `glm` function.

Example on the CORIS dataset:

```
> fm = glm(formula = chd ~ .,
           family = "binomial",
           data = coris)
> display(fm)
            coef.est ...
(Intercept) -6.15    ...
...
famhist      0.93    ...
...
obesity     -0.06    ...
...
```

Interpreting the output

Recall that if a coefficient is *positive*, it increases the probability that the outcome is 1 in the fitted model (since the logistic curve is increasing).

So, for example, `obesity` has a *negative* coefficient. What does this mean? Do you believe the implication?

The “divide by 4” rule

By differentating $g(\vec{X}\hat{\boldsymbol{\beta}})$, we find that the change in the (fitted) probability $Y = 1$ per unit change in X_j is:

$$\left(\frac{\exp(\vec{X}\boldsymbol{\beta})}{[1+\exp(\vec{X}\boldsymbol{\beta})]^2}\right)\hat{\beta}_j.$$

Note that the term in parentheses cannot be any larger than $1/4$.

Therefore, $|\hat{\beta}_j|/4$ is an upper bound on the magnitude of the change in the fitted probability that $Y = 1$, per unit change in X_j.

Classification

Logistic regression serves as a classifier in the following natural way:

- Given the estimated coefficients $\hat{\boldsymbol{\beta}}$, and a new covariate vector $\vec{X}$, compute $g(\vec{X}\hat{\boldsymbol{\beta}})$.

Classification

Logistic regression serves as a classifier in the following natural way:

- Given the estimated coefficients $\hat{\boldsymbol{\beta}}$, and a new covariate vector $\vec{X}$, compute $g(\vec{X}\hat{\boldsymbol{\beta}})$.
- If the resulting value is $> 1/2$ (equivalently, if $\vec{X}\hat{\boldsymbol{\beta}} > 0$), return $Y = 1$ as the predicted value.

Classification

Logistic regression serves as a classifier in the following natural way:

- Given the estimated coefficients $\hat{\beta}$, and a new covariate vector $\vec{X}$, compute $g(\vec{X}\hat{\beta})$.
- If the resulting value is $> 1/2$ (equivalently, if $\vec{X}\hat{\beta} > 0$), return $Y = 1$ as the predicted value.
- If the resulting value is $< 1/2$ (equivalently, if $\vec{X}\hat{\beta} < 0$), return $Y = 0$ as the predicted value.

(Note that logistic regression is an example of a *linear* classifier: the boundary between covariate vectors where we predict $Y = 1$ (resp., $Y = 0$) is linear.)

Tuning logistic regression

As with other classifiers, we can *tune* logistic regression to trade off false positives and false negatives.

In particular, suppose we choose a threshold $0 < t < 1$, and predict $Y = 1$ whenever $g(\vec{X}\hat{\beta}) > t$.

- When $t = 0$, we recover the classification rule on the preceding slide. This is the rule that minimizes average 0-1 loss on the training data.
- What happens to our classifier when $t \to \infty$?
- What happens to our classifier when $t \to -\infty$

As usual, you can plot an ROC curve for the resulting family of classifiers as t varies.

Regularized logistic regression

As with linear regression, *regularized* logistic regression is often used in the presence of many features.

In practice, the most common regularization technique is to add the penalty $-\lambda \sum_j |\hat{\beta}_j|$ to the maximum log likelihood problem; this is the equivalent of lasso for logistic regression.

As for linear regression, this penalty has the effect of selecting a subset of the parameters (for sufficient large values of the regularization penalty λ).

Neural networks and gradient descent

From linear to nonlinear models

So far we've seen learning algorithms that produce *linear* models:

- OLS: Fitted values are linear in covariates
- Logistic regression: Resulting classifier is linear

From linear to nonlinear models

So far we've seen learning algorithms that produce *linear* models:

- OLS: Fitted values are linear in covariates
- Logistic regression: Resulting classifier is linear

What if the true relationship between $\mathbf{X}$ and Y is *nonlinear*?

From linear to nonlinear models

So far we've seen learning algorithms that produce *linear* models:

- OLS: Fitted values are linear in covariates
- Logistic regression: Resulting classifier is linear

What if the true relationship between $\mathbf{X}$ and Y is *nonlinear*?

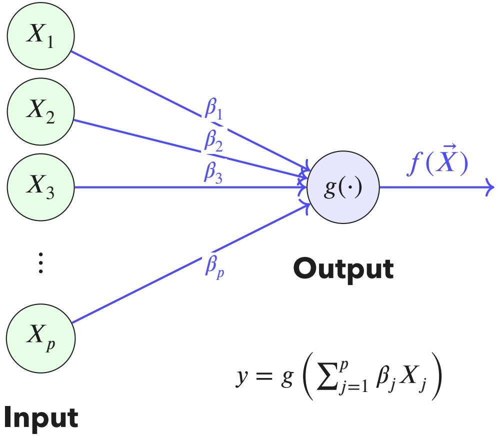
Neural networks provide a natural extension to capture nonlinear relationships.

Logistic regression as a one-layer neural network

To give a sense of how neural networks are constructed for classification, we recast logistic regression as a *one-layer neural network* (with sigmoid *activation function*):

Given *weights* $\boldsymbol{\beta}$, the *inputs* (the feature vector) $\vec{X}$ are mapped to output $f(\vec{X})$ according to:

$$f(\vec{X}) = g(\vec{X}\boldsymbol{\beta}).$$

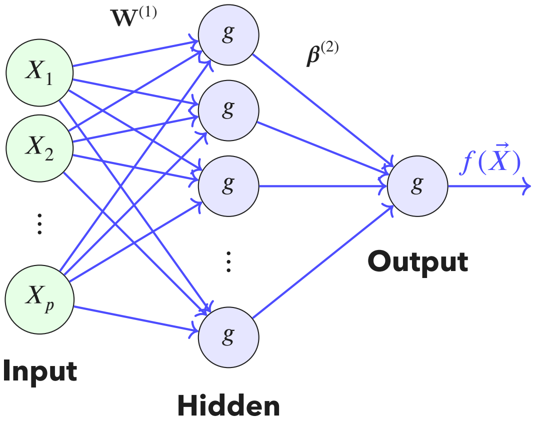


Adding layers: Architecture

We can build more complex, nonlinear relationships between features $\vec{X}$ and the outcome by *repeating* this approach.

E.g., a simple two-layer network with k hidden nodes:

- *Input layer*: Features $\vec{X} = (X_1, \ldots, X_p)$
- *Hidden layer*: k nodes, each computing $g(\cdot)$
- *Output layer*: Single node computing $g(\cdot)$



Adding layers: Mathematical formulation

Hidden layer: Each hidden node $j = 1, \dots, k$ computes

$$h_j = g\Big(\sum_{i=1}^{p} W_{ji}^{(1)} X_i + b_j^{(1)} \Big),$$

where $W_{ji}^{(1)}$ is the weight from input X_i to node j.

In vector form: $\mathbf{h} = g(\mathbf{W}^{(1)} \vec{X} + \mathbf{b}^{(1)})$, where $\mathbf{W}^{(1)}$ is a $k \times p$ *matrix*.

Output layer: The output node computes

$$f(\vec{X}) = g\Big(\sum_{j=1}^{k} \beta_j^{(2)} h_j + b^{(2)} \Big) = g(\boldsymbol{\beta}^{(2)\top} \mathbf{h} + b^{(2)}).$$

The constants $\mathbf{b}^{(1)}$ and $b^{(2)}$ are *bias* terms – analogous to intercept terms.

Fitting neural networks via optimization

Neural network *parameters* are $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \boldsymbol{\beta}^{(2)}, b^{(2)})$ (weights and bias terms). We write $f(\vec{X}|\boldsymbol{\theta})$ for the output given parameters $\boldsymbol{\theta}$.

We again interpret $f(\vec{X}; \boldsymbol{\theta})$ as a "probability" that $Y = 1$.

The parameters are *optimized* so that a target *loss function* is minimized; e.g, a common choice for classification is actually the log loss:

$$\text{Choose } \hat{\boldsymbol{\theta}} \text{ to minimize } -\sum_{i=1}^{n} \Big[Y_i \log f(\mathbf{X}_i; \boldsymbol{\theta}) + (1 - Y_i) \log(1 - f(\mathbf{X}_i; \boldsymbol{\theta})) \Big].$$

Fitting neural networks via optimization

Neural network *parameters* are $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \boldsymbol{\beta}^{(2)}, b^{(2)})$ (weights and bias terms). We write $f(\vec{X}|\boldsymbol{\theta})$ for the output given parameters $\boldsymbol{\theta}$.

We again interpret $f(\vec{X};\boldsymbol{\theta})$ as a "probability" that $Y = 1$.

The parameters are *optimized* so that a target *loss function* is minimized; e.g, a common choice for classification is actually the log loss:

$$\text{Choose } \hat{\boldsymbol{\theta}} \text{ to minimize } -\sum_{i=1}^{n} \Big[Y_i \log f(\mathbf{X}_i;\boldsymbol{\theta}) + (1 - Y_i)\log(1 - f(\mathbf{X}_i;\boldsymbol{\theta}))\Big].$$

In the case of a single layer network, this reduces *exactly* to fitting logistic regression by maximum likelihood.

The challenge: Non-convex optimization [$*$]

Like logistic regression, no closed-form solution exists.

However, *unlike* logistic regression, fitting neural networks is a *non-convex* optimization problem in general.

- Multiple local minima and “saddle points”
- No guarantee of finding the global optimum (i.e., global loss-minimizing choice of parameters)

This is fundamentally different from the convex problems we’ve seen before.

Gradient descent algorithm [$*$]

Basic idea: "Roll a ball downhill" to find the bottom of the loss surface.

Algorithm: Start with random parameters, then repeatedly update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \times \text{gradient of training loss at } \boldsymbol{\theta}_t$$

where:

- $\boldsymbol{\theta}_t$ = parameters at iteration t
- α = learning rate (step size)

Intuition: Move in the direction of steepest descent.

Backpropagation [$*$]

Challenge: How do we compute gradients efficiently through multiple layers?

Solution: *Backpropagation*: Use the chain rule to compute gradients layer by layer.

This algorithmic breakthrough made training deep neural networks practical.

Wide application

Neural networks can be made "deeper" by adding more layers.

More layers $\implies$ more parameters $\implies$ greater computational cost.

Neural networks have proven incredibly successful for a wide range of applications over the last two decades, including not only classification problems but also regression problems (e.g., using MSE as the training loss).

Note: In contrast to linear and logistic regression, there is no "simple" interpretation of the parameters!

Neural networks: R and Python [$*$]

Python is the language of choice for neural networks, with a number of widely adopted packages (notably, PyTorch, TensorFlow, and Keras).

In R, you can use the `torch` package for an R native library for building deep neural networks.

Tree-based methods and discrete optimization

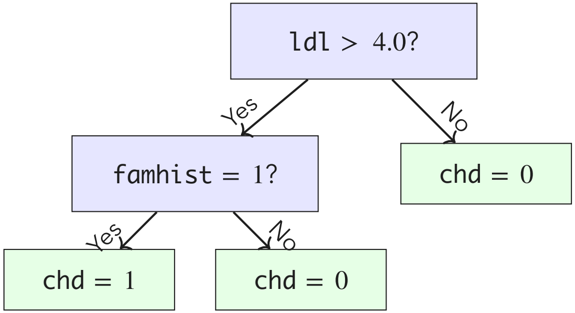
Decision trees: A different approach to prediction

Decision trees are an interpretable approach to prediction that also handle non-linear relationships naturally.

Basic idea: Partition the feature space using a series of binary splits, creating a tree-like structure.

- *Internal nodes*: Represent decisions
- *Branches*: Represent outcomes (Yes/No)
- *Leaf nodes*: Contain predictions

Example for CORIS data:



A different optimization paradigm [$*$]

In contrast to maximum likelihood and neural networks, tree-based methods involve *discrete/combinatorial* optimization:

- *Discrete* choices (which feature(s) to split on; what levels to split on, for factors)
- No derivatives available
- Different mathematical tools needed

“Greedy” optimization for decision trees

Typical algorithm (e.g., the CART algorithm): Build the tree one binary split at a time, choosing the “best” split at each step (according to a chosen training loss).

Best split: The one that minimizes loss function given the split.

“Greedy” choice: Locally optimal choice of split at each step, but resulting tree is not necessarily globally optimal.

Why greedy? Full optimization over all possible trees is computationally intractable.

See problem set to learn more.

Ensemble methods: Combining multiple trees

Individual decision trees can be prone to overfitting. *Ensemble methods* combine multiple trees to improve prediction:

Key examples:

- *Random forests*: Train many trees on different subsets of data and features, then average their predictions
  ("Wisdom of crowds": averaging predictions from multiple models often works better than any single model)

- *Gradient boosting*: Train trees sequentially, with each new tree correcting errors from previous trees

These methods typically achieve much better prediction accuracy than single trees.

Gradient boosting as optimization

Intuitive idea: Like learning to hit a bullseye – your first shot misses, so you look at where you missed and aim to correct that specific error with your next shot. Each new tree focuses on fixing the mistakes of all previous trees combined.

Example:

- Suppose true values: $(Y_1, Y_2, Y_3) = [3, 5, 3]$
- Tree 1 predicts: $[2, 4, 1]$
- Residuals (errors): $[3 - 2, 5 - 4, 3 - 1] = [1, 1, 2]$

Gradient boosting as optimization

Intuitive idea: Like learning to hit a bullseye – your first shot misses, so you look at where you missed and aim to correct that specific error with your next shot. Each new tree focuses on fixing the mistakes of all previous trees combined.

Example:

- Suppose true values: $(Y_1, Y_2, Y_3) = [3, 5, 3]$
- Tree 1 predicts: $[2, 4, 1]$
- Residuals (errors): $[3 - 2, 5 - 4, 3 - 1] = [1, 1, 2]$
- Tree 2 learns to predict these residuals, and predicts: $[0.8, 0.9, 1.7]$

Gradient boosting as optimization

Intuitive idea: Like learning to hit a bullseye – your first shot misses, so you look at where you missed and aim to correct that specific error with your next shot. Each new tree focuses on fixing the mistakes of all previous trees combined.

Example:

- Suppose true values: $(Y_1, Y_2, Y_3) = [3, 5, 3]$
- Tree 1 predicts: $[2, 4, 1]$
- Residuals (errors): $[3 - 2, 5 - 4, 3 - 1] = [1, 1, 2]$
- Tree 2 learns to predict these residuals, and predicts: $[0.8, 0.9, 1.7]$
- Combined prediction: $[2, 4, 1] + [0.8, 0.9, 1.7] = [2.8, 4.9, 2.7]$ $\implies$ closer to true values

Gradient boosting as optimization [$*$]

Suppose $\hat{f}^{(k)}(\vec{X})$ denotes the prediction made by the k'th tree built.

Compute the residuals $\hat{r}_i^{(k)} = Y_i - \hat{f}^{(k)}(\mathbf{X}_i)$.

Build a tree $\hat{h}^{(k)}$ to predict the residuals, using features $\mathbf{X}$.

"Move" predictions in the direction of the new tree:

$$\hat{f}^{(k+1)}(\vec{X}) = \hat{f}^{(k)}(\vec{X}) + \alpha \cdot h^{(k)}(\vec{X})$$

where $\alpha > 0$ is a parameter that governs how fast predictions are adjusted.

XGBoost [$*$]

Many successful machine learning approaches involves similar approaches to building high performance tree-based predictive models.

XGBoost (eXtreme Gradient Boost) is one such model that has enjoyed wide practical success across numerous applications for both classification and regression. Try it on your project!

Tree-based methods: R and Python [$*$]

In R, `rpart`, `ranger`, `gbm` can be used for decision trees, random forests, and gradient boosted trees, respectively.

In Python, all three are available through `scikit-learn`.

XGBoost is available as the `xgboost` package for both R and Python.

Comparing optimization approaches

Three optimization paradigms [$*$]

Three different approaches:

1. *Convex optimization*: OLS, logistic regression
   - Global optimum guaranteed
   - Often closed-form solutions available
2. *Non-convex smooth optimization*: Neural networks
   - Local optima, no global guarantee
   - Iterative methods
3. *Discrete/combinatorial optimization*: Trees
   - Greedy heuristics
   - NP-hard optimization problems
   - Ensemble methods

Computational trade-offs

Different algorithms have very different computational requirements; in general:

- OLS, logistic regression, and single decision trees are relatively *fast* to train
- Ensemble tree methods (random forests, gradient boosting) are *slower* depending on how many trees are trained
- Neural networks are typically *slowest* to train when used with many layers

(Of course, exact performance is context-dependent!)

Sample size trade-offs

The “best” algorithm depends heavily on your training sample size:

More complex models (neural networks, ensemble tree-based methods) have *many more parameters*,
and can *overfit* the training data when there are not enough training data points.

Linear methods, particularly with regularization, can be surprisingly powerful, even for large datasets.

On very large datasets, ensemble methods and neural networks, as well as their variants have been key drivers of the remarkable performance of machine learning for prediction.

The interpretability spectrum

Note that for OLS, logistic regression, and decision trees, it is straightforward to interpret the fitted model to understand how predictions are made given features.

By contrast, interpretation is not straightforward for the other methods:

- Neural networks: Complex nonlinear transformations through hidden layers
- Random forests: Averaging many trees obscures individual decisions
- Gradient boosting: Sequential corrections difficult to interpret

The bigger picture

All learning algorithms involve optimization, but they make different trade-offs.

There is no “best” algorithm – there are tradeoffs depending on:

- Computational constraints (time, resources)
- Size of training sample
- Interpretation of predictions

Appendix: Motivating the logistic population model [$*$]

Linearity of log odds ratio [$*$]

In these slides we'll talk about a few different motivations for the logistic population model that underlies logistic regression.

One way to interpret the model:

- Note that given a probability $0 < q < 1$, $q/(1-q)$ is called the *odds ratio*. The odds ratio lies in $(0, \infty)$.
- So logistic regression uses a model that suggests the *log odds ratio* of Y given $\vec{X}$ is linear in the covariates. Note the log odds ratio lies in $(-\infty, \infty)$.

Latent variables [$*$]

Another way to interpret the model is the *latent variable* approach:

- Suppose given a vector of covariates $\vec{X}$, a *logistic* random variable Z is sampled independently:

$$\mathbb{P}(Z < z) = g^{-1}(z).$$

- Define $Y = 1$ if $\vec{X}\boldsymbol{\beta} > Z$, and $Y = 0$ otherwise.
- By this definition, the event $Y = 1$ has probability $g^{-1}(\vec{X}\boldsymbol{\beta})$ – exactly the logistic population model.

Discrete choice modeling [$*$]

The latent variable interpretation is particularly popular in econometrics, where it is a first example of a *discrete choice modeling*.
For example, in modeling customer choice over whether or not to buy a product, suppose:

- Each customer has a feature vector $\vec{X}$.
- This customer's *reservation* utility level is Z: The customer purchases the item if $\vec{X}\boldsymbol{\beta} > Z$, and does not purchase otherwise.
- The probability the customer purchases the item is exactly the logistic population model.

This is a very basic example of a *random utility model* for customer choice.