Book Notes

Statistical Learning with R

Jon Geiger

March 24, 2022

Tentative Schedule

Winter 2022:

- Chapter 1: Introduction
- Chapter 2: Statistical Learning
- Chapter 4: Classification
- Chapter 5: Resampling Methods
- Chapter 6: Linear Model Selection & Regularization
- Chapter 8: Tree-Based Methods

Spring 2022:

- Chapter 7: Moving Beyond Linearity
- Chapter 9: Support Vector Machines
- Chapter 12: Unsupervised Learning

Organization

Chapters get single-# headers, and sections get double-## headers.

Sub-sections (e.g. Chapter 2.1.3) are separated with horizontal lines (created with three asterisks):

Chapter 1: Introduction

Statistical Learning is split up into **supervised learning** and **unsupervised learning**.

- *Supervised learning* "involves building a statistical modeling for predicting, or estimating, an output based on one or more inputs."
- *Unsupervised learning* involves "inputs but no supervising output," allowing us to learn structure from the data.

Some Key Datasets

`Wage` Data:

- Includes a number of factors relating to wages for a specific group of men from the Atlantic region of the U.S.
- Involves predicting a *quantitative* output, useful in *regression*

`Smarket` (Stock Market) Data:

- Contains daily movements in the S&P 500 in the five-year period between 2001 and 2005
- The goal is *classification*, or to predict whether the prices will increase or decrease on a given day.

`NCI60` Gene Expression Data:

- Contains 6,830 gene expression measurements for each of 64 cancer cell lines.
- This is a *clustering* problem, and we can analyze *principal components* of the data.

Purpose of the Book

"The purpose of *An Introduction to Statistical Learning* (ISL) is to facilitate the transition of statistical learning from an academic to a mainstream field."

ISL is based on four principles:

1. Many statistical learning methods are relevant and useful in a wide range of academic and non-academic disciplines, beyond just the statistical sciences.
2. Statistical learning should not be viewed as a series of black boxes.
3. While it is important to know what job is performed by each cog, it is not necessary to have the skills to construct the machine inside the box.
4. We presume that the reader is interested in applying statistical learning methods to real-world problems.

Notation

- $\mathbf{X}$ is an $n \times p$ matrix whose (i, j)th element is x_{ij}
  - Rather than $m \times n$ for m observations of n variables, we can think of $\mathbf{X}$ as a matrix (or spreadsheet) with n rows and p columns, or n observations of p variables.
- Vectors are column vectors by default.
- x_i is the vector of **rows** of $\mathbf{X}$, with length p:

$$x_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix}$$

- $\mathbf{x}$ is the vector of the **columns** of $\mathbf{X}$, with length n:

$$\mathbf{x}_j = \begin{pmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{nj} \end{pmatrix}$$

- y_i denotes the ith observation of the variable on which we wish to make predictions. We can write the set of all n observations in vector form as:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

- A vector of length n will be denoted in lower-case bold, such as:

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

Chapter 2: Statistical Learning

2.1: What is Statistical Learning?

Input Variables are also called *predictors*, *independent variables*, *features*, or just *variables*, denoted by X_i's. **Output Variables** are also called *responses* or *dependent variables*, denoted using the symbol Y.

With a quantitative response Y and p predictors $X = (X_1, X_2, \ldots, X_p)$, we assume there is some relationship between Y and X which takes the general form:

$$Y = f(X) + \epsilon$$

f is an unknown function of all the X_i's, and ϵ is a random *error term*, independent of X with a zero mean. f represents the *systematic* information that X provides about Y.

Statistical Learning refers to a set of approaches for estimating f.

We estimate f for two reasons: **prediction**, and **inference**.

We can *predict* Y with $\hat{Y} = \hat{f}(X)$, where $\hat{f}(X)$ represents our estimate for f, and $\hat{Y}$ is the resulting prediction of Y. Our goal is to minimize the reducible error (the error which can be changed by modifying f), keeping in mind that ϵ cannot be reduced.

In *inference*, we wish to understand the association between Y and each of the X_i's. A problem of inference could be driven by the following questions:

- Which predictors are associated with the response?
- What is the relationship between the response and the predictor?
- Can the relationship between Y and each predictor be adequately summarized using a linear equation, or is the relationship more complicated?

There are two methods to estimating f: **Parametric Methods**, and **Non-Parametric Methods**.

Parametric Methods involve a two-step, model-based approach:

1. We make an assumption about the functional form of f. One simple example might be that f is linear in X:
   $$f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$
   With a linear model, we estimate $p + 1$ coefficients from β_0 to β_p.
2. After model selection, we need to *fit* or *train* the model. This is the process of estimating the parameters β_0 through β_p such that in the case of the linear model above,
   $$Y \approx \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

The process is called *parametric* because we are *estimating parameters* to fit the model. This is generally simpler because it is easier to fit a set of parameters than it is to fit an entirely arbitrary function f. A parametric model has the disadvantage in that it typically does not match the true unknown form of f.

Non-parametric models have the advantage of not assuming a form for f, and can thus fit a wider range of possible shapes of f. These models need many more observations to work effectively. An example of a non-parametric model is a *thin-plate spline*, which can be either rough or smooth, where the roughness is analogous to overfitting data in a parametric model.

In choosing a model, there's a tradeoff between the *interpretability* of that model and the *flexibility* of that model. Generally, the more flexible a model is (bagging, boosting, SVMs, Deep Learning), the less interpretable the output is. Likewise, less flexible models (Subset Selection, Lasso, Least Squares) tend to be more interpretable.

- Choosing a more *restrictive* (less flexible) model is preferable in problems of inference, as the outcome needs to be interpretable.
- Conversely, more flexible models can be better for prediction problems when interpretability is not strictly necessary.

All examples from this past chapter have been examples of *supervised learning*. These techniques include linear regression, logistic regression, Generalized Additive Models (GAMs), boosting, Support Vector Machines (SVMs), and boosting. These all have in common that for a single observation x_i, there is an associated response measurement y_i.

Unsupervised learning tackles a different challenge, namely, that for each observation i, we have a set of measurements x_i with no associated response y_i. These types of problems are usually *cluster analysis* problems, where we try to find patterns, order, and/or groups in the data. More specifically, the goal is to figure whether or not the data fall into distinct groups, and if they do, which observations comprise those groups.

There do exist some problems which can be categorized as *semi-supervised learning* problems, namely scenarios in which our response data is incomplete. Namely, if we have both predictor and response measurements for m observations, where $m < n$, then for the remaining $m - n$ measurements we have predictor data but no response data.

Regression problems have a *quantitative response*. e.g. Least Squares.

Classification problems have a *categorical response*. e.g. Despite its name, logistic regression.

Some statistical methods, such as K-nearest neighbors (KNN) and boosting, can be used for either quantitative or categorical responses.

2.2: Assessing Model Accuracy

In evaluating statistical learning methods, we need some measure of how well its predictions match the observed data. In the case of regression, this is the *mean squared error* (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

A "good prediction" will yield a small MSE, as the error term $(y_i - \hat{f}(x_i))$ will be small for each observation.

With the case of linear regression, though, it is quite possible to overfit the data to perfectly capture each response (thus the MSE would be zero). We don't care how our model performs on our *training data*, but rather how it performs when we feed it *test data*. Because of this, we care about fitting parameters for $\hat{f}$ that will minimize the *test MSE* rather than the *training MSE*.

Often when choosing a model, it is useful to plot the training MSE and test MSE as a function of flexibility. Typically, the most flexible models will can yield a very low training MSE but will fail to produce a low test MSE (overfitting). Similarly, the least flexible models cannot produce a low MSE in either the training nor the test data, so the best models have a training MSE and test MSE that are roughly equal.

The test MSE for a given value x_0 can always be decomposed into the sum of three fundamental quantities:

1. The *variance* of $\hat{f}(x_0)$
2. The squared *bias* of $\hat{f}(x_0)$
3. The variance of the error terms ϵ

$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \text{Var}(\hat{f}(x_0)) + \left[\text{Bias}(\hat{f}(x_0))\right]^2 + \text{Var}(\epsilon)$$

In this case, $E\left(y_0 - \hat{f}(x_0)\right)^2$ is the *expected test MSE* at x_0. If our goal is to minimize the MSE, we need to estimate f such that the learning method achieves both a *low bias* and a *low variance*. This is known as the **bias-variance trade-off**.

The **variance** of a statistical learning method refers to the amount by which $\hat{f}$ would change if we estimated it using a different training data set. Ideally, the estimate for f should not vary much between training sets. This would be a *low variance*. In general, more flexible statistical methods have higher variance (overfitting to the training data).

The **bias** of a statistical learning method refers to the error that is introduced by approximating a real-life problem (usually very complicated) by a much simpler model. It's unlikely that a real-life problem actually has a linear relationship (as an example), so performing linear regression will result in some bias in estimating f.

It's very easy to achieve a model with low bias and high variance (draw a line that goes through every data point), as well as a model with high bias and low variance (fit a horizontal line through the data). Minimizing the test MSE is the key to finding the perfect bias-variance trade-off.

The bias-variance tradeoff also applies to the classification setting. In the case of classification, we assess our estimate of f with the training **error rate**, or the proportion of mistakes that are made if we apply our estimate to the training observations:

$$\frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i) \quad \text{where} \quad I = \begin{cases} 1 & \text{if } y_i \neq \hat{y}_i \\ 0 & \text{if } y_i = \hat{y}_i \end{cases}$$

In other words, I makes a binary indication of whether or not a classification error was made. Applying a classification method to test data, then, the *test error rate* with a set of test observations (x_0, y_0) is given by

$$\text{Ave}(I(y_0 \neq \hat{y}_0))$$

A good classifier is one for which the test error is minimized.

The **Bayes Classifier** is one which utilizes conditional probability to classify points in a two-class problem. Particularly, it seeks to find $Pr(Y = j|X = x_0)$, or "given that the observation is x_0, what's the probability that it belongs to class j?"

- The *Bayes error rate* is given by $1 - \max_j Pr(Y = j|X = x_0)$
- In general, the overall Bayes error rate is given by $1 - E\left(\max_j \Pr(Y = j|X = x_0)\right)$

The K**-Nearest Neighbors Classifier** (KNN) allows us to construct a decision boundary based on real data. It accomplishes this by choosing a test observation x_0, identifying the K closest points ($\mathcal{N}_0$), then estimates the conditional probability that x_0 would be in class j as the fraction of points whose response values equal j:

$$\Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

Similar to regression methods from earlier sections, KNN can be very flexible with low values of K, and for any set of training data, it can be important to optimize the value of K selected to minimize the test error.

Chapter 4: Classification

Classification is a statistical method used with a *qualitative* response variable.

Some classifiers covered in this chapter include logistic regression, linear discriminant analysis, quadratic discriminant analysis, naive Bayes, and K-nearest neighbors. These topics are used to segue into Generalized Linear Models and Poisson Regression.

4.1: An Overview of Classification

The `Default` data set will be used extensively, in predicting whether an individual will default on their credit card payment, on the basis of their annual income and monthly credit card balance.

We'll be building a model to predict `default` from `balance` (X_1) and `income` (X_2).

4.2: Why Not Linear Regression?

Linear regression doesn't work for classification problems because in order to predict an outcome, we need to *numerically encode* the categories as a quantitative response variable. This doesn't work because the categories rarely ever have any logical order.

In the case of predicting medical diagnoses, a response variable might look like:

$$Y = \begin{cases} 1 & \text{if } \texttt{stroke}, \\ 2 & \text{if } \texttt{drug overdose}, \\ 3 & \text{if } \texttt{epileptic seizure}. \end{cases}$$

The situation improves slightly if we choose to use the *dummy variable* approach, where we code a response variable which looks like:

$$Y = \begin{cases} 0 & \text{if } \texttt{stroke}; \\ 1 & \text{if } \texttt{drug overdose}. \end{cases}$$

In this case, we could have some estimates outside of the $[0, 1]$ range, which leads to difficulty interpreting probabilities.

Overall: (1) regression cannot accommodate non-binary classification, and (b) regression methods will not provide meaningful estimates of $\Pr(Y|X)$, even with just two classes.

4.3: Logistic Regression

Consider the binary classification problem, namely with the `Default` data set.

Logistic regression models the *probability* that Y belongs to a particular class, rather than modeling the classification directly.

The probability of defaulting given a certain balance can be expressed as

$$\Pr(\texttt{balance} = \texttt{Yes}|\texttt{balance}) \equiv p(\texttt{balance})$$

In general, we notate that $p(X) = \Pr(Y = 1|X)$. In logistic regression, we use the logistic function, which we can rearrange to create a linear regression problem.

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}} \quad \text{Logistic Function}$$

$$\therefore \quad \frac{p(X)}{1 - p(X)} = e^{\beta_0+\beta_1 X} \quad \text{Odds}$$

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X \quad \text{Log Odds / Logit}$$

Odds close to zero indicate low probabilities of default, and values close to ∞ indicate high probabilities of default. Interpreting this final equation is a bit tricky, as a one-unit increase in X will yield a β_1 increase in the log odds.

The coefficients β_0 and β_1 must be estimated to best fit our training data. This is done using *maximum likelihood estimation*. The likelihood function is:

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

The estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ are chosen to maximize this likelihood function.

In a problem of inference looking for association between `default` and `balance`, our null hypothesis would be $H_0 : \beta_1 = 0$, which makes sense because a one-unit increase in X should not affect Y at all. So the null logistic function will be $p(X) = \frac{e^{\beta_0}}{1+e^{\beta_0}}$. Similarly to linear regression, the z-statistic associated with β_1 is $z = \hat{\beta}_1/\text{SE}(\hat{\beta}_1)$.

For making predictions, we plug our estimates into the logistic function, so we have the equation:

$$\hat{p}(X) = \frac{e^{\hat{\beta}_0+\hat{\beta}_1 X}}{1 + e^{\hat{\beta}_0+\hat{\beta}_1 X}}$$

Multiple Logistic Regression extends nicely out from simple logistic regression, where we can generalize our log odds equation from earlier:

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

Where we have p predictors. This also means that our logistic equation for making predictions becomes:

$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}$$

We would also use maximum likelihood estimation to estimate all the coefficients $\beta_0, \beta_1, \ldots, \beta_p$.

What happens when we want to classify something which has more than one outcome? In the previous section we dealt with the three classes being `stroke`, `drug overdose`, and `epileptic seizure`. This is a problem of **multinomial logistic regression**. We select a single class to act as the *baseline*, then we say that:

$$\Pr(Y = k|X = x) = \begin{cases} \dfrac{e^{\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p}}{1+\sum_{l=1}^{K-1} e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}} & \text{for } k = 1, \ldots, K-1, \\ \dfrac{1}{1+\sum_{l=1}^{K-1} e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}} & \text{for } k = K. \end{cases}$$

Additionally, it can be shown that

$$\log\left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)}\right) = \beta_{k0} + \beta_{k1}x_1 + \cdots + \beta_{kp}x_p$$

An alternative coding for multinomial logistic regression is known as *softmax* coding, where, rather than selecting a baseline class, we treat all K classes symmetrically, and assume that for $k = 1, \ldots, K$,

$$\Pr(Y = k|X = x) = \frac{e^{\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p}}{1+\sum_{l=1}^{K} e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}}$$

So, we actually estimate coefficients for all K classes rather than just for $K-1$ classes. As a result of this, the log odds ratio between the kth and k'th classes is

$$\log\left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = k'|X = x)}\right) = (\beta_{k0} - \beta_{k'0}) + (\beta_{k1} - \beta_{k'1})x_1 + \cdots + (\beta_{kp} - \beta_{k'p})x_p$$

4.4: Generative Models for Classification

An alternative model for classification makes use of *Bayes' Theorem*, which states that

$$\Pr(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

This is not how the textbook defines Bayes' Theorem, so we will derive the textbook definition from this representation to make sense of the book's definition.

When talking about multinomial classification, let us assume that we have K total classes, and we want to find the probability that an observation belongs to a class k given information about that observation X. We can express this with Bayes' Theorem as:

$$\Pr(Y = k|X = x) = \frac{\Pr(X = x|Y = k) \cdot \Pr(Y = k)}{\Pr(X = x)}$$

Cleaning this up a bit, we can rewrite some of the probabilities:

$$\Pr(k|x) = p_k(x) = \frac{\Pr(x|k) \cdot \Pr(k)}{\Pr(x)}$$

If we have classes $k = 1, 2, 3, \ldots, K$, we can use the law of total probability to expand this formula a bit in

order to figure out what $\Pr(x)$ is.

$$\begin{aligned} p_k(x) &= \frac{\Pr(x|k) \cdot \Pr(k)}{\Pr(x)} \\ &= \frac{\Pr(x|k) \cdot \Pr(k)}{\Pr(x \cap 1) + \Pr(x \cap 2) + \cdots + \Pr(x \cap K)} \\ &= \frac{\Pr(x|k) \cdot \Pr(k)}{\Pr(x|1) \cdot \Pr(1) + \Pr(x|2) \cdot \Pr(2) + \cdots + \Pr(x|K) \cdot \Pr(K)} \\ &= \frac{\Pr(x|k) \cdot \Pr(k)}{\sum_{l=1}^{K} \Pr(x|l) \cdot \Pr(l)} \\ &= \frac{\Pr(k) \cdot \Pr(x|k)}{\sum_{l=1}^{K} \Pr(l) \cdot \Pr(x|l)} \end{aligned}$$

If we let $f_k(x)$ be the *(probability) density function* of X for an observation which comes from class k, and we let π_k represent the probability that an observation comes from class k (also called the **prior probability**), then we have:

$$p_k(x) = \frac{\pi_k f_k(x)}{\sum_{\ell=1}^{K} \pi_\ell f_\ell(x)}$$

Rather than directly modeling $p_k(x)$ (also called the **posterior probability**) as we do with logistic regression, we can estimate the different prior probabilities $\pi_1, \ldots, \pi_K$, and the various density functions for the K classes $f_1(x), \ldots, f_K(x)$. The prior probabilities are relatively easy to estimate by taking a random sample, but estimating $f_k(x)$ proves to be a slightly bigger challenge.

We'll talk about three classifiers that all use different estimates of $f_k(x)$: *linear discriminant analysis*, *quadratic discriminant analysis*, and *naive Bayes*.

Let's assume the following:

- We have $p = 1$ predictors.
- $f_k(x)$ is *normal* or *Gaussian*. In order to approximate $f_k(x)$, we have to make some assumptions about its shape.
- The variances of all the classes are equal. That is, that there is a shared variance term among all the classes, σ^2.

Then,

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{1}{2\sigma^2}(x - \mu_k)^2\right)$$

and

$$p_k(x) = \frac{\pi_k \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x - \mu_k)^2\right)}{\sum_{\ell=1}^{K} \pi_\ell \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x - \mu_\ell)^2\right)}$$

Our goal with classification is to choose the class k such that $p_k(x)$ is maximized. Maximizing $p_k(x)$ is equivalent to maximizing its logarithm, thus we want to assign x to a class k for which $\delta_k(x)$ is maximized, where $\delta_k(x)$ is given by:

$$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log(\pi_k)$$

In reality, we do not know the parameters of the Gaussian distribution from which we assume the observations come, we use *Linear Discriminant Analysis* to replace the parameters with statistics. We use the following estimations:

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i \qquad \hat{\sigma}^2 = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2 \qquad \hat{\pi}_k = \frac{n_k}{n}$$

The LDA classifier then assigns an observation $X = x$ to the class for which the *discriminant function* $\hat{\delta}_k(x)$ is the largest:

$$\hat{\delta}_k(x) = x \cdot \frac{\hat{\mu}_k}{\hat{\sigma}^2} - \frac{\hat{\mu}_k^2}{2\hat{\sigma}^2} + \log(\hat{\pi}_k)$$

Linear Disciminant Analysis is *linear* because the discriminant functions are linear functions of x.

When we have $p > 1$, we need to assume a multivariate Gaussian distribution with a class-specific mean vector and a common covariance matrix.

Without going through all the details, the Bayes classifier assigns an observation $X = x$ to the class for which $\delta_k(x)$ is the largest, where we have:

$$\delta_k(x) = x^T \mathbf{\Sigma}^{-1} \mu_k - \frac{1}{2} \mu_k^T \mathbf{\Sigma}^{-1} \mu_k + \log \pi_k$$

Here, x is an observation vector with p components, $\mathbf{\Sigma}$ is a common covariance matrix.

Quadratic Discriminant Analysis makes the same assumption as LDA about the Gaussian shape of f, but assumes that each class has its own covariance matrix. The discriminant function is then

$$\delta_k(x) = -\frac{1}{2}(x - \mu_k)^T \mathbf{\Sigma}_k^{-1} (x - \mu_k) - \frac{1}{2} \log |\mathbf{\Sigma}_k| + \log \pi_k$$

Naive Bayes takes a different approach than LDA and QDA in estimating $f_k(x)$. Rather than worrying about the covariances between the different predictors, we can assume that they are independent of one another. This means, then, that for f_{kj} being the density function of the jth predictor among observations in the kth class,

$$f_k(x) = f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)$$

This is a big convenience for modeling, since we don't need to worry about including marginal or joint distributions of the predictors. This assumption doesn't always hold, but is extremely convenient much of the time.

Now, the posterior probability is given by:

$$\Pr(Y = k | X = x) = p_k(x) = \frac{\pi_k \times f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)}{\sum_{\ell=1}^{K} \pi_\ell \times f_{\ell 1}(x_1) \times f_{\ell 2}(x_2) \times \cdots \times f_{\ell p}(x_p)}$$

The question now becomes, how do we model or estimate f_{kj}? We have a few options.

- If X is quantitative, we could simpply assume that each X_j is normally distributed, such that $(X_j | Y = k) \sim N(\mu_{jk}, \sigma_{jk}^2)$. In other words, we would assume that within each class, the jth predictor is drawn from a univariate normal distribution with mean μ_{jk} and variance σ_{jk}^2.

- Another option for a quantitative X is to use a non-parametric method such as creating a histogram or using a kernel density estimator (essentially a smoothed histogram) as an estimate for $f_{kj}(x_j)$.
- For qualitative X_j, we can use a sample proportion according to each class to estimate f_{kj}. Let's say that $X_j \in \{1, 2, 3\}$, and we have 100 observations in the kth class. Suppose that the jth predictor takes on values of 1, 2, and 3 in 32, 55, and 13 of those observations, respectively. Then we have:

$$\hat{f}_{kj}(x_j) = \begin{cases} 0.32 & \text{if } x_j = 1 \\ 0.55 & \text{if } x_j = 2 \\ 0.13 & \text{if } x_j = 3 \end{cases}$$

To recap, with K classes and p predictors, we are estimating $K \times p$ density functions. In other words, for each class k, there will be p density estimates. Applying the observation X to each of these density estimates in a class k should yield a probability (posterior probability) that X belongs to that class. See Figure 4.10 and its caption for a good visual explanation.

With a low number of predictors, naive Bayes will not necessarily outperform LDA or QDA because the reduction in variance is not super important. In scenarios with many predictors or with few training examples, though, naive Bayes will outperform LDA or QDA because of the assumption of independence.

4.5: A Comparison of Classification Methods

Without writing out all the derivations, we can look at a mathematical comparison between the different classification settings. In a setting with K classes, we look to maximize

$$\log\left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)}\right)$$

for $k = 1, \ldots, K$.

For the LDA setting, we end up with:

$$\begin{aligned} \log\left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)}\right) &= \log\left(\frac{\pi_k f_k(x)}{\pi_K f_K(x)}\right) \\ &= \cdots \\ &= a_k + \sum_{j=1}^{p} b_{kj} x_j \end{aligned}$$

Where $a_k = \log\left(\frac{\pi_k}{\pi_K}\right) - \frac{1}{2}(\mu_k + \mu_K)^T \mathbf{\Sigma}^{-1}(\mu_k - \mu_K)$ and b_{kj} is the jth component of $\mathbf{\Sigma}^{-1}(\mu_k - \mu_K)$. So LDA assumes that the log odds of the posterior probabilities is linear in x, just like logistic regression.

In the QDA setting, we have:

$$\begin{aligned} \log\left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)}\right) &= \log\left(\frac{\pi_k f_k(x)}{\pi_K f_K(x)}\right) \\ &= \cdots \\ &= a_k + \sum_{j=1}^{p} b_{kj} x_j + \sum_{j=1}^{p} \sum_{l=1}^{p} c_{kjl} x_j x_l \end{aligned}$$

where a_k, b_kj, and c_kjl are functions of π_k, π_K, μ_k, μ_K, $\mathbf{\Sigma}_k$, and $\mathbf{\Sigma}_K$

Finally, in the naive Bayes setting, we have:

$$\begin{aligned}\log\left(\frac{\Pr(Y=k|X=x)}{\Pr(Y=K|X=x)}\right) &= \log\left(\frac{\pi_k f_k(x)}{\pi_K f_K(x)}\right)\\ &= \cdots\\ &= a_k + \sum_{j=1}^{p} g_{kj}(x_j)\end{aligned}$$

where $a_k = \log\left(\frac{\pi_k}{\pi_K}\right)$ and $g_{kj}(x_j) = \log\left(\frac{f_{kj}(x_j)}{f_{Kj}(x_j)}\right)$. This takes the form of a *generalized additive model*, which is covered in chapter 7.

We can notice a few things about these results:

- LDA is a special case of QDA with $c_{kjl} = 0$ for all k, j, l.
- Any classifier with a linear decision boundary is a special case of naive Bayes with $g_{kj}(x_j) = b_{kj}x_j$. This also means that LDA is a specific case of naive Bayes.
  - This is not directly intuitive, as for LDA we assumed a shared within-class covariance matrix, and for naive Bayes we assumed independence across all features.
- Modeling f with naive Bayes using a one-dimensional Gaussian distribution gives us the LDA classifier where $\mathbf{\Sigma}$ is a diagonal matrix with the jth diagonal element is equal to σ_j^2.
- QDA and naive Bayes are completely separate from one another, but LDA is a special case of both. Because QDA has interaction terms, it has the potential to be more accurate in scenarios with high interactions between classes.

There is no one classifier that can do it all. In six scenarios, six different classifiers were tested on each scenario. In addition to Logistic Regression, LDA, QDA, and naive Bayes, K-nearest neighbors with $K = 1$ and with K chosen automatically from a cross-validation set were compared. When the decision boundary is highly non-linear and we have many observations, the nonparametric approaches such as KNN tend to work much better than parametric models.

4.6: Generalized Linear Models

For this section, consider the `Bikeshare` data, which describes the number of hourly users of a bike sharing program in DC. The response is `bikers`, which is a *count* variable, and the covariates are `mnth`, `hr`, `workingday`, `temp`, and `weathersit`.

First, let's explore multiple linear regression on the data. We fit the model:

$$Y = \sum_{j=1}^{p} X_j\beta_j + \epsilon$$

Performing multiple linear regression on the `Bikeshare` data is problematic for two reasons:

1. The `bikers` variable is fundamentally integer-valued, but a linear model assumes a continuous, normally distributed error term ϵ.
2. With just multiple linear regression, it turns out that 9.6% of the fitted values are negative, which doesn't make sense at all.

To remedy the second issue, we could perhaps fit a model to the log of bikers, which would yield all positive predictions:

$$\log(Y) = \sum_{j=1}^{p} X_j \beta_j + \epsilon$$

This, however, leads to difficulty in interpretation, namely that we would say that *a one-unit increase in X_j is associated with an increase in the mean of the log of Y by an amount β_j*. Also, this technique cannot be applied where 0 is in the range of Y.

To combat these issues, we can use **Poisson regression**.

If a random variable Y follows a Poisson distribution, it has a probability mass function given by:

$$\Pr(Y = k) = \frac{e^{-\lambda}\lambda^k}{k!} \quad \text{for } k = 0, 1, 2, ...$$

With a Poisson distribution, we know that the expected value *and* variance are both equal to λ.

The Poisson distribution is used to model *counts*. Let's consider:

- A particular hour of the day
- A particular set of weather conditions
- During a particular month of the year

For these specific parameters, we might have a Poisson distribution with $E(Y) = \lambda = 5$. So, for example, we would have $P(Y = 2) = \frac{e^{-5}5^2}{2!} = 0.084$. We expect this value of λ to vary according to each of our covariates (or predictors), so we write λ as a function of each of the predictors:

$$\lambda = \lambda(X_1, X_2, \ldots, X_p)$$

Because λ can only take on values greater than zero, we model the logarithm of lambda as being a linear function of the covariates:

$$\log(\lambda(X_1, ..., X_p)) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

Or equivalently,

$$\lambda(X_1, ..., X_p) = e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}$$

Now that we have a form for λ as a function of the covariates, we can write out our likelihood function:

$$L(\beta_0, \beta_1, \ldots, \beta_p) = \prod_{i=1}^{n} \frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}$$

And we can use this along with maximum likelihood estimation to find the values of our coefficients $\beta_0, \beta_1, \ldots, \beta_p$ that fit our model.

In order to interpret the Poisson model, we should notice that increasing X_j by one unit changes $\lambda = E(Y)$ by a factor of $\exp(\beta_j)$. With regards to the model itself, we also assume that the mean bike usage in an hour equals the variance of bike usage during that same hour. This relationship holds up nicely. Additionally, there are no negative fitted values since the Poisson distribution only holds for values ≥ 0.

Linear, logistic, and Poisson regression all share a few common characteristics:

1. Each approach uses predictors $X_1, \ldots, X_p$ to predict a response Y. We assume that Y belongs to a family of distributions. For linear, we assume Gaussian; for logistic, we assume Bernoulli; for Poisson, we assume a Poisson distribution.
2. Each approach models the mean of Y as a function of the predictors.
   For Linear Regression, we have:

$$E(Y|X_1, \ldots, X_p) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p.$$

   For Logistic Regression, we have:

$$\begin{aligned} E(Y|X_1, \ldots, X_p) &= \Pr(Y = 1|X_1, \ldots, X_p) \\ &= \frac{e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}}. \end{aligned}$$

   For Poisson Regression, we have:

$$E(Y|X_1, \ldots, X_p) = e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}.$$

The previous three equations can be expressed in terms of a *link function*, given by η. That is, that

$$\eta(E(Y|X_1, \ldots, X_p)) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

The link functions for these distributions are:

$$\begin{aligned} \text{Linear:} \quad & \eta(\mu) = \mu \\ \text{Logistic:} \quad & \eta(\mu) = \log\left(\frac{\mu}{1-\mu}\right) \\ \text{Poisson:} \quad & \eta(\mu) = \log(\mu) \end{aligned}$$

These three distributions (Gaussian, Bernoulli, and Poisson) all come from the *exponential family* of distributions, of which the Exponential, Gamma, and Negative Binomial Distributions are also members.

We create a regression problem by modeling Y as coming from a member of the exponential family, then transforming the mean of the response so that it's a linear function of the predictors, then we bunch those transformations up into the link function η. A regression approach that follows this idea is called a **generalized linear model (GLM)**.

Chapter 5: Resampling Methods

Resampling Methods refer to techniques which sample specific subsets of a training set to reveal more information about the fitted model. The two most common resampling methods are *cross-validation* and *bootstrap*.

Model Assessment: The process of evaluating a model's performance.

Model Selection: The process of selecting the proper level of flexibility for a model.

5.1: Cross-Validation

When evaluating the accuracy of a statistical learning method, it's difficult to calculate the test error when there is no designated test set. One method we can use is to hold out a certain subset of training observations, then applying the method to those held out observations.

This idea of holding out some observations brings upon the idea of the *validation set approach*, in which the available observations are broken into two parts: a *training set* and a *validation set/hold-out set*.

Validating the model on the validation set allows us to get a better estimate of the MSE, thus if we are testing many models, we can get a sense of which model is the best at predicting by choosing the model which minimizes the test MSE.

Two issues with the validation set approach are:

1. The MSE on the validation set is highly variable and can be unreliable with smaller data sets.
2. The validation set error rate may tend to *overestimate* the test error rate for the model fit on the whole data set.

Leave-one-out Cross-Validation (LOOCV) is a type of cross-validation which tries to address the validation set approach's drawbacks.

LOOCV involves splitting the data (size n) into two chunks: one of size $n-1$, and the other of size 1. In other words, a single observation is used for validation. However, this is done with every possible value of (x_i, y_i) in the training set, up to (x_n, y_n).

In LOOCV, the model is fit using the $n-1$ data points, and the remaining data point is used to calculate the test error. This is done for each data point, and the average error is taken, such that:

$$\text{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n} \text{MSE}_i$$

In this case, MSE_i is the individual error calculated from the ith data point, and CV_n simply takes the average of all those errors. This method is very *consistent* because it uses $n-1$ data points to fit the model each time, but in turn it is *computationally expensive* because it needs to fit the model n times.

For linear and polynomial regression, there's a fantastic shortcut which allows you to perform LOOCV with only one model fit:

$$\text{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1-h_i}\right)^2 \quad \text{where} \quad h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \bar{x})^2} \text{ is the leverage of point } x_i$$

While this particular form of the equation cannot be fit to every model, LOOCV is very general and can be fit to any kind of predictive model.

---

k-fold Cross-Validation is an alternative to LOOCV, where the set of observations is randomly split up into k *folds* of equal size. The first fold is used as the validation set, and the remaining $k-1$ folds are used to fit the model. This process is repeated such that every fold is used as the validation set, so it is repeated k times. We then have:

$$\text{CV}_{(k)} = \frac{1}{k}\sum_{i=1}^{k} \text{MSE}_i$$

By thinking for a second or two, we can notice that k-fold CV is a more general form of LOOCV, or rather, LOOCV is a special case of k-fold CV with $k = n$.

When we are trying to calculate the test MSE as a function of the flexibility, we are trying to find the amount of flexibility such that our test MSE is minimized. In this case, k-fold CV provides good estimates of the test MSE as compared to LOOCV, while being less computationally expensive.

It turns out that k-fold CV also provides better estimates of the test error rate than LOOCV does. Also, k-fold CV both reduces bias *and* variance in the procedure.

Generally, $k = 5$ or $k = 10$ are perfectly fine in estimating LOOCV.

Cross-Validation methods can also be used in classification problems, where the LOOCV error would be given by:

$$\text{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n} \text{Err}_i$$

Where $\text{Err}_i = I(y_i \neq \hat{y}_i)$ is the indicator function taking on values of 0 and 1.

In the case where we were to use a polynomial logistic regression model, we could use Cross-validation to estimate the degree of polynomial for which the CV error would be minimized. Adding polynomial terms to a logistic regression model makes a curved decision boundary, and adding too many degrees onto the polynomial will begin to increase the test error rate.

We can also use k-fold Cross Validation for something such as KNN, where we can choose the value of K which minimizes the CV error.

5.2: The Bootstrap

The **Bootstrap** is a statistical tool used to quantify the uncertainty associated with a given estimator or statistical learning method. It is commonly used to assess the variability associated with coefficients in fitting a statistical model.

As an example, consider a problem where we want to invest money in financial assets with returns of X and Y, respectively, where X and Y are random quantities (RVs). We'll invest a fraction α of our money in X, and the remaining $1-\alpha$ in Y. We want to minimize the risk, or variance of the investment, so we minimize $Var(\alpha X + (1-\alpha)Y)$. The value of α which minimizes this variance is

$$\alpha = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}}$$

Because we don't know these quantities, we can estimate them using samples, such that:

$$\hat{\alpha} = \frac{\hat{\sigma}_Y^2 - \hat{\sigma}_{XY}}{\hat{\sigma}_X^2 + \hat{\sigma}_Y^2 - 2\hat{\sigma}_{XY}}$$

Bootstrapping allows us to emulate the process of obtaining new sample sets, so that we can estimate the variability of a parameter without generating additional samples. In this case, we will repeatedly sample *with replacement* from the original data set, such that a single observation can be repeatedly chosen for the same sample.

From a data set Z, we can choose a value of B where we take B different bootstrap data sets $Z^{*1}, Z^{*2}, \ldots, Z^{*B}$ and calculate the sample statistics from those bootstrap sets, $\hat{\alpha}^{*1}, \hat{\alpha}^{*2}, \ldots, \hat{\alpha}^{*B}$.

We can calculate the mean and standard error of these parameter estimates with:

$$\hat{\alpha} = \frac{1}{B}\sum_{r=1}^{B} \hat{\alpha}^{*r} \quad \text{and} \quad \mathrm{SE}_B(\hat{\alpha}) = \sqrt{\frac{1}{B-1}\sum_{r=1}^{B}\left(\hat{\alpha}^{*r} - \frac{1}{B}\sum_{r'=1}^{B}\hat{\alpha}^{*r'}\right)^2}$$

Essentially, the bootstrap method is a technique which takes a sample of data, creates B sub-samples of that data with replacement, and calculates parameters and their variabilities based on those sub-samples.

Chapter 6: Linear Model Selection and Regularization

While the standard linear model, $Y = \beta_0 + \beta_1 X + \cdots + \beta_p X_p + \epsilon$, is very useful for problems of inference and even prediction, but there are improvements which can be made.

Alternative models can offer better *prediction accuracy* and *model interpretability*:

- If $n \gg p$, a linear model will have very low variance (deviation in the model's behavior from one set to the next). However, if p is comparable to n, or worse, $p > n$, the variance can be very high or even *infinite*
- It's easiest to interpret a model with only relevant variables. Since least squares coefficients are almost never *exactly* zero, we need to compensate by selecting the variables we use very carefully, using **feature selection** or **variable selection**.

This chapter looks at three alternatives to using least squares to fit a model:

1. **Subset Selection**: identifying a subset of the p predictors which we believe to be related to the response, then fitting least squares on that set.
2. **Shrinkage**: fitting a model involving all p predictors, then shrinking the estimated coefficients relative to the least squares estimates. This is also known as **regularization**. Because this shrinks coefficients, it can estimate some to be almost exactly zero, so this method also performs a form of automatic variable selection.
3. **Dimension Reduction**: involves projecting the p predictors into an M-dimensional subspace, where we have M different linear combinations of the variables. We then use these projections to fit a regression model.

6.1: Subset Selection

The **Best Subset Selection** method fits a linear model for each possible combination of predictors, so 2^p total models will be fit. This is done by the following algorithm:

1. Let $\mathcal{M}_0$ denote the *null model*, which contains no predictors. This model just predicts the sample mean for each observation ($\beta_0 = \bar{y}$)
2. For $k = 1, 2, \ldots, p$:
    a. Fit all $\binom{p}{k}$ models that contain exactly k predictors.
    b. Pick the best among these $\binom{p}{k}$ models, and call it $\mathcal{M}_k$. Here *best* is defined as having the smallest RSS, or equivalently largest R^2.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ using cross-validated prediction error, C_p (AIC), BIC, or adjusted R^2.

We cannot use RSS or R^2 to select from the $p + 1$ models because these values improve monotonically with more predictors. So we must use some other method for choosing the "best" model. In other words, increasing the number of predictors will always decrease the training set error, but we care about the test set error.

For something such as logistic regression, instead of ordering models by RSS in Step 2 of the algorithm, we would use *deviance*, which is negative two times the maximized log-likelihood.

Generally, best subset selection is very computationally intensive, since if $p = 20$, we would need to fit over a million models.

Stepwise Selection is a much more computationally efficient process which only fits a fraction of the models which Best Subset Selection uses.

The *Forward Stepwise Selection* algorithm is as follows:

1. Let $\mathcal{M}_0$ denote the null model, containing no predictors.
2. For all $k = 0, \ldots, p-1$:
    a. Consider all $p-k$ models that augment the predictors in $\mathcal{M}_k$ with one additional predictor.
    b. Choose the *best* among these $p-k$ models, and call it $\mathcal{M}_{k+1}$. Here *best* is defined as smallest RSS or highest R^2.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ using C_p (AIC), BIC, or adjusted R^2.

This process essentially starts out with a model with no predictors, then adds the best predictor on top of that. Then another predictor is placed on top of that, and so on. Then, out of all the $p+1$ models with $0, \ldots, p$ predictors, the best model is chosen.

The *Backward Stepwise Selection* algorithm is as follows:

1. Let $\mathcal{M}_p$ denote the full model, containing all p predictors.
2. For all $k = p, p-1, \ldots, 1$:
    a. Consider all k models that contain all but one of the predictors in $\mathcal{M}_k$, for a total of $k-1$ predictors.
    b. Choose the *best* among these k models, and call it $\mathcal{M}_{k-1}$. Here *best* is defined as smallest RSS or highest R^2.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ using C_p (AIC), BIC, or adjusted R^2.

This is the same as forward stepwise selection, but it begins with a full model and works its way backwards.

Generally, the training error is a poor estimate of the test error. We can either *adjust* the training error to account for bias, or we can directly estimate the test error using cross-validation.

Estimating the test error without a cross-validation method involves calculating several estimates. These are: C_p, *Akaike information criterion* (AIC), *Bayesian information criterion* (BIC), and *adjusted* R^2.

For a least squares model containing d predictors with n observations, we have:

- $C_p = \frac{1}{n}(\text{RSS} + 2d\hat{\sigma}^2)$ \ The C_p adds a penalty of $2d\hat{\sigma}^2$ to the training RSS to adjust for the underestimation of the test error. It turns out that if $\hat{\sigma}^2$ is an unbiased estimator of σ^2. We select the model with the lowest C_p value.
- $\text{AIC} = \frac{1}{n}(\text{RSS} + 2d\hat{\sigma}^2)$ \ The AIC is defined for a large class of models fit by maximum likelihood with Gaussian errors. In this case, the AIC is proportional to C_p, where constants are irrelevant due to minimization.
- $\text{BIC} = \frac{1}{n}(\text{RSS} + \log(n)d\hat{\sigma}^2)$ \ The BIC is derived from a Bayesian point of view, but gives a larger penalty to models with larger amounts of variables, so results in smaller model selections.
- Adjusted $R^2 = 1 - \frac{\text{RSS}/(n-d-1)}{\text{TSS}/(n-1)}$ \ Where normal R^2 is defined as $1 - \text{RSS}/\text{TSS}$, the adjusted R^2 takes into account the ratio of the number of predictors to the number of observations. In other words, adjusted R^2 will decrease with noise predictors, or predictors that add to d while only causing a small decrease in the RSS. The best model should be that which maximizes the Adjusted R^2.

Alternatively, the test error can be directly estimated using cross-validation. This used to be computationally prohibitive, but with modern computing power this is becoming a more attractive option.

Models can be selected using the *one-standard-error rule*. This rule says that we calculate the standard error of the test MSE for each model size, then select the smallest model for which the estimated test error is within one standard error of the lowest point on the curve. The rationale for this is that if models seem to be roughly equally good, we might as well choose the simplest model.

6.2: Shrinkage Methods

As an alternative to just choosing a subset of predictors (6.1), we can fit a model which contains all p predictors and *constrains* or *regularizes* the coefficient estimates towards zero. Recall that:

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

Least squares involves choosing parameters which minimize the RSS. In **Ridge Regression**, the ridge regression coefficient estimates $\hat{\beta}^R$ are the values which maximize:

$$\text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

where $\lambda \geq 0$ is a tuning parameter, determined separately. The second term here is called a *shrinkage penalty*, and is smallest when $\beta_1, \ldots, \beta_p$ are all close to zero. The tuning parameter λ, then, controls the impact that the second term has on estimating the regression coefficients. Ridge regression will estimate coefficients $\hat{\beta}^R$ for each value of λ, and selecting a good value of λ will yield a better test MSE.

If $\hat{\beta}$ is the vector of least squares coefficient estimates, then $||\hat{\beta}||$ is the magnitude or *norm* of this vector, and we call $||\hat{\beta}||_2$ is the ℓ_2 *norm*, and is defined as $||\hat{\beta}||_2 = \sqrt{\sum_{j=1}^{p} \hat{\beta}_j^2}$, and the ℓ_2 norm of the ridge regression coefficients are defined similarly. As the tuning parameter increases, the ratio of $||\hat{\beta}_\lambda^R||_2 / ||\hat{\beta}||_2$ will decrease. In other words, the parameters will generally decrease in magnitude as the tuning parameter is increased.

In standard least squares, coefficients are *scale invariant*, in that if each X_j is multiplied by some constant c, the coefficient will simply be multiplied by $1/c$ to compensate. This is not the case with ridge regression. Because of this, we need to make sure that all the predictors are on the same scale, so we standardize them using the formula:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_{ij} - \bar{x}_j)^2}}$$

where the denominator of the fraction is the estimated standard deviation of the jth predictor. This means that all the standardized predictors will have a standard deviation of one.

In terms of bias-variance trade-off, ridge regression decreases variance at the cost of increasing bias. This is because from set to set, the coefficients will generally be smaller, so the variability in the coefficients will also be smaller.

The **Lasso** is another alternative to simple linear regression, which overcomes the interpretive difficulty of not being able to fully eliminate some variables from the model. With the RSS as the same quantity as with ridge regression, the lasso coefficients, $\hat{\beta}_\lambda^L$ minimize the quantity

$$\text{RSS} + \lambda \sum_{j=1}^{p} |\beta_j|$$

The lasso and ridge regression are similar, but the lasso uses the ℓ_1 norm rather than the ℓ_2 norm, where the ℓ_1 norm of a coefficient vector β is given by $||\beta||_1 = \sum |\beta_j|$. With the tuning parameter λ sufficiently large, some coefficients will be exactly equal to zero. This creates a *sparse model*, which is a model which includes only a subset of the variables.

Because the lasso also serves as a model selection technique, it implicitly assumes that a number of the features have coefficients exactly equal to zero. This assumption is not always a good one, but in the case where there are certain features with low correlation, it can be useful to eliminate them completely. Neither regression technique always dominates the other.

Generally, ridge regression performs better with many predictors, and the lasso performs better with a smaller number of predictors.

Selecting the Tuning Parameter λ can be a tricky problem. Cross-validation allows us to choose a grid of λ values and compute the CV error for each value of λ as described in Chapter 5. Then, we'll select the value of λ for which the CV error is minimized, then re-fit the model using all available observations and the selected value of the tuning parameter.

This is highly important for determining which variables are *noise* and which are *signal*. The lasso does an excellent job at picking out variables which are actually significant without picking up variables which just add noise. See Figure 6.13 in the textbook (p. 251).

6.3: Dimension Reduction Methods

Let $Z_1, Z_2, \ldots, Z_M$ represent $M < p$ linear combinations of our original p predictors. That is,

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

for some constraints $\phi_{1m}, \phi_{2m}, \ldots, \phi_{pm}$, for $m = 1, \ldots, M$. In other words, the Z_m are the new, transformed predictors that we can use to fit our model with. We can then fit the linear regression model

$$y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} + \epsilon_i, \quad i = 1, \ldots, n$$

using least squares. The regression coefficients for reduced dimensionality are given by $\theta_0, \theta_1, \ldots, \theta_M$. Rather than fitting a model with p predictors in X space, we fit a model with M predictors in Z space. We can notice that:

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{m=1}^{M} \theta_m \sum_{j=1}^{p} \phi_{jm} x_{ij} = \sum_{j=1}^{p} \sum_{m=1}^{M} \theta_m \phi_{jm} x_{ij} = \sum_{j=1}^{p} \beta_j x_{ij}$$

where

$$\beta_j = \sum_{m=1}^{M} \theta_m \phi_{jm}$$

So we can treat this as a special case of linear regression. When $p > n$, selecting a value of $M \ll p$ can significantly reduce variance of the final model.

Choosing $Z_1, Z_2, \ldots, Z_M$, or choosing ϕ_{jm}'s, can be done in different ways. We'll look at *principal components* and *partial least squares*.

Principal Components Analysis (PCA) is useful for transforming a large number of features into a small number.

The **First Principal Component** direction of the data is that along which the observations *vary the most*. In other words, if the points were projected onto the line, and the variance along that line were measured, it would be maximized. This is also the line which is *as close as possible* to the data.

The **Second Principal Component** direction must be orthogonal to that of the first principal component. Similarly, this is such that the variance of the projected data on this component will be maximized.

Principal Components Regression involves taking the first M principal components and using those components as predictors in a linear regression model fit using least squares.

- We assume that *the directions in which* $X_1, \ldots, X_p$ *show the most variation are the directions that are associated with* Y. This turns out to be a pretty good approximation in most cases.
- We mitigate overfitting by being able to fit a model on fewer predictors since most or all of the information is contained within $Z_1, \ldots, Z_M$.
- For analyzing model performance, plots of variance, squared bias, and test MSE versus number of components are useful.
- When $M = p$, PCR just results in regular multiple linear regression.
- PCR is *not* guaranteed to outperform ridge regression or lasso, but can perform well if most of the data is described by a low number of principal components.
- PCR is *not* a feature selection method, and is very closely related to ridge regression, and (allegedly) ridge regression can even be thought of "as a continuous version of PCR." (consult section 3.5 of ESL for more detail)
- PCR is an *unsupervised* technique because the response Y is *not used* to determine the directions of the principal components.

Partial Least Squares is a *supervised* alternative to PCR which *does* use the response to form new predictors. Similar to PCR, we compute the PLS directions in order to perform regression.

- After standardizing the predictors, PLS computes the first direction Z_1 by setting each ϕ_{j1} equal to the coefficient from the simple linear regression of Y onto each X_j. That is, simple linear regression is performed for each predictor individually for Y.
- In computing $Z_1 = \sum_{j=1}^{p} \phi_{j1} X_j$, PLS places the highest weight on the variables that are most strongly related to the response.
- "To identify the second PLS direction we first *adjust* each of the variables for Z_1, by regressing each variable on Z_1 and taking *residuals*. These residuals can be interpreted as the remaining information that has not been explained by the first PLS direction. We the compute Z_2 using this *orthogonalized* data in exactly the same fashion as Z_1 was computed based on the original data."
- Once $Z_1, \ldots, Z_M$ are computed, we use least squares to fit a linear model to predict Y the same way as in PCR. As with PCR, the value of M is chosen by cross-validation. We standardize the predictors and response before performing PLS.
- PLS can reduce bias, but it has the potential to increase variance, so typically, PCR is used in the place of PLS.

6.4: Considerations in High Dimensions

With advances in technology over the past 20 years, it is commonplace to collect an extremely high amount of feature measurements (p very large), whereas the number of observations n is limited due to cost, availability, or other considerations. Two examples are as follows:

1. Rather than predicting blood pressure based on age, sex, and BMI, one might also collect measurements for half a million individual DNA mutations common to the population (SNPs). In this case, $n \approx 200$ and $p \approx 500,000$.
2. A marketing analyst might look at the words which appear in an individual user's search history. For a given user, each of the p search terms is scored present (0) or absent (1), creating a huge feature vector. If information is gathered for 1000 people, then $n = 1000$ and p is incredibly large, equal to the number of words in the digital lexicon being used.

If a data set has $p > n$, it is classified as *high-dimensional*.

If data is high-dimensional, least squares regression should not be performed. With $p \geq n$, least squares will always form a perfect fit to the training data, with zero variance. This is overfitting and we need to consider other methods for fitting data such as shrinkage methods and dimension reduction methods.

In high dimensions, as the number of variables increases to $p \approx n$, the value of R^2 tends to 1, the training MSE tends to zero, and the test MSE increases at an incredible rate due to it being incredibly biased.

When performing regression with high-dimensional data, it is useful to fit *less flexible* least squares models, such as using forward stepwise selection, ridge regression, lasso, and PCR.

As an example, when performing lasso with $p = 20$ features, a relatively low value of λ can be chosen to minimize the test MSE. However, when we have many more features not contributing any useful information, a relatively high value of λ should be chosen to scale down or eliminate the noise variables.

There are three main important points about regression in high dimensional data:

1. Regularization or shrinkage plays a key role in high-dimensional problems
2. Appropriate tuning parameter selection is crucial for good predictive performance
3. The test error tends to increase as the dimensionality of the problem increases, unless the additional features are truly associated with the response. This is known as the *curse of dimensionality*.

In general, *adding additional signal features that are truly associated with the response will improve the fitted model*. However, adding noise will lead to deterioration in the fitted model and an increased test set error.

Multicollinearity is a *huge* problem in high-dimensional data, since any feature can basically be written as a linear combination of other features.

In the case of having half-a-million SNPs, we might conclude from forward stepwise selection that 17 of those SNPs lead to a good predictive model on the training data. This just means that those 17 form one good model out of many possible models.

Reporting errors is also difficult with high-dimensional data, as we've seen that it's very easy to get a useless model with zero residuals, and thus SSE, p-values, and R^2 values should not be used. Because of this, it's very important to use a cross-validation set and calculate independent test MSE. Test MSE or R^2 is a valid measure of model fit, but training MSE is not.

Chapter 8: Tree-Based Methods

Tree-Based Methods are methods for regression and classification which *stratify* or *segment* the predictors into a number of simple regions. They are useful for interpretation, but typically do not compete well with other supervised learning methods. Tree-based methods are often combined to improve accuracy.

8.1: The Basics of Decision Trees

Decision trees can be used for both regression and classification problems.

A **regression tree** is a decision tree used for regression. In the context of baseball data, we can think of predicting a hitter's salary based on the number of years and number of hits that person has. We might first segment the data according to how long people have played, saying that they have either < 4.5 years or ≥ 4.5 years. Then, for the people who have played for more than 4.5 years, we might arrange the data based on who has more or less than 117.5 hits. We would then make predictions that estimate the salary for these two groups of people to be in the middle of these salaries.

The final regions on the tree are called the *terminal nodes* or *leaves*. The places where the predictors are split are called *internal nodes*. The segments that connect the nodes are called *branches*.

Stratifying the Feature Space is how we build a regression tree. There are two main steps:

1. We divide the predictor space $(X_1, X_2, \ldots, X_p)$ into J distinct, non-overlapping regions, $R_1, R_2, \ldots, R_J$.
2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observatiosn in R_j.

Our goal is to find "boxes" that minimize the RSS, given by:

$$\sum_{j=1}^{j} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean response for a specific region R_j.

Rather than considering every possible combination of regions, we take a *top-down, greedy* approach knwon as **recursive binary splitting**. This consists of the following steps:

1. Select a predictor X_j and a cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS. In particular, if we define the pair of half-planes (or regions) as

   $$R_1(j,s) = \{X|X_j < s\} \quad \text{and} \quad R_2(j,s) = \{X|X_j \geq s\},$$

   we seek the values of j and s which minimize the RSS for both of the regions, or:

   $$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2,$$

   where $\hat{y}_{R_1}$ is the mean response for the training observations in $R_1(j,s)$ and $\hat{y}_{R_2}$ is the mean response for the training observations in $R_2(j,s)$.
2. Repeat step 1 to split the data further, splitting one of the two previously identified regions. This will increase the number of regions by one, and the process can be repeated until we have the regions $R_1, \ldots, R_J$.

3. We then predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.

A good method for constructing trees which don't overfit to data are formed in a process called *tree pruning*, which creates a large tree T_0, and *prunes* it back to obtain a *subtree*. Creating every possible tree and testing it using a test MSE or cross-validation is very time-intensive and not efficient, so we can use a method called **cost complexity pruning**. This is summarized below.

We say that α is a nonnegative tuning parameter which controls a trade-off between a subtree's complexity and its fit to the training data. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{T} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T| \quad \text{or} \quad \alpha|T| + \sum_{m=1}^{T} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2$$

is as small as possible. $|T|$ is the number of terminal nodes of the tree T, R_m is the rectangle (subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the predicted response associated with R_m.

When $\alpha = 0$, $T = T_0$ (no pruning), but as we increase α, there's a penalty for choosing a tree with many nodes. This is similar to the shrinkage methods we saw in Chapter 6, in which λ was used to control the complexity of a linear model. α can be selected using cross-validation. The algorithm for building a regression tree is as follows:

1. Use recursive binary splitting to grow a large tree T_0 on the training data, stopping when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. That is, divide the trainign observations into K folds. For each $k = 1, \ldots, K$:
   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.
   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

   Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

A **classification tree** is a decision tree used for classification rather than regression. For regression trees, we used the mean response in a region. For classification trees, we use the *most commonly occurring class* for the training observations. We are also interested in the *class proportions* for each leaf (terminal node) of the tree.

The process of constructing a classification tree also uses recursive binary splitting, but since we can't use RSS, we could use the *classification error rate*, which is the fraction of training observations in that region which don't belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk})$$

where $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class. Unfortunately, this is not sensitive enough for tree growing, so we could use two other measures.

The **Gini index** is given by:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

which is a measure of the total variance across all the K classes. This takes on a low value if $\hat{p}_{mk}$ is close to zero or one, and can be thought of as *node purity*.

The **entropy** is an alternative to the Gini index, given by:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk},$$

and it turns out that entropy is very numerically similar to the Gini index. These three values are used as follows:

- When *building* a classification tree, either the Gini index or the entropy is used.
- When *pruning* a classification tree, any of the three approaches can be used.
- When *measuring the accuracy* of a classification tree, the classification error rate is used.

In comparing trees and linear models, we can make the fairly clear claim that when the data can be well-explained by a linear model, a linear model will outperform a tree-based approach. However, if there is a highly non-linear and complex relationship between the features and ther esponse, then decision trees will typically outperform classical approaches. Figure 8.7 on p. 339 sums this up quite nicely.

Pros of decision trees:

- Very easy to explain to people. Arguably easier than linear regression!
- Decision trees can be thought to more closely mirror human decision-making than regression and classification approaches from previous chapters.
- Trees can be displayed graphically, and are easily interpreted by non-experts
- Trees handle qualitative predictors without needing to create dummy variables.

Cons of decision trees:

- Trees don't have the same level of predictive accuracy as some other regrssion and classification approaches seen in this book.
- Trees are fairly non-robust. A small change in the data can cause a large change in the final estimated tree, as changes will cascade throughout the whole tree.

By aggregating many decision trees using methods like *bagging*, *random forests*, and *boosting*, the predictive performance of trees can be improved.

8.2: Bagging, Random Forests, Boosting, and Bayesian Additive Regression Trees

An **ensemble method** is an approach combining many simple "building block" models to obtain a single, potentially very powerful, model. The building blocks are sometimes known as *weak learners*, since they lead to mediocre predictions on their own.

Bagging is a general-purpose prodecure for reducing the variance of a statistical learning method, and is very closely related to bootstrapping. Decision trees suffer from very high variance, so if we are able to break our data down into smaller chunks, we can reduce the variance by *averaging a set of observations*.

Given a set of n independent observations $Z_1, \ldots, Z_n$ each with variance σ^2, the variance of $\bar{Z}$ is given by σ^2/n. We can take advantage of this fact to reduce the variance of a statistical learning method by sampling boostrapped data sets. In particular, we wish to take B boostrapped training data sets, we train our method on the bth boostrapped training set in order to get $\hat{f}^{*b}(x)$, and we average all the predictions to obtain:

$$\hat{f}_{\text{bag}} = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions for a final prediction of $\hat{f}_{\text{bag}}(x)$. These trees are deep but haven't been pruned. Each tree has low bias and high variance, and by taking the average of all of the trees, the variance is scaled down by a factor of $1/n$.

To apply bagging to classification trees, the simplest approach is to, instead of taking the average, use the most commonly occurring class, or a *majority vote*, among the B predictions. If $B/4$ of the bootstrapped samples predict class A for observation X_i, and $3B/4$ of the bootstrapped samples predict class B, then $\hat{f}_{\text{bag}}(X_i)$ would predict class B.

Luckily for bagging, since it's a bootstrapping method and involves averaging, using a high value of B will *not* overfit the data, but rather make the test error settle down quite nicely. Additionally, because a single bootstrap data set will, on average, use two-thirds of the data, wen can easily calculate an estimate for the test MSE, known as the *out-of-bag* error estimate, since it's calculated based on all of the observations *not* in the bag (about 1/3). It turns out that with sufficiently large B, the OOB error is virtually equivalent to LOOCV error, thus it's a very good estimate of the test error.

Because bagging is less interpretable than a simple decision tree (though much more flexible), we can get an overall summary of the importance of each predictor using the RSS (for regression) or the Gini index (for classification), measuring how much each of these values decreases due to splits in each of the variables. This summary is known as a *variable importance* summary, and graphical representations are often used with bagging trees since the trees themselves cannot be displayed.

Random Forests provide an improvement over bagged trees by *decorrelating* the trees. Random forests also use bootstrapped samples, but each time a plit is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors, and the split can only use one of those m predictors. We typically choose $m \approx \sqrt{p}$.

How does this work? What is decorrelation?

- Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. In the collection of bagged trees, most or all of the trees will use this strong predictor in the top split. This makes all the bagged trees *correlated* with one another.

- Averaging many highly correlated quantities does not lead to as large a reduction in variance as averaging many uncorrelated quantities.
- Forcing each split to consider only a subset of each predictors allows only $(p-m)/p$ splits to even consider the strong predictor, making the average of the results less variable and more reliable.

Random forests can be thought of as a more general form of bagging, where bagging is a special case of random forests with $m = p$. Random forests generally lead to a reduction in both test error and OOB error over bagging. Especially with something like genetic data, for which there may be certain genes which have a very large impact, random forests should provide a significant improvement over bagging since many of the bagged trees would be highly correlated.

Boosting, in the decision tree context, also involves combining a large number of decision trees, except that the trees are grown *sequentially*: each tree is grown using information from previously grown trees. Each tree is fit on a modified version of the original data set. Boosting is a *slow learner*, meaning that it takes many iterations to give accurate results.

Boosting has three tuning parameters:

- B: the number of trees. Boosting can overfit if B is too large, though it tends to happen slowly. We use cross-validation to select B.
- λ: the shrinkage parameter, a small positive number. This controls the rate at which boosting learns. Very small λ requires larger B to learn effectively. Essentially a rate of convergence, akin to the learning parameter for something like gradient descent.
- d: the number of splits in each tree. Often $d = 1$ works well, in which case there is one split with just two terminal nodes (called a *stump*). This is known as the *interaction depth*.

The algorithm for boosting for regression trees is as follows:

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set
2. For $b = 1, 2, \ldots, B$, repeat:
    a. Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
    b. Update $\hat{f}$ by adding in a shrunken version of the new tree:
    $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
    c. Update the residuals,
    $$r_i \leftarrow r_i - \lambda \hat{f}^b(x)$$
3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

This procedure offers an improvement over bagging or random forests because overfitting is very possible since those algorithms fit the data hard. Because boosting is a slow learner, it does not overfit to the training set and can yield lower test errors than either of the procedures above. Statistical learning approaches that tend to learn slowly, also tend to perform very well.

In boosting, because the growth of a particular tree takes into account the other trees that have already been grown, smaller trees ($d = 1$) are usually sufficient. This aids in interpretability (using stumps leads to an additive model).

Bayesian Additive Regression Trees (BART), is an ensemble learning method which combines principles from bagging/random forests, as well as boosting. Bagging and random forests make predictions from an average of regression trees, with separate trees built. Boosting uses a weighted sum of trees, where each tree tries to capture new information not yet accounted for. BART captures both of these methods' benefits in one algorithm.

Some notation:

- K: the number of regression trees
- B: the number of iterations for which the BART algorithm will run.
- $\hat{f}_k^b(x)$: the prediction at x for the kth regression tree used in the bth iteration.
- $\hat{f}^b(x) = \sum_{k=1}^K \hat{f}_k^b(x)$ for $b = 1, \ldots, B$.

The BART algorithm is as follows:

1. Let $\hat{f}_1^1(x) = \hat{f}_2^1(x) = \cdots = \hat{f}_K^1(x) = \frac{1}{nK} \sum_{i=1}^n y_i$.
2. Compute $\hat{f}^1(x) = \sum_{k=1}^K \hat{f}_k^1(x) = \frac{1}{n} \sum_{i=1}^n y_i$.
3. For $b = 2, \ldots, B$:
   (a) For $k = 1, 2, \ldots, K$:
      i. For $i = 1, \ldots, n$, compute the current partial resitual
      $$r_i = y_i - \sum_{k'<k} \hat{f}_{k'}^b(x_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(x_i)$$
      ii. Fit a new tree, $\hat{f}_k^b(x)$, to r_i, by randomly perturbing the kth tree from the previous iteration, $\hat{f}_k^{b-1}(x)$. Perturbations that improve the fit are favored.
   (b) Compute $\hat{f}^b(x) = \sum_{k=1}^K \hat{f}_k^b(x)$
4. Compute the mean after L burn-in samples,
   $$\hat{f}(x) = \frac{1}{B-L} sum_{b=L+1}^B \hat{f}^b(x)$$

.

In essence, within a particular iteration, the BART algorithm looks at potential changes it could make to the previous tree, makes a change, and then once an iteration is done, a sum of all the trees is computed. Once this sum has been computed for each iteration, the mean $\hat{f}(x)$ is calculated with the formula from step 4 of the algorithm.

BART performs extremely well in comparison with other tree-based methods, even if it is less interpretable than the other ones.

As a summary, we have the following:

- *Bagging* involves growing trees independently from random samples, with many similarities. Bagging can get caught in local optima.

- *Random Forests* involves growing trees independently from random samples, but each split is performed using a random subset of the features. Random forests do not get caught as often in local optima.
- *Boosting* uses the original data without taking random samples, and is a slow learning method. Each new tree is fit to the signal left over from earlier trees, and shrunken before being used.
- *BART* uses the original data, and trees are grown successively. Each tree is perturbed to avoid local optima and achieve a more thorough exploration of the model space.