Chapter 5. Classification

Data scientists are often faced with a problem that requires an automated decision. Is an email an attempt at phishing? Is a customer likely to churn? Is the web user likely to click on an advertisement? These are all *classification* problems. Classification is perhaps the most important form of prediction: the goal is to predict whether a record is a 0 or a 1 (phishing/not-phishing, click/don't click, churn/don't churn), or in some cases, one of several categories (for example, Gmail's filtering of your inbox into "primary," "social," "promotional," or "forums").

Often, we need more than a simple binary classification: we want to know the predicted probability that a case belongs to a class.

Rather than having a model simply assign a binary classification, most algorithms can return a probability score (propensity) of belonging to the class of interest. In fact, with logistic regression, the default output from R is on the log-odds scale, and this must be transformed to a propensity. A sliding cutoff can then be used to convert the propensity score to a decision. The general approach is as follows:

1. Establish a cutoff probability for the class of interest above which we consider a record as belonging to that class.

2. Estimate (with any model) the probability that a record belongs to the class of interest.

3. If that probability is above the cutoff probability, assign the new record to the class of interest.

The higher the cutoff, the fewer records predicted as 1 — that is, belonging to the class of interest. The lower the cutoff, the more records predicted as 1.

This chapter covers several key techniques for classification and estimating propensities; additional methods that can be used both for classification and numerical prediction are described in the next chapter.

MORE THAN TWO CATEGORIES?

The vast majority of problems involve a binary response. Some classification problems, however, involve a response with more than two possible outcomes. For example, at the anniversary of a customer's subscription contract, there might be three outcomes: the customer leaves, or "churns" ($Y=2$), goes on a month-to-month ($Y=1$) contract, or signs a new long-term contract ($Y=0$). The goal is to predict $Y = j$ for $j = 0$, 1 or 2. Most of the classification methods in this chapter can be applied, either directly or with modest adaptations, to responses that have more than two outcomes. Even in the case of more than two outcomes, the problem can often be recast into a series of binary problems using conditional probabilities. For example, to predict the outcome of the contract, you can solve two binary prediction problems:

- Predict whether $Y = 0$ or $Y > 0$.
- Given that $Y > 0$, predict whether $Y = 1$ or $Y = 2$.

In this case, it makes sense to break up the problem into two cases: whether the customer churns, and if they don't churn, what type of contract they will choose. From a model-fitting viewpoint, it is often advantageous to convert the multiclass problem to a series of binary problems. This is particularly true when one category is much more common than the other categories.

Naive Bayes

The naive Bayes algorithm uses the probability of observing predictor values, given an outcome, to estimate the probability of observing outcome $Y = i$, given a set of predictor values.[1]

KEY TERMS FOR NAIVE BAYES

Conditional probability

The probability of observing some event (say $X = i$) given some other event (say $Y = i$), written as $P(X_i \mid Y_i)$.

Posterior probability

The probability of an outcome after the predictor information has been incorporated (in contrast to the *prior probability* of outcomes, not taking predictor information into account).

To understand Bayesian classification, we can start out by imagining "non-naive" Bayesian classification. For each record to be classified:

1. Find all the other records with the same predictor profile (i.e., where the predictor values are the same).
2. Determine what classes those records belong to and which class is most prevalent (i.e., probable).
3. Assign that class to the new record.

The preceding approach amounts to finding all the records in the sample that are exactly like the new record to be classified in the sense that all the predictor values are identical.

NOTE

Predictor variables must be categorical (factor) variables in the standard naive Bayes algorithm. See "Numeric Predictor Variables" for two workarounds for using continuous variables.

Why Exact Bayesian Classification Is Impractical

When the number of predictor variables exceeds a handful, many of the records to be classified will be without exact matches. This can be understood in the context of a model to predict voting on the basis of demographic variables. Even a sizable sample may not contain even a single match for a new record who is a male Hispanic with high income from the US Midwest who voted in the last election, did not vote in the prior election, has three daughters and one son, and is divorced. And this is just eight variables, a small number for most classification problems. The addition of just a single new variable with five equally frequent categories reduces the probability of a match by a factor of 5.

WARNING

Despite its name, naive Bayes is not considered a method of Bayesian statistics. Naive Bayes is a data–driven, empirical method requiring relatively little statistical expertise. The name comes from the *Bayes rule*–like calculation in forming the predictions — specifically the initial calculation of predictor value probabilities given an outcome, and then the final calculation of outcome probabilities.

The Naive Solution

In the naive Bayes solution, we no longer restrict the probability calculation to those records that match the record to be classified. Instead, we use the entire data set. The naive Bayes modification is as follows:

1. For a binary response $Y = i$ ($i = 0$ or 1), estimate the individual conditional probabilities for each predictor $P(X_j \mid Y = i)$; these are the probabilities that the predictor value is in the record when we observe $Y = i$. This probability is estimated by the proportion of X_j values among the $Y = i$ records in the training set.

2. Multiply these probabilities by each other, and then by the proportion of records belonging to $Y = i$.

3. Repeat steps 1 and 2 for all the classes.

4. Estimate a probability for outcome i by taking the value calculated in step 2 for class i and dividing it by the sum of such values for all classes.

5. Assign the record to the class with the highest probability for this set of predictor values.

This naive Bayes algorithm can also be stated as an equation for the probability of observing outcome $Y = i$, given a set of predictor values $X_1, \cdots, X_p$:

$$P(X_1, X_2, \cdots, X_p)$$

The value of $P(X_1, X_2, \cdots, X_p)$ is a scaling factor to ensure the probability is between 0 and 1 and does not depend on Y:

$$P(X_1, X_2, \cdots, X_p) = P(Y = 0)(P(X_1 \mid Y = 0)P(X_2 \mid Y = 0) \cdots P(X_p \mid Y = 0)) + P(Y = 1)(P(X_1 \mid Y = 1)P(X_2 \mid Y = 1) \cdots P(X_p \mid Y = 1))$$

Why is this formula called “naive”? We have made a simplifying assumption that the *exact conditional probability* of a vector of predictor values, given observing an outcome, is sufficiently well estimated by the product of the individual conditional probabilities $P(X_j \mid Y = i)$. In other words, in estimating

$P(X_j \mid Y = i)$ instead of $P(X_1, X_2, \cdots X_p \mid Y = i)$, we are assuming X_j is *independent* of all the other predictor variables X_k for $k \neq j$.

Several packages in R can be used to estimate a naive Bayes model. The following fits a model using the `klaR` package:

```
library(klaR)
naive_model <- NaiveBayes(outcome ~ purpose_ + home_ + emp_len_,
                          data = na.omit(loan_data))
naive_model$table
$purpose_
          var
grouping   credit_card debt_consolidation home_improvement major_purchase
  paid off   0.1857711          0.5523427       0.07153354     0.05541148
  default    0.1517548          0.5777144       0.05956086     0.03708506
          var
grouping      medical      other small_business
  paid off 0.01236169 0.09958506     0.02299447
  default  0.01434993 0.11415111     0.04538382

$home_
          var
grouping    MORTGAGE        OWN      RENT
  paid off 0.4966286 0.08043741 0.4229340
  default  0.4327455 0.08363589 0.4836186

$emp_len_
          var
grouping    > 1 Year   < 1 Year
  paid off 0.9690526 0.03094744
  default  0.9523686 0.04763140
```

The output from the model is the conditional probabilities $P(X_j \mid Y = i)$. The model can be used to predict the outcome of a new loan:

```
new_loan
        purpose_    home_  emp_len_
1 small_business MORTGAGE  > 1 Year
```

In this case, the model predicts a default:

```
predict(naive_model, new_loan)
$class
[1] default
Levels: paid off default

$posterior
      paid off   default
[1,] 0.3717206 0.6282794
```

The prediction also returns a `posterior` estimate of the probability of default.

The naive Bayesian classifier is known to produce *biased* estimates. However, where the goal is to *rank* records according to the probability that $Y = 1$, unbiased estimates of probability are not needed and naive Bayes produces good results.

Numeric Predictor Variables

From the definition, we see that the Bayesian classifier works only with categorical predictors (e.g., with spam classification, where presence or absence of words, phrases, characters, and so on, lies at the heart of the predictive task). To apply naive Bayes to numerical predictors, one of two approaches must be taken:

- Bin and convert the numerical predictors to categorical predictors and apply the algorithm of the previous section.
- Use a probability model — for example, the normal distribution (see "Normal Distribution") — to estimate the conditional probability $P(X_j \mid Y = i)$.

> **CAUTION**
>
> When a predictor category is absent in the training data, the algorithm assigns *zero probability* to the outcome variable in new data, rather than simply ignoring this variable and using the information from other variables, as other methods might. This is something to pay attention to when binning continuous variables.

KEY IDEAS

- Naive Bayes works with categorical (factor) predictors and outcomes.
- It asks, "Within each outcome category, which predictor categories are most probable?"
- That information is then inverted to estimate probabilities of outcome categories, given predictor values.

Further Reading

- *Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, 2009).

- There is a full chapter on naive Bayes in *Data Mining for Business Analytics*, 3rd ed., by Galit Shmueli, Peter Bruce, and Nitin Patel (Wiley, 2016, with variants for R, Excel, and JMP).

Discriminant Analysis

Discriminant analysis is the earliest statistical classifier; it was introduced by R. A. Fisher in 1936 in an article published in the *Annals of Eugenics* journal.[2]

KEY TERMS FOR DISCRIMINANT ANALYSIS

Covariance
A measure of the extent to which one variable varies in concert with another (i.e., similar magnitude and direction).

Discriminant function
The function that, when applied to the predictor variables, maximizes the separation of the classes.

Discriminant weights
The scores that result from the application of the discriminant function, and are used to estimate probabilities of belonging to one class or another.

While discriminant analysis encompasses several techniques, the most commonly used is *linear discriminant analysis*, or *LDA*. The original method proposed by Fisher was actually slightly different from LDA, but the mechanics are essentially the same. LDA is now less widely used with the advent of more sophisticated techniques, such as tree models and logistic regression.

However, you may still encounter LDA in some applications and it has links to other more widely used methods (such as principal components analysis; see "Principal Components Analysis"). In addition, discriminant analysis can provide a measure of predictor importance, and it is used as a computationally efficient method of feature selection.

WARNING

Linear discriminant analysis should not be confused with Latent Dirichlet Allocation, also referred to as LDA. Latent Dirichlet Allocation is used in text and natural language processing and is unrelated to linear discriminant analysis.

Covariance Matrix

To understand discriminant analysis, it is first necessary to introduce the concept of *covariance* between two or more variables. The covariance measures the relationship between two variables x and z. Denote the mean for each variable by $\bar{x}$ and $\bar{z}$ (see "Mean"). The covariance $s_{x,z}$ between x and z is given by:

$$s_{x,z} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(z_i - \bar{z})}{n-1}$$

where n is the number of records (note that we divide by $n - 1$ instead of n: see "Degrees of Freedom, and n or $n - 1$?").

As with the correlation coefficient (see "Correlation"), positive values indicate a positive relationship and negative values indicate a negative relationship. Correlation, however, is constrained to be between –1 and 1, whereas covariance is on the same scale as the variables x and z. The *covariance matrix* Σ for x and z consists of the individual variable variances, s_x^2 and s_y^2, on the diagonal (where row and column are the same variable) and the covariances between variable pairs on the off-diagonals.

$$\hat{\Sigma} = \begin{bmatrix} s_x^2 & s_{x,z} \\ s_{x,z} & s_z^2 \end{bmatrix}$$

NOTE

Recall that the standard deviation is used to normalize a variable to a *z*-score; the covariance matrix is used in a multivariate extension of this standardization process. This is known as

Mahalanobis distance (see Other Distance Metrics) and is related to the LDA function.

Fisher's Linear Discriminant

For simplicity, we focus on a classification problem in which we want to predict a binary outcome *y* using just two continuous numeric variables (x, z). Technically, discriminant analysis assumes the predictor variables are normally distributed continuous variables, but, in practice, the method works well even for nonextreme departures from normality, and for binary predictors. Fisher's linear discriminant distinguishes variation *between* groups, on the one hand, from variation *within* groups on the other. Specifically, seeking to divide the records into two groups, LDA focuses on maximizing the "between" sum of squares $SS_{between}$ (measuring the variation between the two groups) relative to the "within" sum of squares SS_{within} (measuring the within-group variation). In this case, the two groups correspond to the records (x_0, z_0) for which $y = 0$ and the records (x_1, z_1) for which $y = 1$. The method finds the linear combination $w_x x + w_z z$ that maximizes that sum of squares ratio.

$$\frac{SS_{between}}{SS_{within}}$$

The between sum of squares is the squared distance between the two group means, and the within sum of squares is the spread around the means within each group, weighted by the covariance matrix. Intuitively, by maximizing the between sum of squares and minimizing the within sum of squares, this method yields the greatest separation between the two groups.

A Simple Example

The `MASS` package, associated with the book *Modern Applied Statistics with S* by W. N. Venables and B. D. Ripley (Springer, 1994), provides a function for LDA with R. The following applies this function to a sample of loan data using two predictor variables, `borrower_score` and `payment_inc_ratio`, and prints out the estimated linear discriminator weights.

```
library(MASS)
loan_lda <- lda(outcome ~ borrower_score + payment_inc_ratio,
                    data=loan3000)
loan_lda$scaling
                         LD1
borrower_score     -6.2962811
payment_inc_ratio   0.1288243
```

USING DISCRIMINANT ANALYSIS FOR FEATURE SELECTION

If the predictor variables are normalized prior to running LDA, the discriminator weights are measures of variable importance, thus providing a computationally efficient method of feature selection.

The `lda` function can predict the probability of "default" versus "paid off":

```
pred <- predict(loan_lda)
head(pred$posterior)
       paid off   default
25333 0.5554293 0.4445707
27041 0.6274352 0.3725648
7398  0.4014055 0.5985945
35625 0.3411242 0.6588758
17058 0.6081592 0.3918408
2986  0.6733245 0.3266755
```

A plot of the predictions helps illustrate how LDA works. Using the output from the `predict` function, a plot of the estimated probability of default is produced as follows:

```
lda_df <- cbind(loan3000, prob_default=pred$posterior[,'default'])
ggplot(data=lda_df,
       aes(x=borrower_score, y=payment_inc_ratio, color=prob_default)) +
  geom_point(alpha=.6) +
  scale_color_gradient2(low='white', high='blue') +
geom_line(data=lda_df0, col='green', size=2, alpha=.8) +
```

The resulting plot is shown in Figure 5-1.

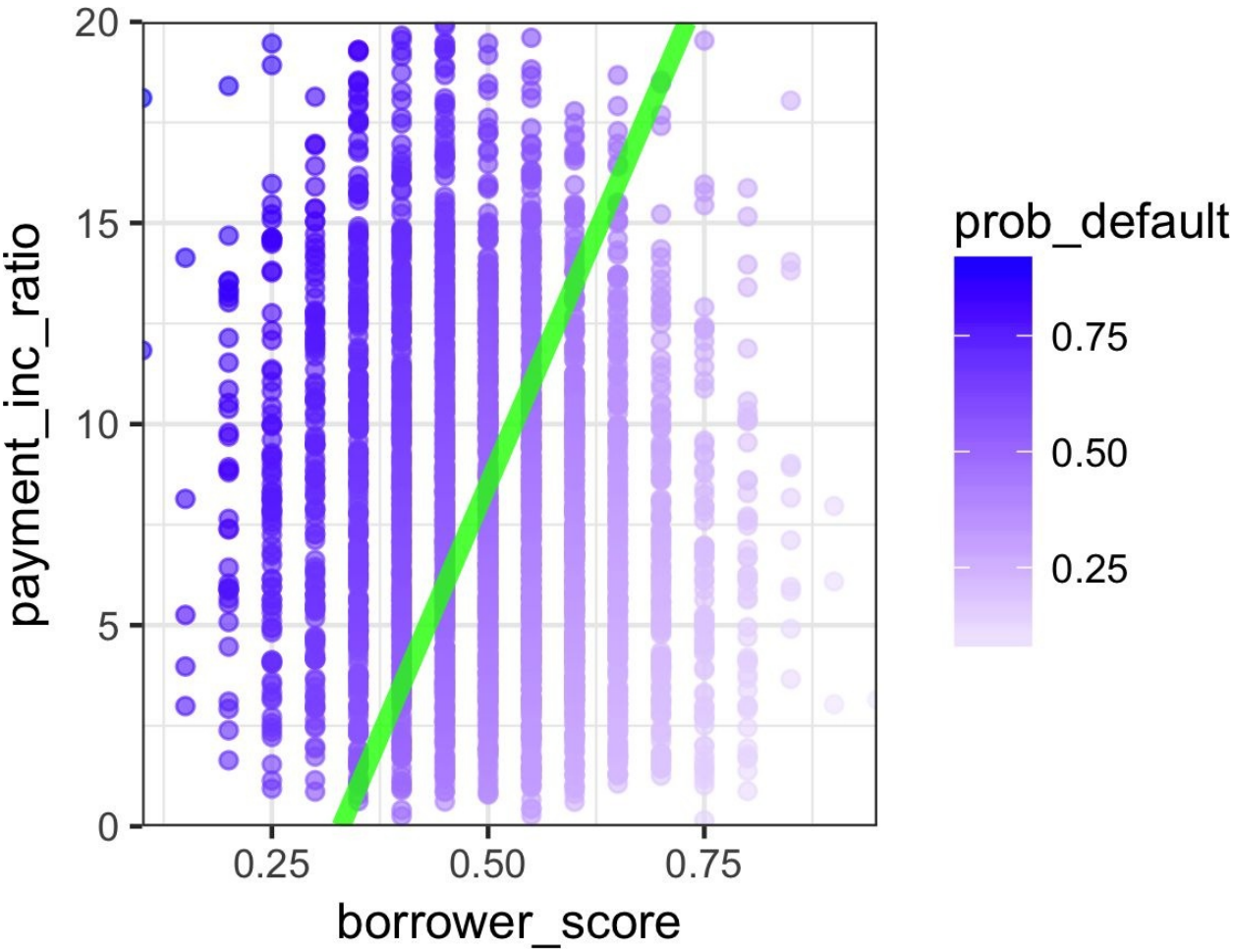

Figure 5-1. LDA prediction of loan default using two variables: a score of the borrower's creditworthiness and the payment to income ratio.

Using the discriminant function weights, LDA splits the predictor space into two regions as shown by the solid line. The predictions farther away from the line have a higher level of confidence (i.e., a probability further away from 0.5).

EXTENSIONS OF DISCRIMINANT ANALYSIS

More predictor variables: while the text and example in this section used just two predictor variables, LDA works just as well with more than two predictor variables. The only limiting factor is the number of records (estimating the covariance matrix requires a sufficient number of records per variable, which is typically not an issue in data science applications).

Quadratic Discriminant Analysis: There are other variants of discriminant analysis. The best known is quadratic discriminant analysis (QDA). Despite its name, QDA is still a linear discriminant function. The main difference is that in LDA, the covariance matrix is assumed to be the same for the two groups corresponding to $Y = 0$ and $Y = 1$. In QDA, the covariance matrix is allowed to be different for the two groups. In practice, the difference in most applications is not critical.

KEY IDEAS FOR DISCRIMINANT ANALYSIS

- Discriminant analysis works with continuous or categorical predictors, as well as categorical outcomes.
- Using the covariance matrix, it calculates a *linear discriminant function*, which is used to distinguish records belonging to one class from those belonging to another.
- This function is applied to the records to derive weights, or scores, for each record (one weight for each possible class) that determines its estimated class.

Further Reading

- *Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, Jerome Freidman, and its shorter cousin, *An Introduction to Statistical Learning*, by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (both from Springer). Both have a section on discriminant analysis.

- *Data Mining for Business Analytics*, 3rd ed., by Galit Shmueli, Peter Bruce, and Nitin Patel (Wiley, 2016, with variants for R, Excel, and JMP) has a full chapter on discriminant analysis.

- For historical interest, Fisher's original article on the topic, "The Use of Multiple Measurements in Taxonomic Problems," as published in 1936 in *Annals of Eugenics* (now called *Annals of Genetics*) can be found online.

Logistic Regression

Logistic regression is analogous to multiple linear regression, except the outcome is binary. Various transformations are employed to convert the problem to one in which a linear model can be fit. Like discriminant analysis, and unlike *K*-Nearest Neighbor and naive Bayes, logistic regression is a structured model approach, rather than a data-centric approach. Due to its fast computational speed and its output of a model that lends itself to rapid scoring of new data, it is a popular method.

KEY TERMS FOR LOGISTIC REGRESSION

Logit

The function that maps the probability of belonging to a class with a range from ± ∞ (instead of 0 to 1).

Synonym

Log odds (see below)

Odds

The ratio of "success" (1) to "not success" (0).

Log odds

The response in the transformed model (now linear), which gets mapped back to a probability.

How do we get from a binary outcome variable to an outcome variable that can be modeled in linear fashion, then back again to a binary outcome?

Logistic Response Function and Logit

The key ingredients are the *logistic response function* and the *logit*, in which we map a probability (which is on a 0–1 scale) to a more expansive scale suitable for linear modeling.

The first step is to think of the outcome variable not as a binary label, but as the probability *p* that the label is a "1." Naively, we might be tempted to model *p* as a linear function of the predictor variables:

$$p = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_q x_q,$$

However, fitting this model does not ensure that *p* will end up between 0 and 1, as a probability must.

Instead, we model *p* by applying a *logistic response* or *inverse logit* function to the predictors:

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_q x_q)}}.$$

This transform ensures that the *p* stays between 0 and 1.

To get the exponential expression out of the denominator, we consider *odds* instead of probabilities. Odds, familiar to bettors everywhere, are the ratio of "successes" (1) to "nonsuccesses" (0). In terms of probabilities, odds are the probability of an event divided by the probability that the event will not occur. For example, if the probability that a horse will win is 0.5, the probability of "won't win" is (1 – 0.5) = 0.5, and the odds are 1.0.

$$\text{Odds}(Y = 1) = \frac{p}{1 - p}.$$

We can obtain the probability from the odds using the inverse odds function:

$$p = \frac{\text{Odds}}{1 + \text{Odds}}.$$

We combine this with the logistic response function, shown earlier, to get:

$$\text{Odds}(Y = 1) = e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_q x_q}.$$

Finally, taking the logarithm of both sides, we get an expression that involves a linear function of the predictors:

$$\log\left(\text{Odds}(Y = 1)\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_q x_q.$$

The *log-odds* function, also known as the *logit* function, maps the probability p from $(0, 1)$ to any value $(-\infty, +\infty)$: see Figure 5-2. The transformation circle is complete; we have used a linear model to predict a probability, which, in turn, we can map to a class label by applying a cutoff rule — any record with a probability greater than the cutoff is classified as a 1.

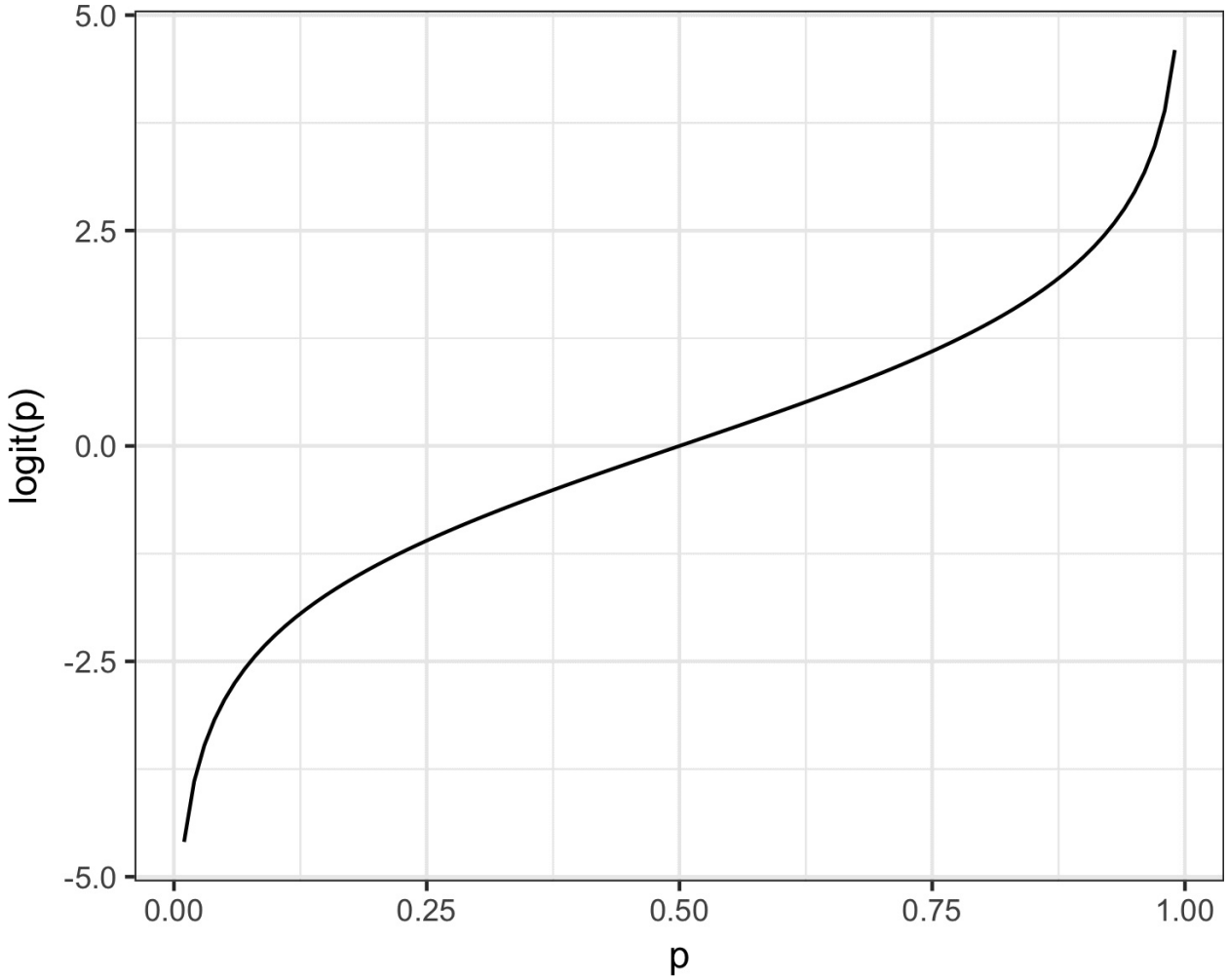

Figure 5-2. The function that maps a probability to a scale suitable for a linear model (logit)

Logistic Regression and the GLM

The response in the logistic regression formula is the log odds of a binary outcome of 1. We only observe the binary outcome, not the log odds, so special statistical methods are needed to fit the equation. Logistic regression is a special instance of a *generalized linear model* (GLM) developed to extend linear regression to other settings.

In R, to fit a logistic regression, the `glm` function is used with the `family` parameter set to `binomial`. The following code fits a logistic regression to the personal loan data introduced in "K-Nearest Neighbors".

```
logistic_model

Call:  glm(formula = outcome ~ payment_inc_ratio + purpose_ + home_ +
    emp_len_ + borrower_score, family = "binomial", data = loan_data)

Coefficients:
                (Intercept)           payment_inc_ratio
                    1.26982                     0.08244
purpose_debt_consolidation    purpose_home_improvement
                    0.25216                     0.34367
    purpose_major_purchase             purpose_medical
                    0.24373                     0.67536
             purpose_other      purpose_small_business
                    0.59268                     1.21226
                  home_OWN                   home_RENT
                    0.03132                     0.16867
         emp_len_ < 1 Year              borrower_score
                    0.44489                    -4.63890

Degrees of Freedom: 46271 Total (i.e. Null);  46260 Residual
Null Deviance:      64150
Residual Deviance: 58530        AIC: 58550
```

The response is `outcome`, which takes a 0 if the loan is paid off and 1 if the loan defaults. `purpose_` and `home_` are factor variables representing the purpose of the loan and the home ownership status. As in regression, a factor variable with P levels is represented with $P - 1$ columns. By default in R, the *reference* coding is used and the levels are all compared to the reference level (see "Factor Variables in Regression"). The reference levels for these factors are `credit_card` and `MORTGAGE`, respectively. The variable `borrower_score` is a score from 0 to 1 representing the creditworthiness of the borrower (from poor to excellent). This variable was created from several other variables using K-Nearest Neighbor: see "KNN as a Feature Engine".

Generalized Linear Models

Generalized linear models (GLMs) are the second most important class of models besides regression. GLMs are characterized by two main components:

- A probability distribution or family (binomial in the case of logistic regression)
- A link function mapping the response to the predictors (logit in the case of logistic regression)

Logistic regression is by far the most common form of GLM. A data scientist will encounter other types of GLMs. Sometimes a log link function is used instead of the logit; in practice, use of a log link is unlikely to lead to very different results for most applications. The poisson distribution is commonly used to model count data (e.g., the number of times a user visits a web page in a certain amount of time). Other families include negative binomial and gamma, often used to model elapsed time (e.g., time to failure). In contrast to logistic regression, application of GLMs with these models is more nuanced and involves greater care. These are best avoided unless you are familiar with and understand the utility and pitfalls of these methods.

Predicted Values from Logistic Regression

The predicted value from logistic regression is in terms of the log odds: $\hat{Y} = \log(\text{Odds}(Y = 1))$. The predicted probability is given by the logistic response function:

$$\hat{p} = \frac{1}{1 + e^{-\hat{Y}}}$$

For example, look at the predictions from the model `logistic_model`:

```
pred <- predict(logistic_model)
summary(pred)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-2.728000 -0.525100 -0.005235  0.002599  0.513700  3.658000
```

Converting these values to probabilities is a simple transform:

```
prob <- 1/(1 + exp(-pred))
> summary(prob)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.06132 0.37170 0.49870 0.50000 0.62570 0.97490
```

These are on a scale from 0 to 1 and don't yet declare whether the predicted value is default or paid off. We could declare any value greater than 0.5 as default, analogous to the *K*-Nearest Neighbors classifier. In practice, a lower cutoff is often appropriate if the goal is to identify members of a rare class (see "The Rare Class Problem").

Interpreting the Coefficients and Odds Ratios

One advantage of logistic regression is that it produces a model that can be scored to new data rapidly, without recomputation. Another is the relative ease of interpretation of the model, as compared with other classification methods. The key conceptual idea is understanding an *odds ratio*. The odds ratio is easiest to understand for a binary factor variable X:

$$\text{odds ratio} = \frac{\text{Odds}(Y = 1 \mid X = 1)}{\text{Odds}(Y = 1 \mid X = 0)}$$

This is interpreted as the odds that $Y = 1$ when $X = 1$ versus the odds that $Y = 1$ when $X = 0$. If the odds ratio is 2, then the odds that $Y = 1$ are two times higher when $X = 1$ versus $X = 0$.

Why bother with an odds ratio, instead of probabilities? We work with odds because the coefficient β_j in the logistic regression is the log of the odds ratio for X_j.

An example will make this more explicit. For the model fit in "Logistic Regression and the GLM", the regression coefficient for `purpose_small_business` is 1.21226. This means that a loan to a small business compared to a loan to pay off credit card debt reduces the odds of defaulting versus being paid off by $exp(121226) \approx 34$. Clearly, loans for the purpose of creating or expanding a small business are considerably riskier than other types of loans.

Figure 5-3 shows the relationship between the odds ratio and log-odds ratio for odds ratios greater than 1. Because the coefficients are on the log scale, an increase of 1 in the coefficient results in an increase of $exp(1) \approx 272$ in the odds ratio.

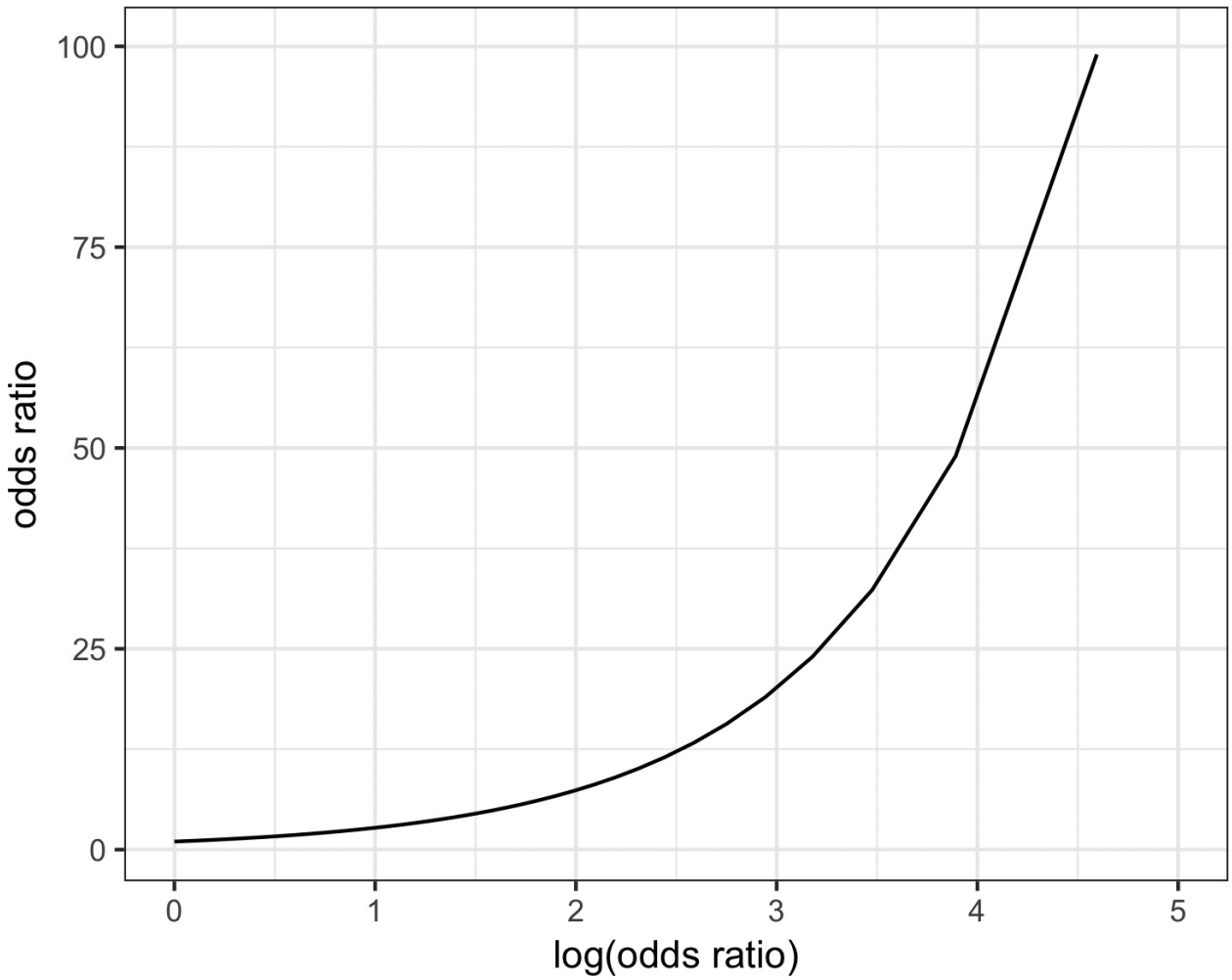

Figure 5-3. The relationship between the odds ratio and the log-odds ratio

Odds ratios for numeric variables X can be interpreted similarly: they measure the change in the odds ratio for a unit change in X. For example, the effect of increasing the payment to income ratio from, say, 5 to 6 increases the odds of the loan defaulting by a factor of $exp(0.08244) \approx 1.09$. The variable `borrower_score` is a score on the borrowers' creditworthiness and ranges from 0 (low) to 1 (high). The odds of the best borrowers relative to the worst borrowers defaulting on their loans is smaller by a factor of $exp(-4.63890) \approx 0.01$. In other words, the default risk from the borrowers with the poorest creditworthiness is 100 times greater than that of the best borrowers!

Linear and Logistic Regression: Similarities and Differences

Multiple linear regression and logistic regression share many commonalities. Both assume a parametric linear form relating the predictors with the response. Exploring and finding the best model are done in very similar ways. Generalities to the linear model to use a spline transform of the predictor are equally applicable in the logistic regression setting. Logistic regression differs in two fundamental ways:

- The way the model is fit (least squares is not applicable)
- The nature and analysis of the residuals from the model

Fitting the model

Linear regression is fit using least squares, and the quality of the fit is evaluated using RMSE and R-squared statistics. In logistic regression (unlike in linear regression), there is no closed-form solution and the model must be fit using *maximum likelihood estimation* (MLE). Maximum likelihood estimation is a process that tries to find the model that is most likely to have produced the data we see. In the logistic regression equation, the response is not 0 or 1 but rather an estimate of the log odds that the response is 1. The MLE finds the solution such that the estimated log odds best describes the observed outcome. The mechanics of the algorithm involve a quasi-Newton optimization that iterates between a scoring step (*Fisher's scoring*), based on the current parameters, and an update to the parameters to improve the fit.

MAXIMUM LIKELIHOOD ESTIMATION

More detail, if you like statistical symbols: start with a set of data $(X_1, X_2, \cdots, X_n)$ and a probability model $\mathcal{P}_\theta(X_1, X_2, \cdots, X_n)$ that depends on a set of parameters θ. The goal of MLE is to find the set of parameters $\hat{\theta}$ that maximizes the value of $\mathcal{P}_\theta(X_1, X_2, \cdots, X_n)$; that is, it maximizes the probability of observing $(X_1, X_2, \cdots, X_n)$ given the model $\mathcal{P}$..] In the fitting process, the model is evaluated using a metric called *deviance*:

$$\text{deviance} = -2 \log \left(\mathcal{P}_{\hat{\theta}}(\mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_n) \right)$$

Lower deviance corresponds to a better fit.

Fortunately, most users don’t need to concern themselves with the details of the fitting algorithm since this is handled by the software. Most data scientists will not need to worry about the fitting method, other than understanding that it is a way to find a good model under certain assumptions.

HANDLING FACTOR VARIABLES

In logistic regression, factor variables should be coded as in linear regression; see "Factor Variables in Regression". In R and other software, this is normally handled automatically and generally reference encoding is used. All of the other classification methods covered in this chapter typically use the one hot encoder representation (see "One Hot Encoder").

Assessing the Model

Like other classification methods, logistic regression is assessed by how accurately the model classifies new data (see "Evaluating Classification Models"). As with linear regression, some additional standard statistical tools are available to assess and improve the model. Along with the estimated coefficients, R reports the standard error of the coefficients (SE), a *z*-value, and a p-value:

```
summary(logistic_model)

Call:
glm(formula = outcome ~ payment_inc_ratio + purpose_ + home_ +
    emp_len_ + borrower_score, family = "binomial", data = loan_data)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.71430  -1.06806  -0.04482   1.07446   2.11672

Coefficients:
                           Estimate Std. Error z value Pr(>|z|)
(Intercept)                1.269822   0.051929  24.453  < 2e-16 ***
payment_inc_ratio          0.082443   0.002485  33.177  < 2e-16 ***
purpose_debt_consolidation 0.252164   0.027409   9.200  < 2e-16 ***
purpose_home_improvement   0.343674   0.045951   7.479 7.48e-14 ***
purpose_major_purchase     0.243728   0.053314   4.572 4.84e-06 ***
purpose_medical            0.675362   0.089803   7.520 5.46e-14 ***
purpose_other              0.592678   0.039109  15.154  < 2e-16 ***
purpose_small_business     1.212264   0.062457  19.410  < 2e-16 ***
home_OWN                   0.031320   0.037479   0.836    0.403
home_RENT                  0.168670   0.021041   8.016 1.09e-15 ***
emp_len_ < 1 Year          0.444892   0.053342   8.340  < 2e-16 ***
borrower_score            -4.638902   0.082433 -56.275  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 64147  on 46271  degrees of freedom
Residual deviance: 58531  on 46260  degrees of freedom
AIC: 58555

Number of Fisher Scoring iterations: 4
```

Interpretation of the p-value comes with the same caveat as in regression, and should be viewed more as a relative indicator of variable importance (see "Assessing the Model") than as a formal measure of statistical significance. A logistic regression model, which has a binary response, does not have an associated RMSE or R-squared. Instead, a logistic regression model is typically evaluated using more general metrics for classification; see "Evaluating Classification Models".

Many other concepts for linear regression carry over to the logistic regression setting (and other GLMs). For example, you can use stepwise regression, fit interaction terms, or include spline terms. The same concerns regarding confounding and correlated variables apply to logistic regression (see "Interpreting the Regression Equation"). You can fit generalized additive models (see "Generalized Additive Models") using the `mgcv` package:

```
logistic_gam <- gam(outcome ~ s(payment_inc_ratio) + purpose_ +
                        home_ + emp_len_ + s(borrower_score),
                    data=loan_data, family='binomial')
```

One area where logistic regression differs is in the analysis of the residuals. As in regression (see Figure 4-9), it is straightforward to compute partial residuals:

```
terms <- predict(logistic_gam, type='terms')
partial_resid <- resid(logistic_model) + terms
df <- data.frame(payment_inc_ratio = loan_data[, 'payment_inc_ratio'],
                 terms = terms[, 's(payment_inc_ratio)'],
                 partial_resid = partial_resid[, 's(payment_inc_ratio)'])
ggplot(df, aes(x=payment_inc_ratio, y=partial_resid, solid = FALSE)) +
  geom_point(shape=46, alpha=.4) +
  geom_line(aes(x=payment_inc_ratio, y=terms),
            color='red', alpha=.5, size=1.5) +
  labs(y='Partial Residual')
```

The resulting plot is displayed in Figure 5-4. The estimated fit, shown by the line, goes between two sets of point clouds. The top cloud corresponds to a response of 1 (defaulted loans), and the bottom cloud corresponds to a response of 0 (loans paid off). This is very typical of residuals from a logistic regression since the output is binary. Partial residuals in logistic regression, while less valuable than in regression, are still useful to confirm nonlinear behavior and identify highly influential records.

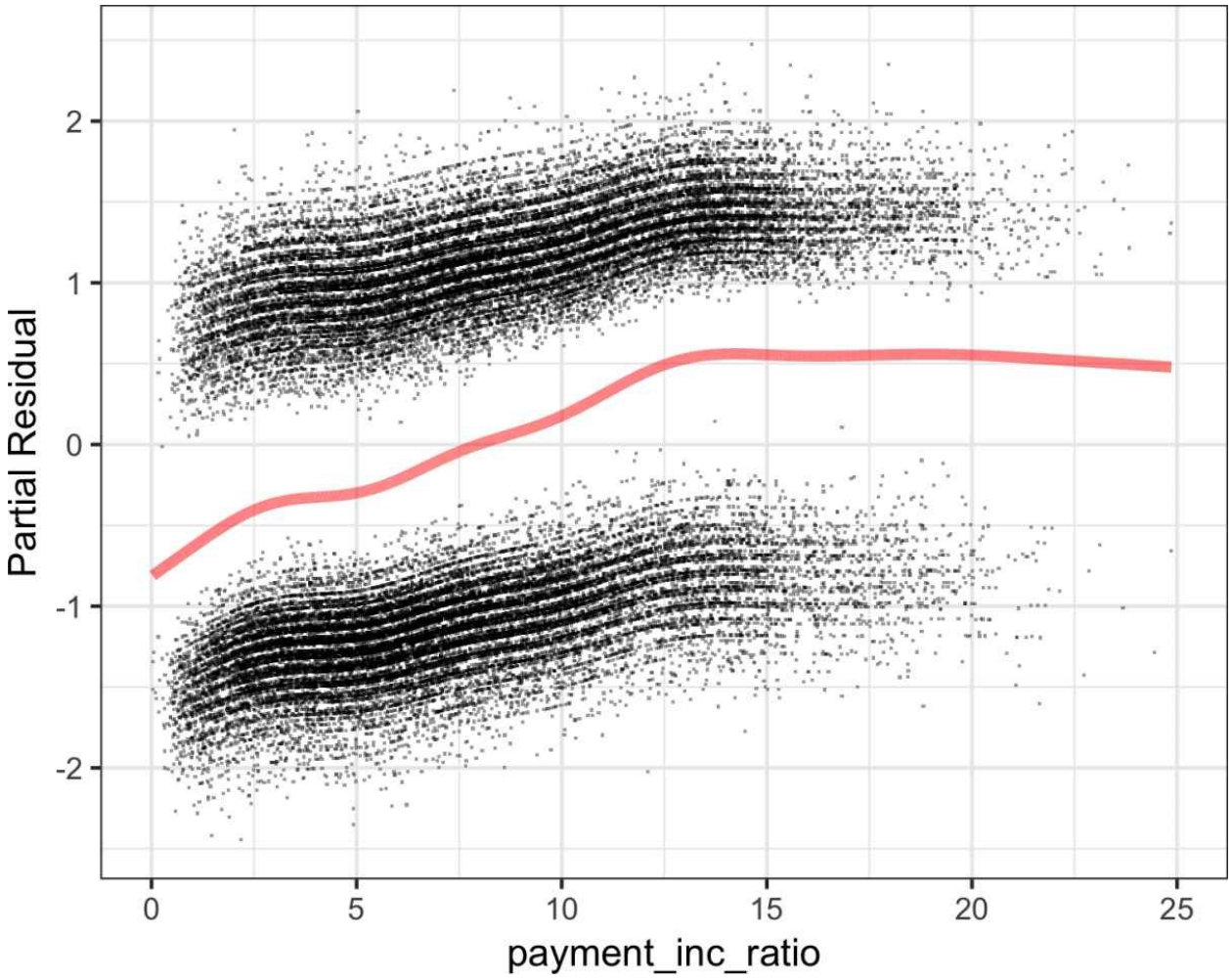

Figure 5-4. Partial residuals from logistic regression

> **WARNING**
>
> Some of the output from the `summary` function can effectively be ignored. The dispersion parameter does not apply to logistic regression and is there for other types of GLMs. The residual deviance and the number of scoring iterations are related to the maximum likelihood fitting method; see "Maximum Likelihood Estimation".

KEY IDEAS FOR LOGISTIC REGRESSION

- Logistic regression is like linear regression, except that the outcome is a binary variable.
- Several transformations are needed to get the model into a form that can be fit as a linear model, with the log of the odds ratio as the response variable.

- After the linear model is fit (by an iterative process), the log odds is mapped back to a probability.
- Logistic regression is popular because it is computationally fast, and produces a model that can be scored to new data without recomputation.

Further Reading

1. The standard reference on logistic regression is *Applied Logistic Regression*, 3rd ed., by David Hosmer, Stanley Lemeshow, and Rodney Sturdivant (Wiley).

2. Also popular are two books by Joseph Hilbe: *Logistic Regression Models* (very comprehensive) and *Practical Guide to Logistic Regression* (compact), both from CRC Press.

3. *Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, Jerome Freidman, and its shorter cousin, *An Introduction to Statistical Learning*, by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (both from Springer) both have a section on logistic regression.

4. *Data Mining for Business Analytics*, 3rd ed., by Galit Shmueli, Peter Bruce, and Nitin Patel (Wiley, 2016, with variants for R, Excel, and JMP) has a full chapter on logistic regression.

Evaluating Classification Models

It is common in predictive modeling to try out a number of different models, apply each to a holdout sample (also called a *test* or *validation sample*), and assess their performance. Fundamentally, this amounts to seeing which produces the most accurate predictions.

KEY TERMS FOR EVALUATING CLASSIFICATION MODELS

Accuracy
: The percent (or proportion) of cases classified correctly.

Confusion matrix
: A tabular display (2×2 in the binary case) of the record counts by their predicted and actual classification status.

Sensitivity
: The percent (or proportion) of 1s correctly classified.

Synonym
: Recall

Specificity
: The percent (or proportion) of 0s correctly classified.

Precision
: The percent (proportion) of predicted 1s that are actually 1s.

ROC curve
: A plot of sensitivity versus specificity.

Lift
: A measure of how effective the model is at identifying (comparitively rare) 1s at different probability cutoffs.

A simple way to measure classification performance is to count the proportion of predictions that are correct.

In most classification algorithms, each case is assigned an "estimated probability of being a 1."[3] The default decision point, or cutoff, is typically 0.50 or 50%. If the probability is above 0.5, the classification is "1," otherwise it is "0." An alternative default cutoff is the prevalent probability of 1s in the data.

Accuracy is simply a measure of total error:

$$\text{accuracy} = \frac{\sum \text{TruePositive} + \sum \text{TrueNegative}}{\text{SampleSize}}$$

Confusion Matrix

At the heart of classification metrics is the *confusion matrix*. The confusion matrix is a table showing the number of correct and incorrect predictions categorized by type of response. Several packages are available in R to compute a confusion matrix, but in the binary case, it is simple to compute one by hand.

To illustrate the confusion matrix, consider the `logistic_gam` model that was trained on a balanced data set with an equal number of defaulted and paid-off loans (see Figure 5-4). Following the usual conventions $Y = 1$ corresponds to the event of interest (e.g., default) and $Y = 0$ corresponds to a negative (or usual) event (e.g., paid off). The following computes the confusion matrix for the `logistic_gam` model applied to the entire (unbalanced) training set:

```
pred <- predict(logistic_gam, newdata=train_set)
pred_y <- as.numeric(pred > 0)
true_y <- as.numeric(train_set$outcome=='default')
true_pos <- (true_y==1) & (pred_y==1)
true_neg <- (true_y==0) & (pred_y==0)
false_pos <- (true_y==0) & (pred_y==1)
false_neg <- (true_y==1) & (pred_y==0)
conf_mat <- matrix(c(sum(true_pos), sum(false_pos),
                     sum(false_neg), sum(true_neg)), 2, 2)
colnames(conf_mat) <- c('Yhat = 1', 'Yhat = 0')
rownames(conf_mat) <- c('Y = 1', 'Y = 0')
conf_mat
      Yhat = 1 Yhat = 0
Y = 1 14635    8501
Y = 0 8236     14900
```

The predicted outcomes are columns and the true outcomes are the rows. The diagonal elements of the matrix show the number of correct predictions and the off-diagonal elements show the number of incorrect predictions. For example, 6,126 defaulted loans were correctly predicted as a default, but 17,010 defaulted loans were incorrectly predicted as paid off.

Figure 5-5 shows the relationship between the confusion matrix for a binary reponse Y and different metrics (see "Precision, Recall, and Specificity" for more on the metrics). As with the example for the loan data, the actual response is along the rows and the predicted response is along the columns. (You may see confusion matrices with this reversed.) The diagonal boxes (upper left, lower right) show when the predictions $\hat{Y}$ correctly predict the response. One important metric not

explicitly called out is the false positive *rate* (the mirror image of precision). When 1s are rare, the ratio of false positives to all predicted positives can be high, leading to the unintuitive situation where a predicted 1 is most likely a 0. This problem plagues medical screening tests (e.g., mammograms) that are widely applied: due to the relative rarity of the condition, positive test results most likely do not mean breast cancer. This leads to much confusion in the public.

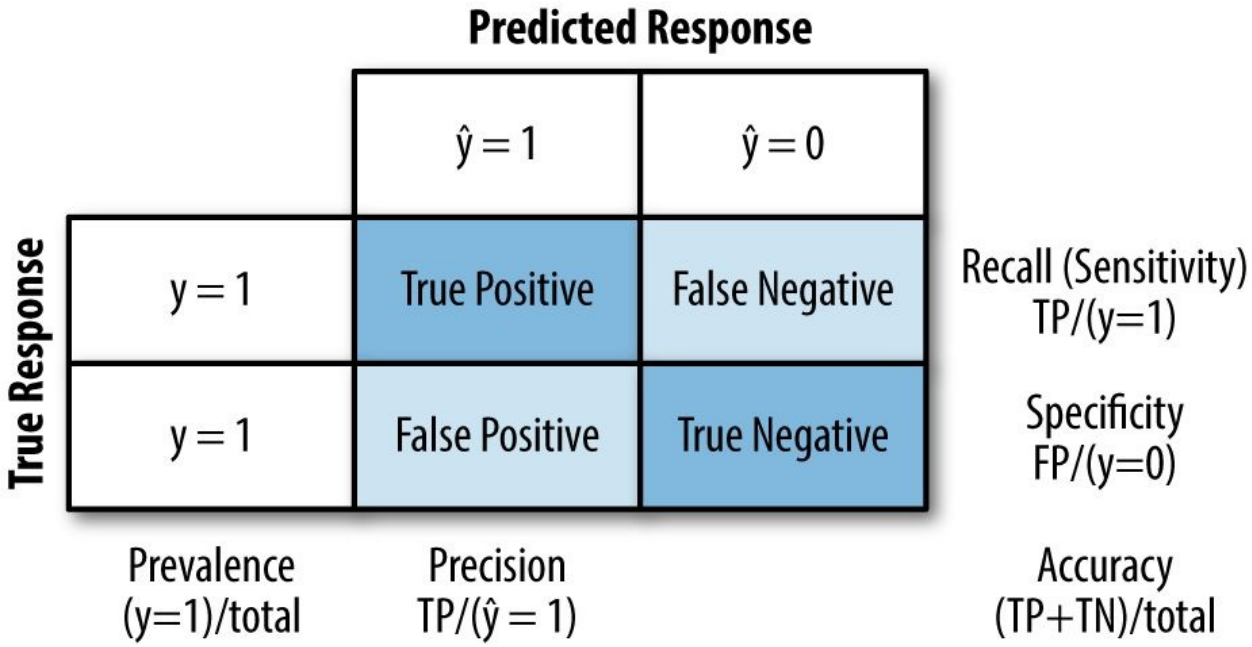

Figure 5-5. Confusion matrix for a binary response and various metrics

The Rare Class Problem

In many cases, there is an imbalance in the classes to be predicted, with one class much more prevalent than the other — for example, legitimate insurance claims versus fraudulent ones, or browsers versus purchasers at a website. The rare class (e.g., the fraudulent claims) is usually the class of more interest, and is typically designated 1, in contrast to the more prevalent 0s. In the typical scenario, the 1s are the more important case, in the sense that misclassifying them as 0s is costlier than misclassfying 0s as 1s. For example, correctly identifying a fraudulent insurance claim may save thousands of dollars. On the other hand, correctly identifying a nonfraudulent claim merely saves you the cost and effort of going through the claim by hand with a more careful review (which is what you would do if the claim were tagged as “fraudulent”).

In such cases, unless the classes are easily separable, the most accurate classification model may be one that simply classifies everything as a 0. For example, if only 0.1% of the browsers at a web store end up purchasing, a model that predicts that each browser will leave without purchasing will be 99.9% accurate. However, it will be useless. Instead, we would be happy with a model that is less accurate overall, but is good at picking out the purchasers, even if it misclassifies some nonpurchasers along the way.

Precision, Recall, and Specificity

Metrics other than pure accuracy — metrics that are more nuanced — are commonly used in evaluating classification models. Several of these have a long history in statistics — especially biostatistics, where they are used to describe the expected performance of diagnostic tests. The *precision* measures the accuracy of a predicted positive outcome (see Figure 5-5):

$$\text{precision} = \frac{\sum \text{TruePositive}}{\sum \text{TruePositive} + \sum \text{FalsePositive}}$$

The *recall*, also known as *sensitivity*, measures the strength of the model to predict a positive outcome — the proportion of the 1s that it correctly identifies (see Figure 5-5). The term *sensitivity* is used a lot in biostatistics and medical diagnostics, whereas *recall* is used more in the machine learning community. The definition of recall is:

$$\text{recall} = \frac{\sum \text{TruePositive}}{\sum \text{TruePositive} + \sum \text{FalseNegative}}$$

Another metric used is *specificity*, which measures a model's ability to predict a negative outcome:

$$\text{specificity} = \frac{\sum \text{TrueNegative}}{\sum \text{TrueNegative} + \sum \text{FalseNegative}}$$

```
# precision
conf_mat[1,1]/sum(conf_mat[,1])
# recall
conf_mat[1,1]/sum(conf_mat[1,])
# specificity
conf_mat[2,2]/sum(conf_mat[2,])
```

ROC Curve

You can see that there is a tradeoff between recall and specificity. Capturing more 1s generally means misclassifying more 0s as 1s. The ideal classifier would do an excellent job of classifying the 1s, without misclassifying more 0s as 1s.

The metric that captures this tradeoff is the "Receiver Operating Characteristics" curve, usually referred to as the *ROC curve*. The ROC curve plots recall (sensitivity) on the y-axis against specificity on the x-axis.[4] The ROC curve shows the trade-off between recall and specificity as you change the cutoff to determine how to classify a record. Sensitivity (recall) is plotted on the y-axis, and you may encounter two forms in which the x-axis is labeled:

- Specificity plotted on the x-axis, with 1 on the left and 0 on the right
- Specificity plotted on the x-axis, with 0 on the left and 1 on the right

The curve looks identical whichever way it is done. The process to compute the ROC curve is:

1. Sort the records by the predicted probability of being a 1, starting with the most probable and ending with the least probable.
2. Compute the cumulative specificity and recall based on the sorted records.

Computing the ROC curve in R is straightforward. The following code computes ROC for the loan data:

```
idx <- order(-pred)
recall <- cumsum(true_y[idx]==1)/sum(true_y==1)
specificity <- (sum(true_y==0) - cumsum(true_y[idx]==0))/sum(true_y==0)
roc_df <- data.frame(recall = recall, specificity = specificity)
ggplot(roc_df, aes(x=specificity, y=recall)) +
  geom_line(color='blue') +
  scale_x_reverse(expand=c(0, 0)) +
  scale_y_continuous(expand=c(0, 0)) +
  geom_line(data=data.frame(x=(0:100)/100), aes(x=x, y=1-x),
            linetype='dotted', color='red')
```

The result is shown in Figure 5-6. The dotted diagonal line corresponds to a classifier no better than random chance. An extremely effective classifier (or, in medical situations, an extremely effective diagnostic test) will have an ROC that

hugs the upper-left corner — it will correctly identify lots of 1s without misclassifying lots of 0s as 1s. For this model, if we want a classifier with a specificity of at least 50%, then the recall is about 75%.

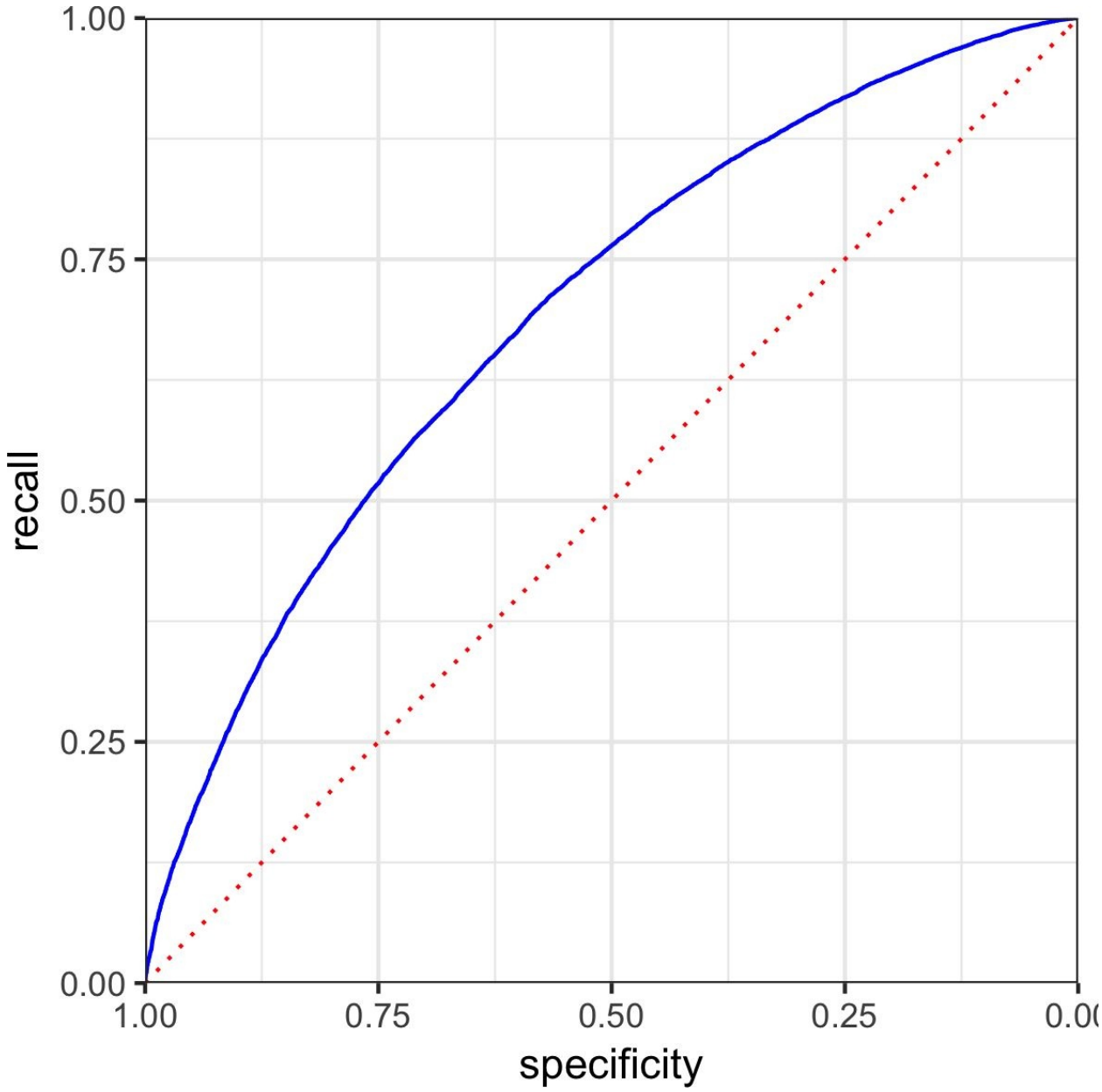

Figure 5-6. ROC curve for the loan data

PRECISION-RECALL CURVE

In addition to ROC curves, it can be illuminating to examine the precision-recall (PR) curve. PR curves are computed in a similar way except that the data is ordered from least to most probable and cumulative precision and recall statistics are computed. PR curves are especially useful in evaluating data with highly unbalanced outcomes.

AUC

The ROC curve is a valuable graphical tool but, by itself, doesn't constitute a single measure for the performance of a classifier. The ROC curve can be used, however, to produce the area underneath the curve (AUC) metric. AUC is simply the total area under the ROC curve. The larger the value of AUC, the more effective the classifier. An AUC of 1 indicates a perfect classifier: it gets all the 1s correctly classified, and doesn't misclassify any 0s as 1s.

A completely ineffective classifier — the diagonal line — will have an AUC of 0.5.

Figure 5-7 shows the area under the ROC curve for the loan model. The value of AUC can be computed by a numerical integration:

```
sum(roc_df$recall[-1] * diff(1-roc_df$specificity))
[1] 0.5924072
```

The model has an AUC of about 0.59, corresponding to a relatively weak classifier.

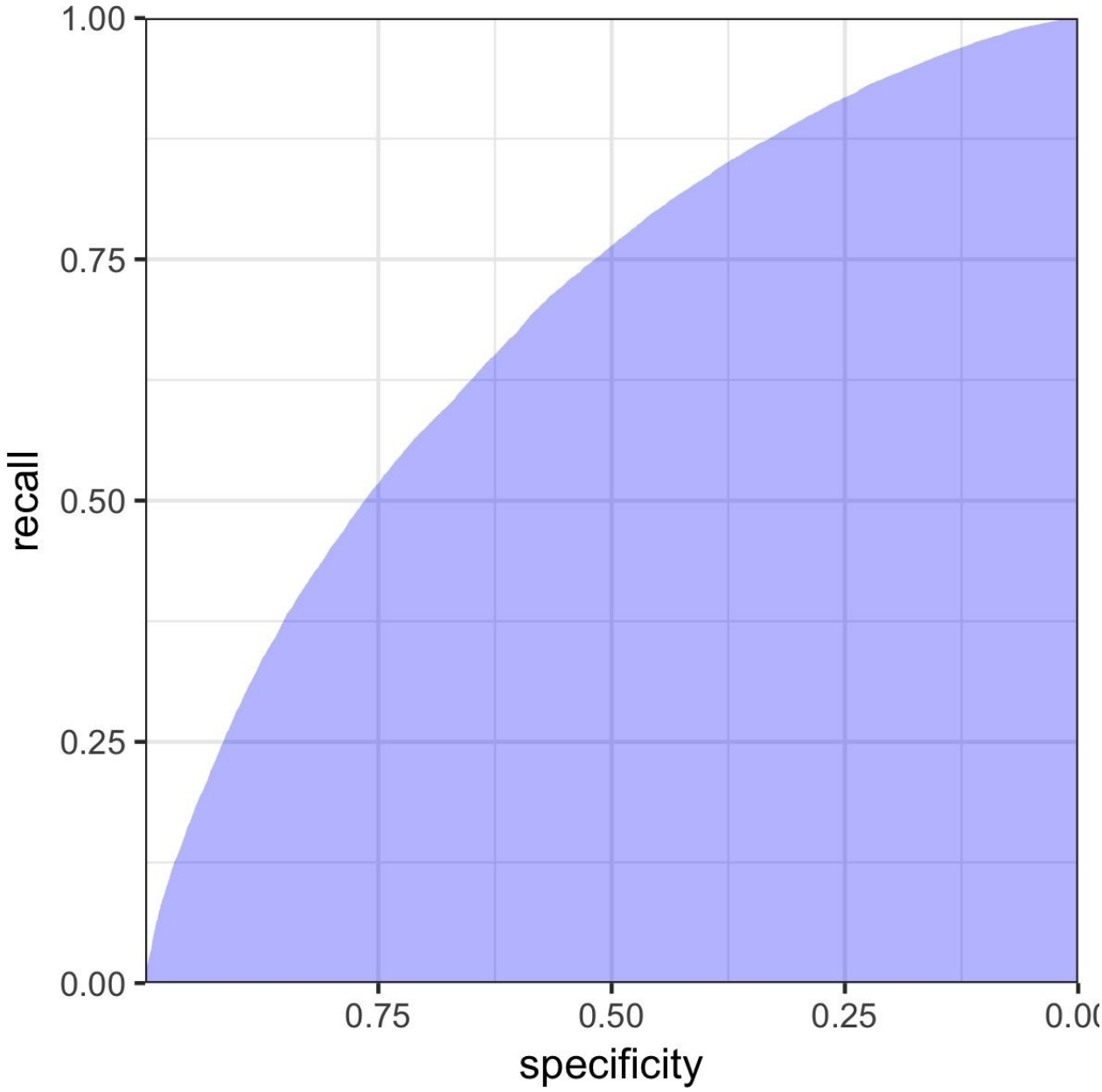

Figure 5-7. Area under the ROC curve for the loan data

FALSE POSITIVE RATE CONFUSION

False positive/negative rates are often confused or conflated with specificity or sensitivity (even in publications and software!). Sometimes the false positive rate is defined as the proportion of true negatives that test positive. In many cases (such as network intrusion detection), the term is used to refer to the proportion of positive signals that are true negatives.

Lift

Using the AUC as a metric is an improvement over simple accuracy, as it can assess how well a classifier handles the tradeoff between overall accuracy and the need to identify the more important 1s. But it does not completely address the rare-case problem, where you need to lower the model's probability cutoff below 0.5 to avoid having all records classified as 0. In such cases, for a record to be classified as a 1, it might be sufficient to have a probability of 0.4, 0.3, or lower. In effect, we end up overidentifying 1s, reflecting their greater importance.

Changing this cutoff will improve your chances of catching the 1s (at the cost of misclassifying more 0s as 1s). But what is the optimum cutoff?

The concept of lift lets you defer answering that question. Instead, you consider the records in order of their predicted probability of being 1s. Say, of the top 10% classified as 1s, how much better did the algorithm do, compared to the benchmark of simply picking blindly? If you can get 0.3% response in this top decile instead of the 0.1% you get overall picking randomly, the algorithm is said to have a *lift* (also called *gains*) of 3 in the top decile. A lift chart (gains chart) quantifies this over the range of the data. It can be produced decile by decile, or continuously over the range of the data.

To compute a lift chart, you first produce a *cumulative gains chart* that shows the recall on the y-axis and the total number of records on the x-axis. The *lift curve* is the ratio of the cumulative gains to the diagonal line corresponding to random selection. *Decile gains charts* are one of the oldest techniques in predictive modeling, dating from the days before internet commerce. They were particularly popular among direct mail professionals. Direct mail is an expensive method of advertising if applied indiscriminantly, and advertisers used predictive models (quite simple ones, in the early days) to identify the potential customers with the likeliest prospect of payoff.

UPLIFT

Sometimes the term *uplift* is used to mean the same thing as lift. An alternate meaning is used in a more restrictive setting, when an A-B test has been conducted and the treatment (A or B) is then used as a predictor variable in a predictive model. The uplift is the improvement in response predicted *for an individual case* with treatment A versus treatment B. This is determined by scoring the individual case first with the predictor set to A, and then again with the predictor toggled to B. Marketers and political campaign consultants use this method to determine which of two messaging treatments should be used with which customers or voters.

A lift curve lets you look at the consequences of setting different probability cutoffs for classifying records as 1s. It can be an intermediate step in settling on an appropriate cutoff level. For example, a tax authority might only have a certain amount of resources that it can spend on tax audits, and want to spend them on the likeliest tax cheats. With its resource constraint in mind, the authority would use a lift chart to estimate where to draw the line between tax returns selected for audit and those left alone.

KEY IDEAS FOR EVALUATING CLASSIFICATION MODELS

- Accuracy (the percent of predicted classifications that are correct) is but a first step in evaluating a model.
- Other metrics (recall, specificity, precision) focus on more specific performance characteristics (e.g., recall measures how good a model is at correctly identifying 1s).
- AUC (area under the ROC curve) is a common metric for the ability of a model to distinguish 1s from 0s.
- Similarly, lift measures how effective a model is in identifying the 1s, and it is often calculated decile by decile, starting with the most probable 1s.

Further Reading

Evaluation and assessment are typically covered in the context of a particular model (e.g., *K*-Nearest Neighbors or decision trees); three books that handle it in its own chapter are:

- *Data Mining*, 3rd ed., by Ian Whitten, Elbe Frank, and Mark Hall (Morgan Kaufmann, 2011).
- *Modern Data Science with R* by Benjamin Baumer, Daniel Kaplan, and Nicholas Horton (CRC Press, 2017).
- *Data Mining for Business Analytics*, 3rd ed., by Galit Shmueli, Peter Bruce, and Nitin Patel (Wiley, 2016, with variants for R, Excel, and JMP).

An excellent treatment of cross-validation and resampling can be found in:

- *An Introduction to Statistical Learning* by Gareth James, et al. (Springer, 2013).

Strategies for Imbalanced Data

The previous section dealt with evaluation of classification models using metrics that go beyond simple accuracy, and are suitable for imbalanced data — data in which the outcome of interest (purchase on a website, insurance fraud, etc.) is rare. In this section, we look at additional strategies that can improve predictive modeling performance with imbalanced data.

KEY TERMS FOR IMBALANCED DATA

Undersample

Use fewer of the prevalent class records in the classification model.

Synonym

Downsample

Oversample

Use more of the rare class records in the classification model, bootstrapping if necessary.

Synonym

Upsample

Up weight or down weight

Attach more (or less) weight to the rare (or prevalent) class in the model.

Data generation

Like bootstrapping, except each new bootstrapped record is slightly different from its source.

Z-score

The value that results after standardization.

K

The number of neighbors considered in the nearest neighbor calculation.

Undersampling

If you have enough data, as is the case with the loan data, one solution is to *undersample* (or downsample) the prevalent class, so the data to be modeled is more balanced between 0s and 1s. The basic idea in undersampling is that the data for the dominant class has many redundant records. Dealing with a smaller, more balanced data set yields benefits in model performance, and makes it easier to prepare the data, and to explore and pilot models.

How much data is enough? It depends on the application, but in general, having tens of thousands of records for the less dominant class is enough. The more easily distinguishable the 1s are from the 0s, the less data needed.

The loan data analyzed in "Logistic Regression" was based on a balanced training set: half of the loans were paid off and the other half were in default. The predicted values were similar: half of the probabilities were less than 0.5 and half were greater than 0.5. In the full data set, only about 5% of the loans were in default:

```
mean(loan_all_data$outcome == 'default')
[1] 0.05024048
```

What happens if we use the full data set to train the model?

```
full_model <- glm(outcome ~ payment_inc_ratio + purpose_ +
                        home_ + emp_len_+ dti + revol_bal + revol_util,
                      data=train_set, family='binomial')
pred <- predict(full_model)
mean(pred > 0)
[1] 0.00386009
```

Only 0.39% of the loans are predicted to be in default, or less than 1/12 of the expected number. The loans that were paid off overwhelm the loans in default because the model is trained using all the data equally. Thinking about it intuitively, the presence of so many nondefaulting loans, coupled with the inevitable variability in predictor data, means that, even for a defaulting loan, the model is likely to find some nondefaulting loans that it is similar to, by chance. When a balanced sample was used, roughly 50% of the loans were predicted to be in default.

Oversampling and Up/Down Weighting

One criticism of the undersampling method is that it throws away data and is not using all the information at hand. If you have a relatively small data set, and the rarer class contains a few hundred or a few thousand records, then undersampling the dominant class has the risk of throwing out useful information. In this case, instead of downsampling the dominant case, you should oversample (upsample) the rarer class by drawing additional rows with replacement (bootstrapping).

You can achieve a similar effect by weighting the data. Many classification algorithms take a weight argument that will allow you to up/down weight the data. For example, apply a weight vector to the loan data using the `weight` argument to `glm`:

```
wt <- ifelse(loan_all_data$outcome=='default',
             1/mean(loan_all_data$outcome == 'default'), 1)
full_model <- glm(outcome ~ payment_inc_ratio + purpose_ +
                    home_ + emp_len_+ dti + revol_bal + revol_util,
                  data=loan_all_data, weight=wt, family='binomial')
pred <- predict(full_model)
mean(pred > 0)
[1] 0.4344177
```

The weights for loans that default are set to $\frac{1}{p}$ where p is the probability of default. The nondefaulting loans have a weight of 1. The sum of the weights for the defaulted loans and nondefaulted loans are roughly equal. The mean of the predicted values is now 43% instead of 0.39%.

Note that weighting provides an alternative to both upsampling the rarer class and downsampling the dominant class.

ADAPTING THE LOSS FUNCTION

Many classification and regression algorithms optimize a certain criteria or *loss function*. For example, logistic regression attempts to minimize the deviance. In the literature, some propose to modify the loss function in order to avoid the problems caused by a rare class. In practice, this is hard to do: classification algorithms can be complex and difficult to modify. Weighting is an easy way to change the loss function, discounting errors for records with low weights in favor of records of higher weights.

Data Generation

A variation of upsampling via bootstrapping (see "Undersampling") is *data generation* by perturbing existing records to create new records. The intuition behind this idea is that since we only observe a limited set of instances, the algorithm doesn't have a rich set of information to build classification "rules." By creating new records that are similar but not identical to existing records, the algorithm has a chance to learn a more robust set of rules. This notion is similar in spirit to ensemble statistical models such as boosting and bagging (see Chapter 6).

The idea gained traction with the publication of the *SMOTE* algorithm, which stands for "Synthetic Minority Oversampling Technique." The SMOTE algorithm finds a record that is similar to the record being upsampled (see "K-Nearest Neighbors") and creates a synthetic record that is a randomly weighted average of the original record and the neighboring record, where the weight is generated separately for each predictor. The number of synthetic oversampled records created depends on the oversampling ratio required to bring the data set into approximate balance, with respect to outcome classes.

There are several implementations of SMOTE in R. The most comprehensive package for handling unbalanced data is `unbalanced`. It offers a variety of techniques, including a "Racing" algorithm to select the best method. However, the SMOTE algorithm is simple enough that it can be implemented directly in R using the `knn` package.

Cost-Based Classification

In practice, accuracy and AUC are a poor man's way to choose a classification rule. Often, an estimated cost can be assigned to false positives versus false negatives, and it is more appropriate to incorporate these costs to determine the best cutoff when classifying 1s and 0s. For example, suppose the expected cost of a default of a new loan is C and the expected return from a paid-off loan is R. Then the expected return for that loan is:

$$\text{expected return} = P(Y = 0) \times R + P(Y = 1) \times C$$

Instead of simply labeling a loan as default or paid off, or determining the probability of default, it makes more sense to determine if the loan has a positive expected return. Predicted probability of default is an intermediate step, and it must be combined with the loan's total value to determine expected profit, which is the ultimate planning metric of business. For example, a smaller value loan might be passed over in favor of a larger one with a slightly higher predicted default probability.

Exploring the Predictions

A single metric, such as AUC, cannot capture all aspects of the appropriateness of a model for a situation. Figure 5-8 displays the decision rules for four different models fit to the loan data using just two predictor variables: `borrower_score` and `payment_inc_ratio`. The models are linear discriminant analysis (LDA), logistic linear regression, logistic regression fit using a generalized additive model (GAM) and a tree model (see "Tree Models"). The region to the upper-left of the lines corresponds to a predicted default. LDA and logistic linear regression give nearly identical results in this case. The tree model produces the least regular rule: in fact, there are situations in which increasing the borrower score shifts the prediction from "paid-off" to "default"! Finally, the GAM fit of the logistic regression represents a compromise between the tree models and the linear models.

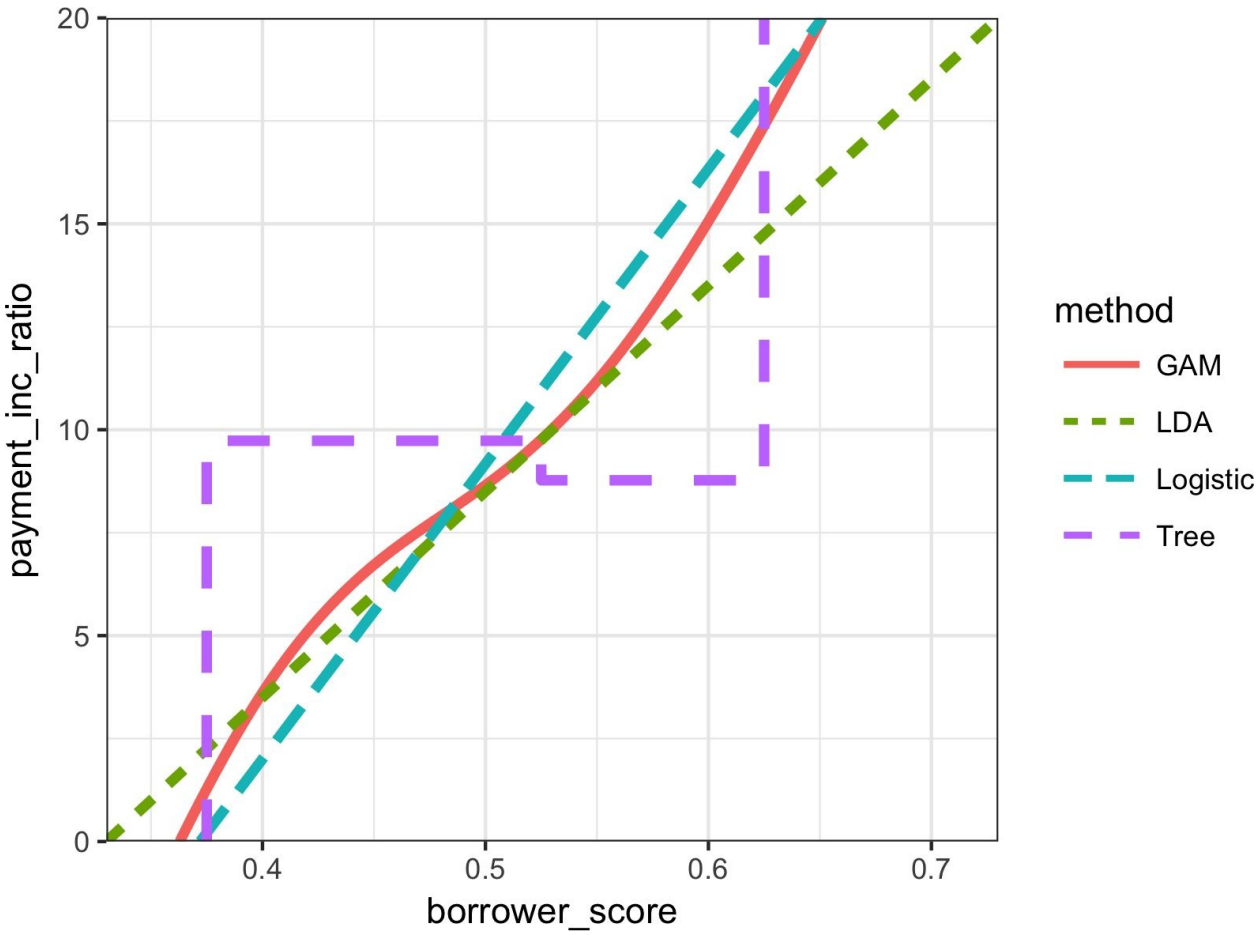

Figure 5-8. Comparison of the classification rules for four different methods

It is not easy to visualize the prediction rules in higher dimensions, or in the case of the GAM and tree model, even generate the regions for such rules.

In any case, exploratory analysis of predicted values is always warranted.

KEY IDEAS FOR IMBALANCED DATA STRATEGIES

- Highly imbalanced data (i.e., where the interesting outcomes, the 1s, are rare) are problematic for classification algorithms.
- One strategy is to balance the training data via undersampling the abundant case (or oversampling the rare case).
- If using all the 1s still leaves you with too few 1s, you can bootstrap the rare cases, or use SMOTE to create synthetic data similar to existing rare cases.
- Imbalanced data usually indicates that correctly classifying one class (the 1s) has higher value, and that value ratio should be built into the assessment metric.

Further Reading

- Tom Fawcett, author of *Data Science for Business*, has a good article on imbalanced classes.

- For more on SMOTE, see Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research* 16 (2002): 321–357.

- Also see the Analytics Vidya Content Team's "Practical Guide to deal with Imbalanced Classification Problems in R," March 28, 2016.

Summary

Classification, the process of predicting which of two (or a small number of) categories a record belongs to, is a fundamental tool of predictive analytics. Will a loan default (yes or no)? Will it prepay? Will a web visitor click on a link? Will she purchase something? Is an insurance claim fraudulent? Often in classification problems, one class is of primary interest (e.g., the fraudulent insurance claim) and, in binary classification, this class is designated as a 1, with the other, more prevalent class being a 0. Often, a key part of the process is estimating a *propensity score*, a probability of belonging to the class of interest. A common scenario is one in which the class of interest is relatively rare. The chapter concludes with a discussion of a variety of model assessment metrics that go beyond simple accuracy; these are important in the rare-class situation, when classifying all records as 0s can yield high accuracy.

---

1 This and subsequent sections in this chapter © 2017 Datastats, LLC, Peter Bruce and Andrew Bruce, used by permission.

2 It is certainly surprising that the first article on statistical classification was published in a journal devoted to eugenics. Indeed, there is a disconcerting connection between the early development of statistics and eugenics.

3 Not all methods provide unbiased estimates of probability. In most cases, it is sufficient that the method provide a ranking equivalent to the rankings that would result from an unbiased probability estimate; the cutoff method is then functionally equivalent.

4 The ROC curve was first used during World War II to describe the performance of radar receiving stations, whose job was to correctly identify (classify) reflected radar signals, and alert defense forces to incoming aircraft.

Chapter 6. Statistical Machine Learning

Recent advances in statistics have been devoted to developing more powerful automated techniques for predictive modeling — both regression and classification. These methods fall under the umbrella of *statistical machine learning*, and are distinguished from classical statistical methods in that they are data-driven and do not seek to impose linear or other overall structure on the data. The *K*-Nearest Neighbors method, for example, is quite simple: classify a record in accordance with how similar records are classified. The most successful and widely used techniques are based on *ensemble learning* applied to *decision trees*. The basic idea of ensemble learning is to use many models to form a prediction as opposed to just a single model. Decision trees are a flexible and automatic technique to learn rules about the relationships between predictor variables and outcome variables. It turns out that the combination of ensemble learning with decision trees leads to the top-performing off-the-shelf predictive modeling techniques.

The development of many of the techniques in statistical machine learning can be traced back to the statisticians Leo Breiman (see Figure 6-1) at the University of California at Berkeley and Jerry Friedman at Stanford University. Their work, along with other researchers at Berkeley and Stanford, started with the development of tree models in 1984. The subsequent development of ensemble methods of bagging and boosting in the 1990s established the foundation of statistical machine learning.

Figure 6-1. Leo Breiman, who was a professor of statistics at Berkeley, was at the forefront in development of many techniques at the core of a data scientist's toolkit

MACHINE LEARNING VERSUS STATISTICS

In the context of predictive modeling, what is the difference between machine learning and statistics? There is not a bright line dividing the two disciplines. Machine learning tends to be more focused on developing efficient algorithms that scale to large data in order to optimize the predictive model. Statistics generally pays more attention to the probabilistic theory and underlying structure of the model. Bagging, and the random forest (see "Bagging and the Random Forest"), grew up firmly in the statistics camp. Boosting (see "Boosting"), on the other hand, has been developed in both disciplines but receives more attention on the machine learning side of the divide. Regardless of the history, the promise of boosting ensures that it will thrive as a technique in both statistics and machine learning.

K-Nearest Neighbors

The idea behind *K*-Nearest Neighbors (KNN) is very simple.[1] For each record to be classified or predicted:

1. Find *K* records that have similar features (i.e., similar predictor values).
2. For classification: Find out what the majority class is among those similar records, and assign that class to the new record.
3. For prediction (also called *KNN regression*): Find the average among those similar records, and predict that average for the new record.

KEY TERMS FOR K-NEAREST NEIGHBORS

Neighbor
A record that has similar predictor values to another record.

Distance metrics
Measures that sum up in a single number how far one record is from another.

Standardization
Subtract the mean and divide by the standard deviation.

Synonym
Normalization

Z-score
The value that results after standardization.

K
The number of neighbors considered in the nearest neighbor calculation.

KNN is one of the simpler prediction/classification techniques: there is no model to be fit (as in regression). This doesn't mean that using KNN is an automatic procedure. The prediction results depend on how the features are scaled, how similarity is measured, and how big *K* is set. Also, all predictors must be in numeric form. We will illustrate it with a classification example.

A Small Example: Predicting Loan Default

Table 6-1 shows a few records of personal loan data from the Lending Club. Lending Club is a leader in peer-to-peer lending in which pools of investors make personal loans to individuals. The goal of an analysis would be to predict the outcome of a new potential loan: paid-off versus default.

Table 6-1. A few records and columns for Lending Club loan data

Outcome	Loan amount	Income	Purpose	Years employed	Home ownership	State
Paid off	10000	79100	debt_consolidation	11	MORTGAGE	NV
Paid off	9600	48000	moving	5	MORTGAGE	TN
Paid off	18800	120036	debt_consolidation	11	MORTGAGE	MD
Default	15250	232000	small_business	9	MORTGAGE	CA
Paid off	17050	35000	debt_consolidation	4	RENT	MD
Paid off	5500	43000	debt_consolidation	4	RENT	KS

Consider a very simple model with just two predictor variables: `dti`, which is the ratio of debt payments (excluding mortgage) to income, and `payment_inc_ratio`, which is the ratio of the loan payment to income. Both ratios are multiplied by 100. Using a small set of 200 loans, `loan200`, with known binary outcomes (default or no-default, specified in the predictor `outcome200`), and with *K* set to 20, the KNN estimate for a new loan to be predicted, `newloan`, with `dti=22.5` and `payment_inc_ratio=9` can be calculated in R as follows:

```
library(FNN)
knn_pred <- knn(train=loan200, test=newloan, cl=outcome200, k=20)
knn_pred == 'default'
[1] TRUE
```

The KNN prediction is for the loan to default.

While R has a native `knn` function, the contributed R package `FNN`, for Fast Nearest Neighbor, scales to big data better and provides more flexibility.

Figure 6-2 gives a visual display of this example. The new loan to be predicted is the square in the middle. The circles (default) and triangles (paid off) are the training data. The black line shows the boundary of the nearest 20 points. In this case, 14 defaulted loans lie within the circle as compared with only 6 paid-off loans. Hence, the predicted outcome of the loan is default.

NOTE

While the output of KNN for classification is typically a binary decision, such as default or paid off in the loan data, KNN routines usually offer the opportunity to output a probability (propensity) between 0 and 1. The probability is based on the fraction of one class in the *K* nearest neighbors. In the preceding example, this probability of default would have been estimated at $\frac{14}{20}$ or 0.7. Using a probability score lets you use classification rules other than simple majority votes (probability of 0.5). This is especially important in problems with imbalanced classes; see "Strategies for Imbalanced Data". For example, if the goal is to identify members of a rare class, the cutoff would typically be set below 50%. One common approach is to set the cutoff at the probability of the rare event.

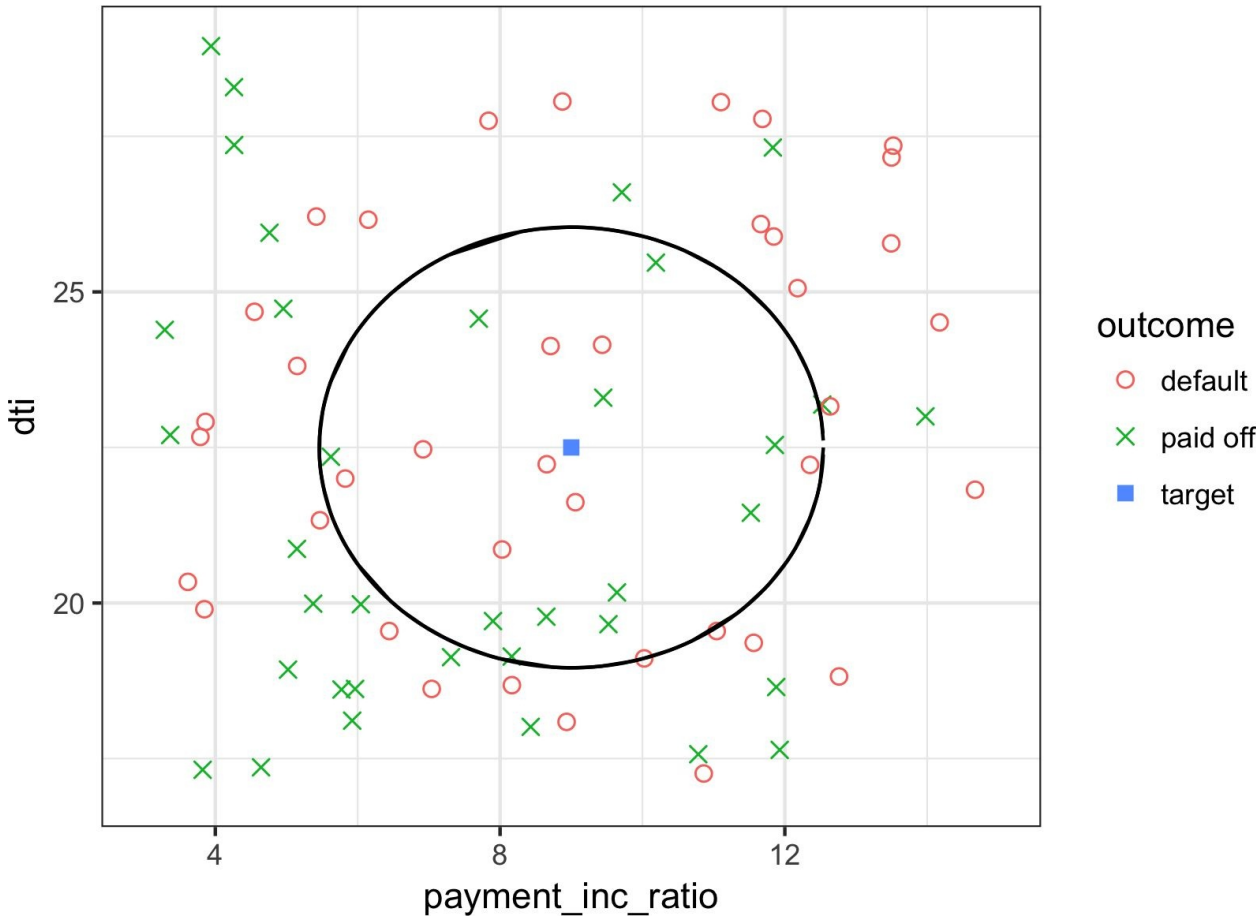

Figure 6-2. KNN prediction of loan default using two variables: debt-to-income ratio and loan payment-to-income ratio

Distance Metrics

Similarity (nearness) is determined using a *distance metric*, which is a function that measures how far two records (x_1, x_2, … x_p) and (u_1, u_2, … u_p) are from one another. The most popular distance metric between two vectors is *Euclidean distance*. To measure the Euclidean distance between two vectors, subtract one from the other, square the differences, sum them, and take the square root:

$$\sqrt{(x_1 - u_1)^2 + (x_2 - u_2)^2 + \cdots + (x_p - u_p)^2}.$$

Euclidean distance offers special computational advantages. This is particularly important for large data sets since KNN involves $K \times n$ pairwise comparisons where n is the number of rows.

Another common distance metric for numeric data is *Manhattan distance*:

$$| x_1 - u_1 | + | x_2 - u_2 | + \cdots + | x_p - u_p |$$

Euclidean distance corresponds to the straight-line distance between two points (e.g., as the crow flies). Manhattan distance is the distance between two points traversed in a single direction at a time (e.g., traveling along rectangular city blocks). For this reason, Manhattan distance is a useful approximation if similarity is defined as point-to-point travel time.

In measuring distance between two vectors, variables (features) that are measured with comparatively large scale will dominate the measure. For example, for the loan data, the distance would be almost solely a function of the income and loan amount variables, which are measured in tens or hundreds of thousands. Ratio variables would count for practically nothing in comparison. We address this problem by standardizing the data; see “Standardization (Normalization, Z-Scores)”.

OTHER DISTANCE METRICS

There are numerous other metrics for measuring distance between vectors. For numeric data, *Mahalanobis distance* is attractive since it accounts for the correlation between two variables. This is useful since if two variables are highly correlated, Mahalanobis will essentially treat these as a single variable in terms of distance. Euclidean and Manhattan distance do not account for the correlation, effectively placing greater weight on the attribute that underlies those features. The downside of using Mahalanobis distance is increased computational effort and complexity; it is computed using the *covariance matrix*; see "Covariance Matrix".

One Hot Encoder

The loan data in Table 6-1 includes several factor (string) variables. Most statistical and machine learning models require this type of variable to be converted to a series of binary dummy variables conveying the same information, as in Table 6-2. Instead of a single variable denoting the home occupant status as "owns with a mortage," "owns with no mortgage," "rents," or "other," we end up with four binary variables. The first would be "owns with a mortgage — Y/N," the second would be "owns with no mortgage — Y/N," and so on. This one predictor, home occupant status, thus yields a vector with one 1 and three 0s, that can be used in statistical and machine learning algorithms. The phrase *one hot encoding* comes from digital circuit terminology, where it describes circuit settings in which only one bit is allowed to be positive (hot).

Table 6-2. Representing home ownership factor data as a numeric dummy variable

MORTGAGE	OTHER	OWN	RENT
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0
0	0	0	1
0	0	0	1

NOTE

In linear and logistic regression, one hot encoding causes problems with multicollinearity; see "Multicollinearity". In such cases, one dummy is omitted (its value can be inferred from the other values). This is not an issue with KNN and other methods.

Standardization (Normalization, Z-Scores)

In measurement, we are often not so much interested in "how much" but "how different from the average." Standardization, also called *normalization*, puts all variables on similar scales by subtracting the mean and dividing by the standard deviation. In this way, we ensure that a variable does not overly influence a model simply due to the scale of its original measurement.

$$z = \frac{x - \bar{x}}{s}$$

These are commonly refered to as *z-scores*. Measurements are then stated in terms of "standard deviations away from the mean." In this way, a variable's impact on a model is not affected by the scale of its original measurement.

> **CAUTION**
>
> *Normalization* in this statistical context is not to be confused with *database normalization*, which is the removal of redundant data and the verification of data dependencies.

For KNN and a few other procedures (e.g., principal components analysis and clustering), it is essential to consider standardizing the data prior to applying the procedure. To illustrate this idea, KNN is applied to the loan data using `dti` and `payment_inc_ratio` (see "A Small Example: Predicting Loan Default") plus two other variables: `revol_bal`, the total revolving credit available to the applicant in dollars, and `revol_util`, the percent of the credit being used. The new record to be predicted is shown here:

```
newloan
  payment_inc_ratio dti revol_bal revol_util
1            2.3932   1      1687        9.4
```

The magnitude of `revol_bal`, which is in dollars, is much bigger than the other variables. The `knn` function returns the index of the nearest neighbors as an attribute `nn.index`, and this can be used to show the top-five closest rows in

`loan_df`:

```
loan_df <- model.matrix(~ -1 + payment_inc_ratio + dti + revol_bal +
                          revol_util, data=loan_data)
knn_pred <- knn(train=loan_df, test=newloan, cl=outcome, k=5)
loan_df[attr(knn_pred,"nn.index"),]
      payment_inc_ratio  dti revol_bal revol_util
36054           2.22024 0.79      1687        8.4
33233           5.97874 1.03      1692        6.2
28989           5.65339 5.40      1694        7.0
29572           5.00128 1.84      1695        5.1
20962           9.42600 7.14      1683        8.6
```

The value of `revol_bal` in these neighbors is very close to its value in the new record, but the other predictor variables are all over the map and essentially play no role in determining neighbors.

Compare this to KNN applied to the standardized data using the R function `scale`, which computes the *z*-score for each variable:

```
loan_std <- scale(loan_df)
knn_pred <- knn(train=loan_std, test=newloan_std, cl=outcome, k=5)
loan_df[attr(knn_pred,"nn.index"),]
      payment_inc_ratio  dti revol_bal revol_util
2081            2.61091 1.03      1218        9.7
36054           2.22024 0.79      1687        8.4
23655           2.34286 1.12       523       10.7
41327           2.15987 0.69      2115        8.1
39555           2.76891 0.75      2129        9.5
```

The five nearest neighbors are much more alike in all the variables providing a more sensible result. Note that the results are displayed on the original scale, but KNN was applied to the scaled data and the new loan to be predicted.

TIP

Using the *z*-score is just one way to rescale variables. Instead of the mean, a more robust estimate of location could be used, such as the median. Likewise, a different estimate of scale such as the interquartile range could be used instead of the standard deviation. Sometimes, variables are "squashed" into the 0–1 range. It's also important to realize that scaling each variable to have unit variance is somewhat arbitrary. This implies that each variable is thought to have the same importance in predictive power. If you have subjective knowledge that some variables are more important than others, then these could be scaled up. For example, with the loan data, it is reasonable to expect that the payment-to-income ratio is very important.

NOTE

Normalization (standardization) does not change the distributional shape of the data; it does not make it normally shaped if it was not already normally shaped (see “Normal Distribution”).

Choosing K

The choice of *K* is very important to the performance of KNN. The simplest choice is to set $K = 1$, known as the 1-nearest neighbor classifier. The prediction is intuitive: it is based on finding the data record in the training set most similar to the new record to be predicted. Setting $K = 1$ is rarely the best choice; you'll almost always obtain superior performance by using *K* > 1-nearest neighbors.

Generally speaking, if *K* is too low, we may be overfitting: including the noise in the data. Higher values of *K* provide smoothing that reduces the risk of overfitting in the training data. On the other hand, if *K* is too high, we may oversmooth the data and miss out on KNN's ability to capture the local structure in the data, one of its main advantages.

The *K* that best balances between overfitting and oversmoothing is typically determined by accuracy metrics and, in particular, accuracy with holdout or validation data. There is no general rule about the best *K* — it depends greatly on the nature of the data. For highly structured data with little noise, smaller values of *K* work best. Borrowing a term from the signal processing community, this type of data is sometimes referred to as having a high *signal-to-noise ratio* (*SNR*). Examples of data with typically high SNR are handwriting and speech recognition. For noisy data with less structure (data with a low SNR), such as the loan data, larger values of *K* are appropriate. Typically, values of *K* fall in the range 1 to 20. Often, an odd number is chosen to avoid ties.

BIAS-VARIANCE TRADEOFF

The tension between oversmoothing and overfitting is an instance of the *bias-variance tradeoff*, an ubiquitous problem in statistical model fitting. Variance refers to the modeling error that occurs because of the choice of training data; that is, if you were to choose a different set of training data, the resulting model would be different. Bias refers to the modeling error that occurs because you have not properly identified the underlying real-world scenario; this error would not disappear if you simply added more training data. When a flexible model is overfit, the variance increases. You can reduce this by using a simpler model, but the bias may increase due to the loss of flexibility in modeling the real underlying situation. A general approach to handling this tradeoff is through *cross-validation*. See "Cross-Validation" for more details.

KNN as a Feature Engine

KNN gained its popularity due to its simplicity and intuitive nature. In terms of performance, KNN by itself is usually not competitive with more sophisticated classification techniques. In practical model fitting, however, KNN can be used to add "local knowledge" in a staged process with other classification techniques.

1. KNN is run on the data, and for each record, a classification (or quasi-probability of a class) is derived.

2. That result is added as a new feature to the record, and another classification method is then run on the data. The original predictor variables are thus used twice.

At first you might wonder whether this process, since it uses some predictors twice, causes a problem with multicollinearity (see "Multicollinearity"). This is not an issue, since the information being incorporated into the second-stage model is highly local, derived only from a few nearby records, and is therefore additional information, and not redundant.

> **NOTE**
>
> You can think of this staged use of KNN as a form of ensemble learning, in which multiple predictive modeling methods are used in conjunction with one another. It can also be considered as a form of feature engineering where the aim is to derive features (predictor variables) that have predictive power. Often this involves some manual review of the data; KNN gives a fairly automatic way to do this.

For example, consider the King County housing data. In pricing a home for sale, a realtor will base the price on similar homes recently sold, known as "comps." In essence, realtors are doing a manual version of KNN: by looking at the sale prices of similar homes, they can estimate what a home will sell for. We can create a new feature for a statistical model to mimic the real estate professional by applying KNN to recent sales. The predicted value is the sales price and the existing predictor variables could include location, total square feet, type of structure, lot size, and number of bedrooms and bathrooms. The new predictor variable (feature) that we add via KNN is the KNN predictor for each record

(analogous to the realtors' comps). Since we are predicting a numerical value, the average of the *K*-Nearest Neighbors is used instead of a majority vote (known as *KNN regression*).

Similarly, for the loan data, we can create features that represent different aspects of the loan process. For example, the following would build a feature that represents a borrower's creditworthiness:

```
borrow_df <- model.matrix(~ -1 + dti + revol_bal + revol_util + open_acc +
                          delinq_2yrs_zero + pub_rec_zero, data=loan_data)
borrow_knn <- knn(borrow_df, test=borrow_df, cl=loan_data[, 'outcome'],
                  prob=TRUE, k=10)
prob <- attr(borrow_knn, "prob")
borrow_feature <- ifelse(borrow_knn=='default', prob, 1-prob)
summary(borrow_feature)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.4000  0.5000  0.5012  0.6000  1.0000
```

The result is a feature that predicts the likelihood a borrower will default based on his credit history.

KEY IDEAS FOR K-NEAREST NEIGHBORS

- *K*-Nearest Neighbors (KNN) classifies a record by assigning it to the class that similar records belong to.
- Similarity (distance) is determined by Euclidian distance or other related metrics.
- The number of nearest neighbors to compare a record to, *K*, is determined by how well the algorithm performs on training data, using different values for *K*.
- Typically, the predictor variables are standardized so that variables of large scale do not dominate the distance metric.
- KNN is often used as a first stage in predictive modeling, and the predicted value is added back into the data as a *predictor* for second-stage (non-KNN) modeling.

Tree Models

Tree models, also called *Classification and Regression Trees* (*CART*),[2] *decision trees*, or just *trees*, are an effective and popular classification (and regression) method initially developed by Leo Breiman and others in 1984. Tree models, and their more powerful descendents *random forests* and *boosting* (see "Bagging and the Random Forest" and "Boosting"), form the basis for the most widely used and powerful predictive modeling tools in data science for both regression and classification.

KEY TERMS FOR TREES

Recursive partitioning

Repeatedly dividing and subdividing the data with the goal of making the outcomes in each final subdivision as homogeneous as possible.

Split value

A predictor value that divides the records into those where that predictor is less than the split value, and those where it is more.

Node

In the decision tree, or in the set of corresponding branching rules, a node is the graphical or rule representation of a split value.

Leaf

The end of a set of if-then rules, or branches of a tree — the rules that bring you to that leaf provide one of the classification rules for any record in a tree.

Loss

The number of misclassifications at a stage in the splitting process; the more losses, the more impurity.

Impurity

The extent to which a mix of classes is found in a subpartition of the data (the more mixed, the more impure).

Synonym

Heterogeneity

Antonym

Homogeneity, purity

Pruning

The process of taking a fully grown tree and progressively cutting its branches back, to reduce overfitting.

A tree model is a set of “if-then-else” rules that are easy to understand and to implement. In contrast to regression and logistic regression, trees have the ability to discover hidden patterns corresponding to complex interactions in the data. However, unlike KNN or naive Bayes, simple tree models can be expressed in terms of predictor relationships that are easily interpretable.

Saishna Budhathoki

DECISION TREES IN OPERATIONS RESEARCH

The term *decision trees* has a different (and older) meaning in decision science and operations research, where it refers to a human decision analysis process. In this meaning, decision points, possible outcomes, and their estimated probabilities are laid out in a branching diagram, and the decision path with the maximum expected value is chosen.

A Simple Example

The two main packages to fit tree models in R are `rpart` and `tree`. Using the `rpart` package, a model is fit to a sample of 3,000 records of the loan data using the variables `payment_inc_ratio` and `borrower_score` (see "K-Nearest Neighbors" for a description of the data).

```
library(rpart)
loan_tree <- rpart(outcome ~ borrower_score + payment_inc_ratio,
                   data=loan_data, control = rpart.control(cp=.005))
plot(loan_tree, uniform=TRUE, margin=.05)
text(loan_tree)
```

The resulting tree is shown in Figure 6-3. These classification rules are determined by traversing through a hierarchical tree, starting at the root until a leaf is reached.

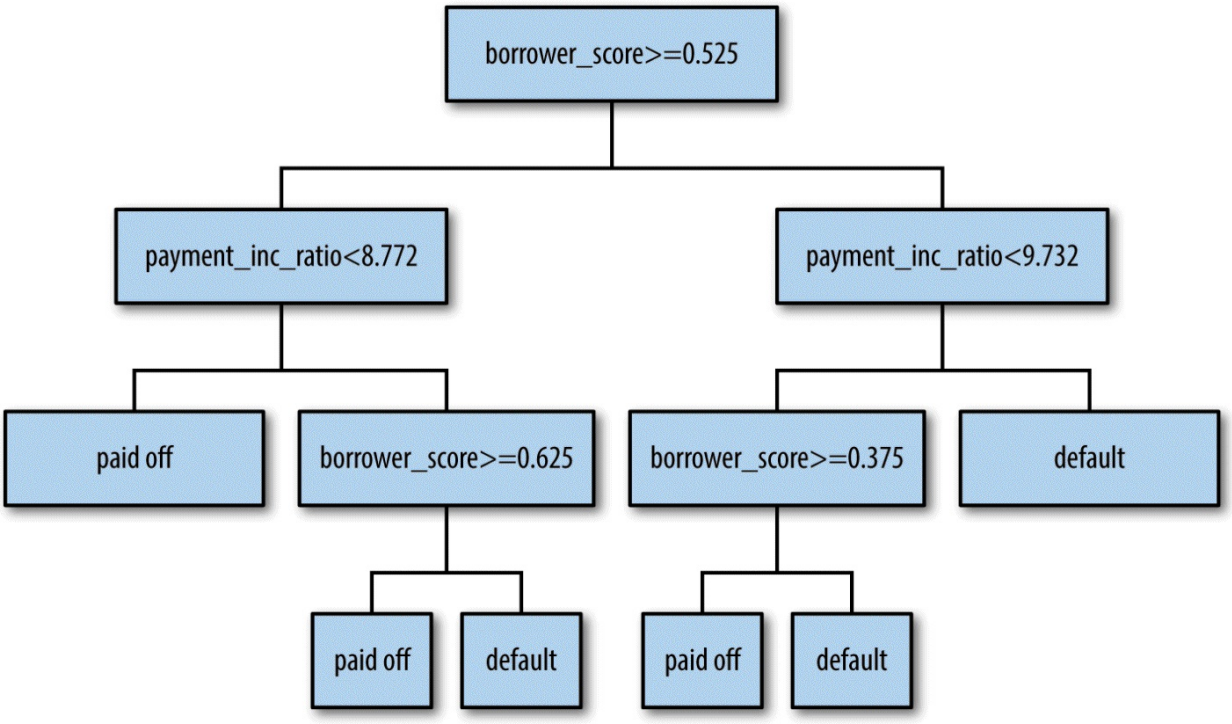

Figure 6-3. The rules for a simple tree model fit to the loan data

Typically, the tree is plotted upside-down, so the root is at the top and the leaves are at the bottom. For example, if we get a loan with `borrower_score` of 0.6 and

a `payment_inc_ratio` of 8.0, we end up at the leftmost leaf and predict the loan will be paid off.

A nicely printed version of the tree is also easily produced:

```
loan_tree
n= 3000

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 3000 1467 paid off (0.5110000 0.4890000)
   2) borrower_score>=0.525 1283  474 paid off (0.6305534 0.3694466)
     4) payment_inc_ratio< 8.772305 845  249 paid off (0.7053254 0.2946746) *
     5) payment_inc_ratio>=8.772305 438  213 default (0.4863014 0.5136986)
      10) borrower_score>=0.625 149   60 paid off (0.5973154 0.4026846) *
      11) borrower_score< 0.625 289  124 default (0.4290657 0.5709343) *
   3) borrower_score< 0.525 1717  724 default (0.4216657 0.5783343)
     6) payment_inc_ratio< 9.73236 1082  517 default (0.4778189 0.5221811)
      12) borrower_score>=0.375 784  384 paid off (0.5102041 0.4897959) *
      13) borrower_score< 0.375 298  117 default (0.3926174 0.6073826) *
     7) payment_inc_ratio>=9.73236 635  207 default (0.3259843 0.6740157) *
```

The depth of the tree is shown by the indent. Each node corresponds to a provisional classification determined by the prevalent outcome in that partition. The “loss” is the number of misclassifications yielded by the provisional classification in a partition. For example, in node 2, there were 474 misclassification out of a total of 1,467 total records. The values in the parentheses correspond to the proportion of records that are paid off and default, respectively. For example, in node 13, which predicts default, over 60 percent of the records are loans that are in default.

The Recursive Partitioning Algorithm

The algorithm to construct a decision tree, called *recursive partitioning* , is straightforward and intuitive. The data is repeatedly partitioned using predictor values that do the best job of separating the data into relatively homogeneous partitions. Figure 6-4 shows a picture of the partitions created for the tree in Figure 6-3. The first rule is `borrower_score >= 0.525` and is depicted by rule 1 in the plot. The second rule is `payment_inc_ratio < 9.732` and divides the righthand region in two.

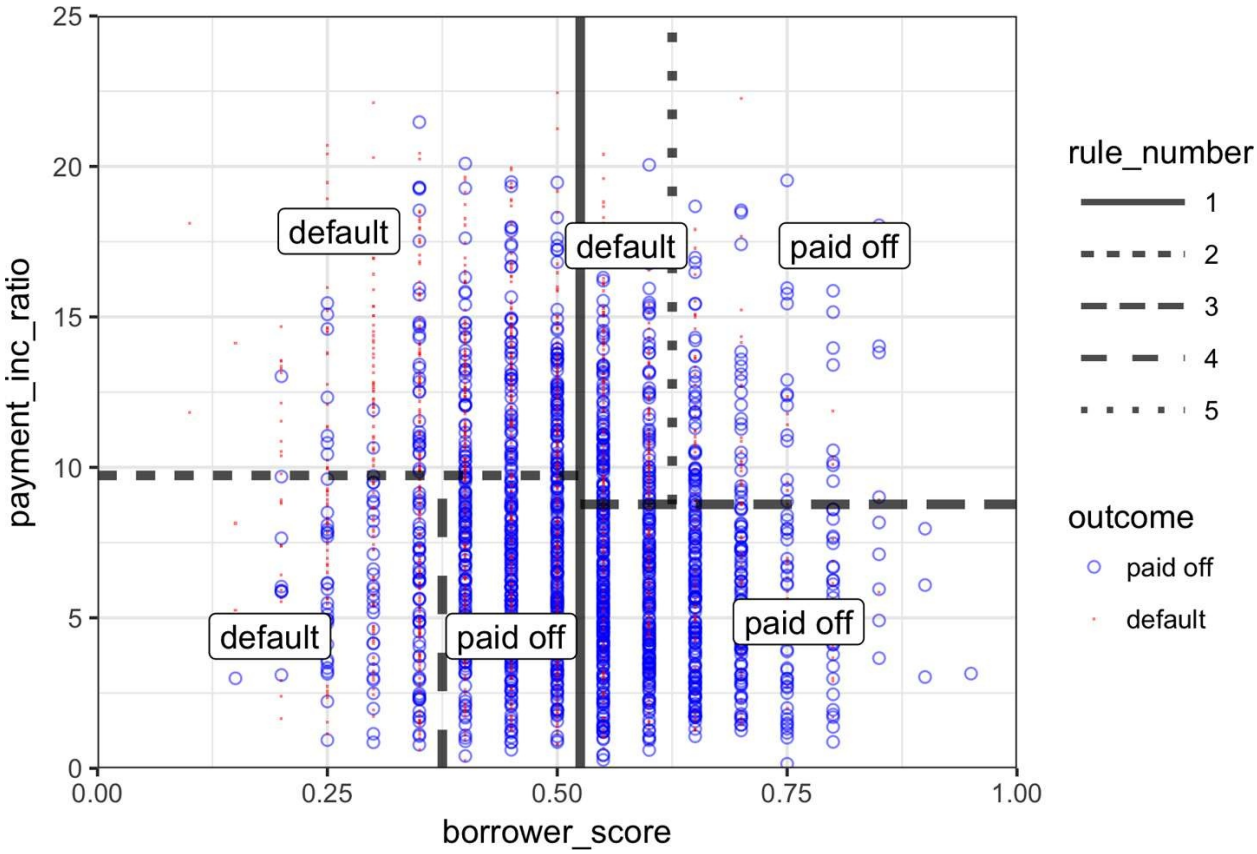

Figure 6-4. The rules for a simple tree model fit to the loan data

Suppose we have a response variable Y and a set of P predictor variables X_j for $j = 1, \cdots, P$. For a partition A of records, recursive partitioning will find the best way to partition A into two subpartitions:

1. For each predictor variable X_j,
    a. For each value s_j of X_j:
        i. Split the records in A with X_j values $< s_j$ as one partition, and the remaining records where $X_j \geq s_j$ as another partition.
        ii. Measure the homogeneity of classes within each subpartition of A.
    b. Select the value of s_j that produces maximum within-partition homogeneity of class.
2. Select the variable X_j and the split value s_j that produces maximum

within-partition homogeneity of class.

Now comes the recursive part:

1. Initialize A with the entire data set.
2. Apply the partitioning algorithm to split A into two subpartitions, A_1 and A_2.
3. Repeat step 2 on subpartitions A_1 and A_2.
4. The algorithm terminates when no further partition can be made that sufficiently improves the homogeneity of the partitions.

The end result is a partitioning of the data, as in Figure 6-4 except in P-dimensions, with each partition predicting an outcome of 0 or 1 depending on the majority vote of the reponse in that partition.

NOTE

In addition to a binary 0/1 prediction, tree models can produce a probability estimate based on the number of 0s and 1s in the partition. The estimate is simply the sum of 0s or 1s in the partition divided by the number of observations in the partition.

$$\text{Prob}(Y = 1) = \frac{\text{Number of 1s in the partition}}{\text{Size of the partition}}$$

The estimated $\text{Prob}(Y = 1)$ can then be converted to a binary decision; for example, set the estimate to 1 if Prob(Y = 1) > 0.5.

Measuring Homogeneity or Impurity

Tree models recursively create partitions (sets of records), *A*, that predict an outcome of $Y = 0$ or $Y = 1$. You can see from the preceding algorithm that we need a way to measure homogeneity, also called *class purity*, within a partition. Or, equivalently, we need to measure the impurity of a partition. The accuracy of the predictions is the proportion p of misclassified records within that partition, which ranges from 0 (perfect) to 0.5 (purely random guessing).

It turns out that accuracy is not a good measure for impurity. Instead, two common measures for impurity are the *Gini impurity* and *entropy* or *information*. While these (and other) impurity measures apply to classification problems with more than two classes, we focus on the binary case. The Gini impurity for a set of records A is:

$$I(A) = p(1 - p)$$

The entropy measure is given by:

$$I(A) = -p\log_2(p) - (1 - p)\log_2(1 - p)$$

Figure 6-5 shows that Gini impurity (rescaled) and entropy measures are similar, with entropy giving higher impurity scores for moderate and high accuracy rates.

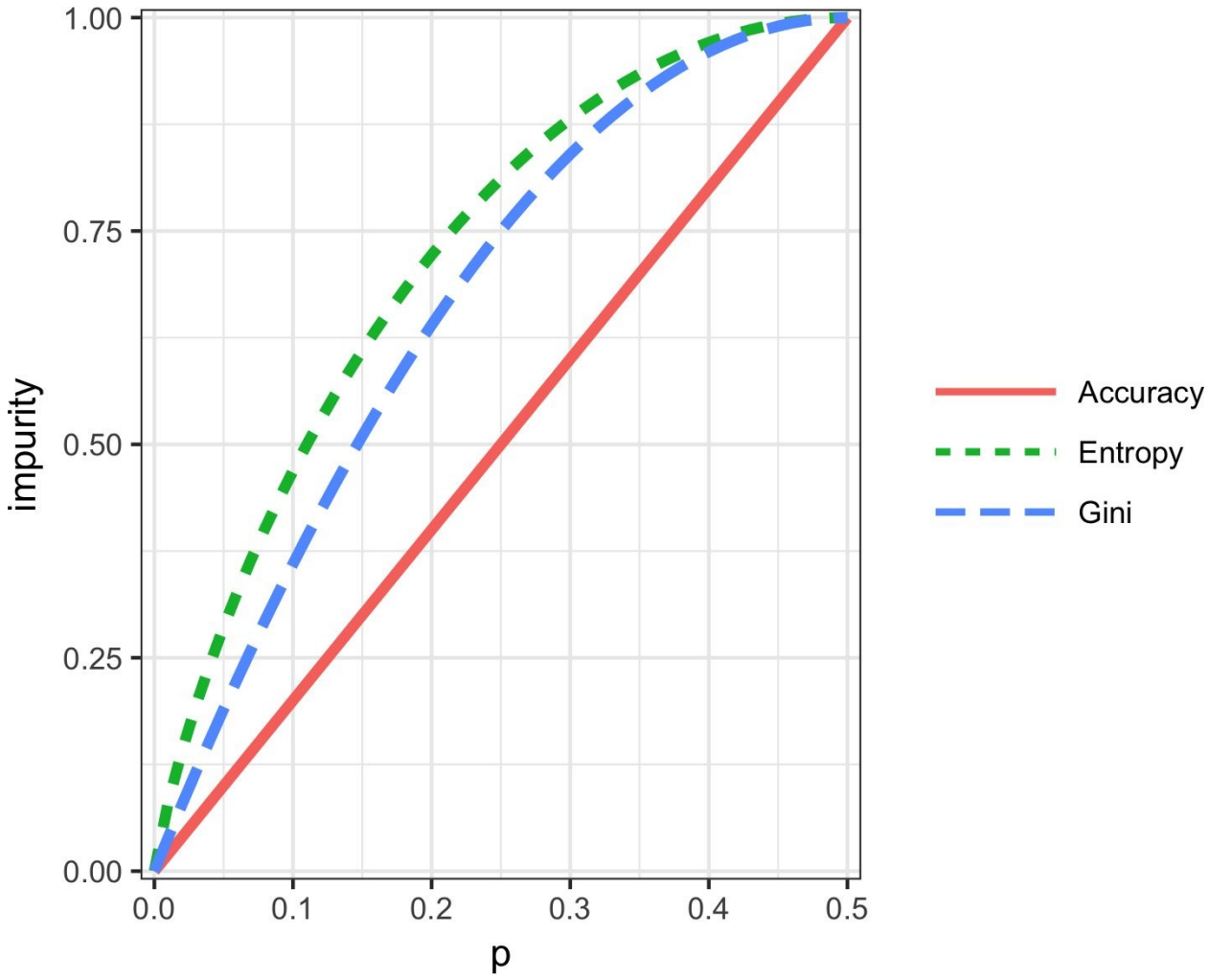

Figure 6-5. Gini impurity and entropy measures

GINI COEFFICIENT

Gini impurity is not to be confused with the *Gini coefficient*. They represent similar concepts, but the Gini coefficient is limited to the binary classification problem and is related to the AUC metric (see "AUC").

The impurity metric is used in the splitting algorithm described earlier. For each proposed partition of the data, impurity is measured for each of the partitions that result from the split. A weighted average is then calculated, and whichever partition (at each stage) yields the lowest weighted average is selected.

Stopping the Tree from Growing

As the tree grows bigger, the splitting rules become more detailed, and the tree gradually shifts from identifying “big” rules that identify real and reliable relationships in the data to “tiny” rules that reflect only noise. A fully grown tree results in completely pure leaves and, hence, 100% accuracy in classifying the data that it is trained on. This accuracy is, of course, illusory — we have overfit (see Bias-Variance Tradeoff) the data, fitting the noise in the training data, not the signal that we want to identify in new data.

PRUNING

A simple and intuitive method of reducing tree size is to *prune* back the terminal and smaller branches of the tree, leaving a smaller tree. How far should the pruning proceed? A common technique is to prune the tree back to the point where the error on holdout data is minimized. When we combine predictions from multiple trees (see "Bagging and the Random Forest"), however, we will need a way to stop tree growth. Pruning plays a role in the process of cross-validation to determine how far to grow trees that are used in ensemble methods.

We need some way to determine when to stop growing a tree at a stage that will generalize to new data. There are two common ways to stop splitting:

- Avoid splitting a partition if a resulting subpartition is too small, or if a terminal leaf is too small. In `rpart`, these constraints are controlled separately by the parameters `minsplit` and `minbucket`, respectively, with defaults of `20` and `7`.

- Don't split a partition if the new partition does not "significantly" reduce the impurity. In `rpart`, this is controlled by the *complexity parameter* `cp`, which is a measure of how complex a tree is — the more complex, the greater the value of `cp`. In practice, `cp` is used to limit tree growth by attaching a penalty to additional complexity (splits) in a tree.

The first method involves arbitrary rules, and can be usful for exploratory work, but we can't easily determine optimum values (i.e., values that maximize predictive accuracy with new data). With the complexity parameter, `cp`, we can estimate what size tree will perform best with new data.

If `cp` is too small, then the tree will overfit the data, fitting noise and not signal. On the other hand, if `cp` is too large, then the tree will be too small and have little predictive power. The default in `rpart` is 0.01, although for larger data sets, you are likely to find this is too large. In the previous example, `cp` was set to `0.005` since the default led to a tree with a single split. In exploratory analysis, it is sufficient to simply try a few values.

Determining the optimum `cp` is an instance of the bias-variance tradeoff (see Bias-Variance Tradeoff). The most common way to estimate a good value of `cp` is via

cross-validation (see "Cross-Validation"):

1. Partition the data into training and validation (holdout) sets.
2. Grow the tree with the training data.
3. Prune it successively, step by step, recording `cp` (using the *training* data) at each step.
4. Note the `cp` that corresponds to the minimum error (loss) on the *validation* data.
5. Repartition the data into training and validation, and repeat the growing, pruning, and `cp` recording process.
6. Do this again and again, and average the `cps` that reflect minimum error for each tree.
7. Go back to the original data, or future data, and grow a tree, stopping at this optimum `cp` value.

In `rpart`, you can use the argument `cptable` to produce a table of the CP values and their associated cross-validation error (`xerror` in R), from which you can determine the CP value that has the lowest cross-validation error.

Predicting a Continuous Value

Predicting a continuous value (also termed *regression*) with a tree follows the same logic and procedure, except that impurity is measured by squared deviations from the mean (squared errors) in each subpartition, and predictive performance is judged by the square root of the mean squared error (RMSE) (see "Assessing the Model") in each partition.

How Trees Are Used

One of the big obstacles faced by predictive modelers in organizations is the perceived "black box" nature of the methods they use, which gives rise to opposition from other elements of the organization. In this regard, the tree model has two appealing aspects.

- Tree models provide a visual tool for exploring the data, to gain an idea of what variables are important and how they relate to one another. Trees can capture nonlinear relationships among predictor variables.

- Tree models provide a set of rules that can be effectively communicated to nonspecialists, either for implementation or to "sell" a data mining project.

When it comes to prediction, however, harnassing the results from multiple trees is typically more powerful than just using a single tree. In particular, the random forest and boosted tree algorithms almost always provide superior predictive accuracy and performance (see "Bagging and the Random Forest" and "Boosting"), but the aforementioned advantages of a single tree are lost.

KEY IDEAS

- Decision trees produce a set of rules to classify or predict an outcome.
- The rules correspond to successive partitioning of the data into subpartitions.
- Each partition, or split, references a specific value of a predictor variable and divides the data into records where that predictor value is above or below that split value.
- At each stage, the tree algorithm chooses the split that minimizes the outcome impurity within each subpartition.
- When no further splits can be made, the tree is fully grown and each terminal node, or leaf, has records of a single class; new cases following that rule (split) path would be assigned that class.
- A fully grown tree overfits the data and must be pruned back so that it captures signal and not noise.
- Multiple-tree algorithms like random forests and boosted trees yield better predictive performance, but lose the rule-based communicative power of single trees.

Further Reading

- Analytics Vidhya Content Team, "A Complete Tutorial on Tree Based Modeling from Scratch (in R & Python)", April 12, 2016.

- Terry M. Therneau, Elizabeth J. Atkinson, and the Mayo Foundation, "An Introduction to Recursive Partitioning Using the RPART Routines", June 29, 2015.

Bagging and the Random Forest

In 1907, the statistician Sir Francis Galton was visiting a county fair in England, at which a contest was being held to guess the dressed weight of an ox that was on exhibit. There were 800 guesses, and, while the individual guesses varied widely, both the mean and the median came out within 1% of the ox's true weight. James Suroweicki has explored this phenomenon in his book *The Wisdom of Crowds* (Doubleday, 2004). This principle applies to predictive models, as well: averaging (or taking majority votes) of multiple models — an *ensemble* of models — turns out to be more accurate than just selecting one model.

KEY TERMS FOR BAGGING AND THE RANDOM FOREST

Ensemble
Forming a prediction by using a collection of models.

Synonym
Model averaging

Bagging
A general technique to form a collection of models by bootstrapping the data.

Synonym
Bootstrap aggregation

Random forest
A type of bagged estimate based on decision tree models.

Synonym
Bagged decision trees

Variable importance
A measure of the importance of a predictor variable in the performance of the model.

The ensemble approach has been applied to and across many different modeling methods, most publicly in the Netflix Contest, in which Netflix offered a $1 million prize to any contestant who came up with a model that produced a 10% improvement in predicting the rating that a Netflix customer would award a movie. The simple version of ensembles is as follows:

1. Develop a predictive model and record the predictions for a given data

set.

2. Repeat for multiple models, on the same data.

3. For each record to be predicted, take an average (or a weighted average, or a majority vote) of the predictions.

Ensemble methods have been applied most systematically and effectively to decision trees. Ensemble tree models are so powerful that they provide a way to build good predictive models with relatively little effort.

Going beyond the simple ensemble algorithm, there are two main variants of ensemble models: *bagging* and *boosting*. In the case of ensemble tree models, these are refered to as *random forest* models and *boosted tree* models. This section focuses on bagging; boosting is covered in "Boosting".

Bagging

Bagging, which stands for "bootstrap aggregating," was introduced by Leo Breiman in 1994. Suppose we have a response Y and P predictor variables $\mathbf{X} = X_1, X_2, \cdots, X_P$ with n records.

Bagging is like the basic algorithm for ensembles, except that, instead of fitting the various models to the same data, each new model is fit to a bootstrap resample. Here is the algorithm presented more formally:

1. Initialize M, the number of models to be fit, and n, the number of records to choose ($n < N$). Set the iteration $m = 1$.
2. Take a bootstrap resample (i.e., with replacement) of n records from the training data to form a subsample Y_m and $\mathbf{X}_m$ (the bag).
3. Train a model using Y_m and $\mathbf{X}_m$ to create a set of decision rules $\hat{f}_m(\mathbf{X})$.
4. Increment the model counter $m = m + 1$. If $m <= M$, go to step 1.

In the case where $\hat{f}_m$ predicts the probability $Y = 1$, the bagged estimate is given by:

$$\hat{f} = \frac{1}{M}\left(\hat{f}_1(\mathbf{X}) + \hat{f}_2(\mathbf{X}) + \cdots \hat{f}_M(\mathbf{X})\right)$$

Random Forest

The *random forest* is based on applying bagging to decision trees with one important extension: in addition to sampling the records, the algorithm also samples the variables.[3] In traditional decision trees, to determine how to create a subpartition of a partition *A*, the algorithm makes the choice of variable and split point by minimizing a criterion such as Gini impurity (see "Measuring Homogeneity or Impurity"). With random forests, at each stage of the algorithm, the choice of variable is limited to a *random subset of variables*. Compared to the basic tree algorithm (see "The Recursive Partitioning Algorithm"), the random forest algorithm adds two more steps: the bagging discussed earlier (see "Bagging and the Random Forest"), and the bootstrap sampling of variables at each split:

1. Take a bootstrap (with replacement) subsample from the *records*.
2. For the first split, sample $p < P$ *variables* at random without replacement.
3. For each of the sampled variables $X_{j(1)}, X_{j(2)}, \ldots, X_{j(p)}$, apply the splitting algorithm:
    a. For each value $s_{j(k)}$ of $X_{j(k)}$:
        i. Split the records in partition *A* with $X_{j(k)} < s_{j(k)}$ as one partition, and the remaining records where $X_{j(k)} \geq s_{j(k)}$ as another partition.
        ii. Measure the homogeneity of classes within each subpartition of *A*.
    b. Select the value of $s_{j(k)}$ that produces maximum within-partition homogeneity of class.
4. Select the variable $X_{j(k)}$ and the split value $s_{j(k)}$ that produces maximum within-partition homogeneity of class.
5. Proceed to the next split and repeat the previous steps, starting with step 2.

6. Continue with additional splits following the same procedure until the tree is grown.

7. Go back to step 1, take another bootstrap subsample, and start the process over again.

How many variables to sample at each step? A rule of thumb is to choose $\sqrt{P}$ where P is the number of predictor variables. The package `randomForest` implements the random forest in R. The following applies this package to the loan data (see "K-Nearest Neighbors" for a description of the data).

```
> library(randomForest)
> rf <- randomForest(outcome ~ borrower_score + payment_inc_ratio,
                     data=loan3000)
Call:
 randomForest(formula = outcome ~ borrower_score + payment_inc_ratio,
              data = loan3000)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 38.53%
Confusion matrix:
         paid off default class.error
paid off     1089     425   0.2807133
default       731     755   0.4919246
```

By default, 500 trees are trained. Since there are only two variables in the predictor set, the algorithm randomly selects the variable on which to split at each stage (i.e., a bootstrap subsample of size 1).

The *out-of-bag* (*OOB*) estimate of error is the error rate for the trained models, applied to the data left out of the training set for that tree. Using the output from the model, the OOB error can be plotted versus the number of trees in the random forest:

```
error_df = data.frame(error_rate = rf$err.rate[,'OOB'],
                      num_trees = 1:rf$ntree)
ggplot(error_df, aes(x=num_trees, y=error_rate)) +
  geom_line()
```

The result is shown in Figure 6-6. The error rate rapidly decreases from over .44 before stabilizing around .385. The predicted values can be obtained from the `predict` function and plotted as follows:

Saishna Budhathoki

```
pred <- predict(loan_lda)
rf_df <- cbind(loan3000, pred_default=pred[,'default']>.5)
ggplot(data=rf_df, aes(x=borrower_score, y=payment_inc_ratio,
                       color=pred_default, shape=pred_default)) +
  geom_point(alpha=.6, size=2) +
  scale_shape_manual( values=c( 46, 4))
```

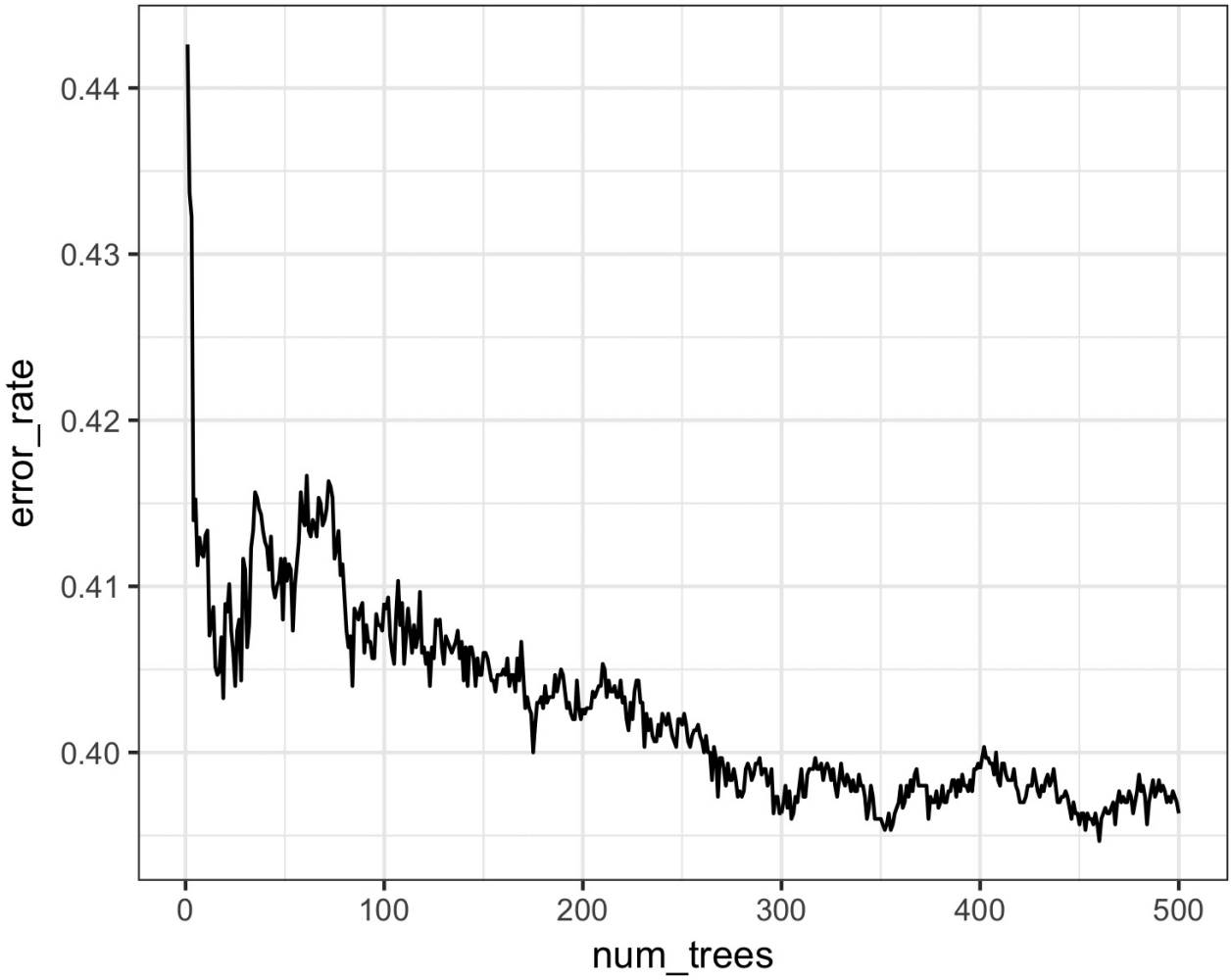

Figure 6-6. The improvement in accuracy of the random forest with the addition of more trees

The plot, shown in Figure 6-7, is quite revealing about the nature of the random forest.

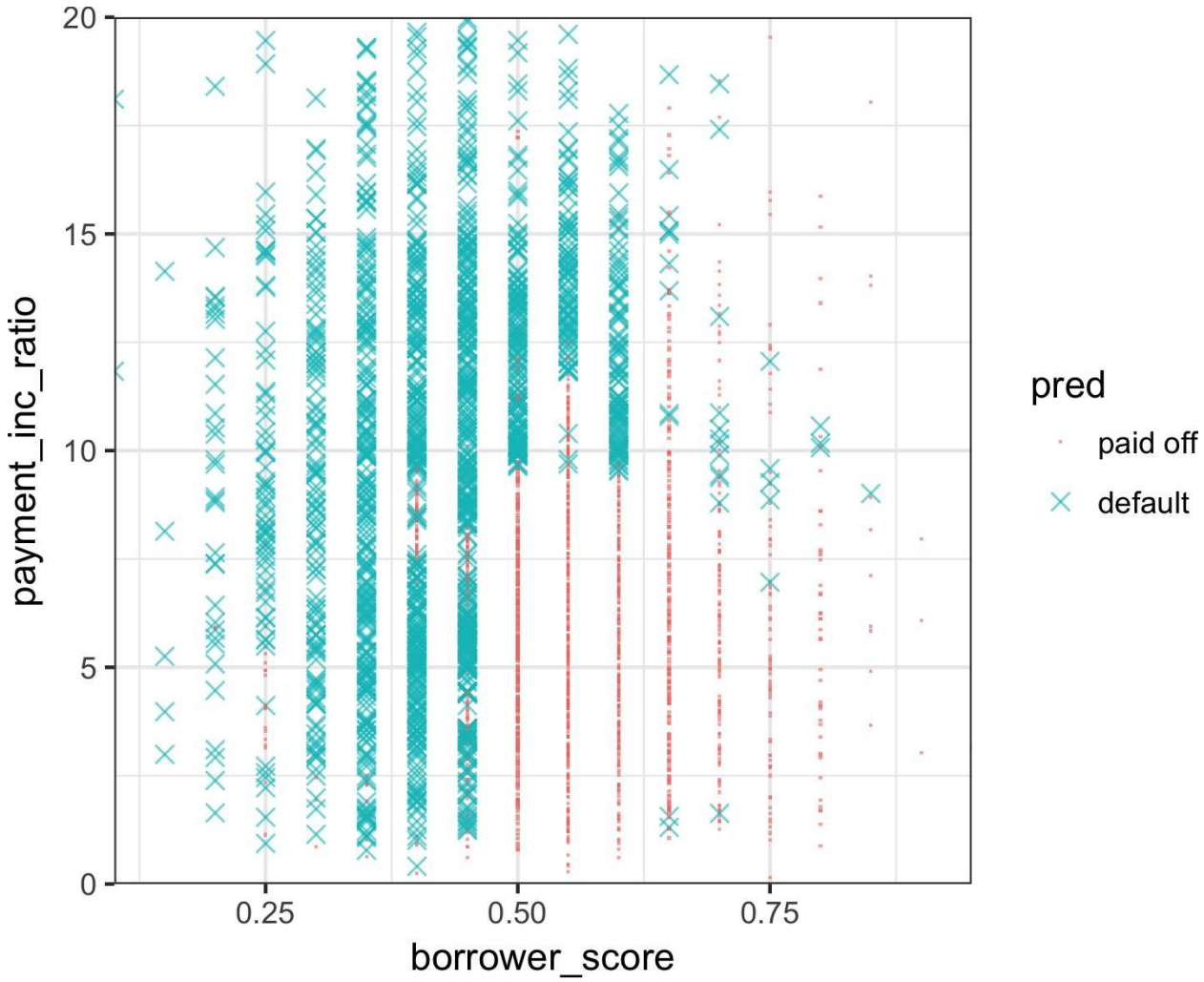

Figure 6-7. The predicted outcomes from the random forest applied to the loan default data

The random forest method is a “black box” method. It produces more accurate predictions than a simple tree, but the simple tree’s intuitive decision rules are lost. The predictions are also somewhat noisy: note that some borrowers with a very high score, indicating high creditworthiness, still end up with a prediction of default. This is a result of some unusual records in the data and demonstrates the danger of overfitting by the random forest (see Bias-Variance Tradeoff).

Variable Importance

The power of the random forest algorithm shows itself when you build predictive models for data with many features and records. It has the ability to automatically determine which predictors are important and discover complex relationships between predictors corresponding to interaction terms (see "Interactions and Main Effects"). For example, fit a model to the loan default data with all columns included:

```
> rf_all <- randomForest(outcome ~ ., data=loan_data, importance=TRUE)
> rf_all

Call:
 randomForest(formula = outcome ~ ., data = loan_data, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 34.38%
Confusion matrix:
         paid off default class.error
paid off    15078    8058   0.3482884
default      7849   15287   0.3392548
```

The argument `importance=TRUE` requests that the `randomForest` store additional information about the importance of different variables. The function `varImpPlot` will plot the relative performance of the variables:

```
varImpPlot(rf_all, type=1)
varImpPlot(rf_all, type=2)
```

The result is shown in Figure 6-8.

Saishna Budhathoki

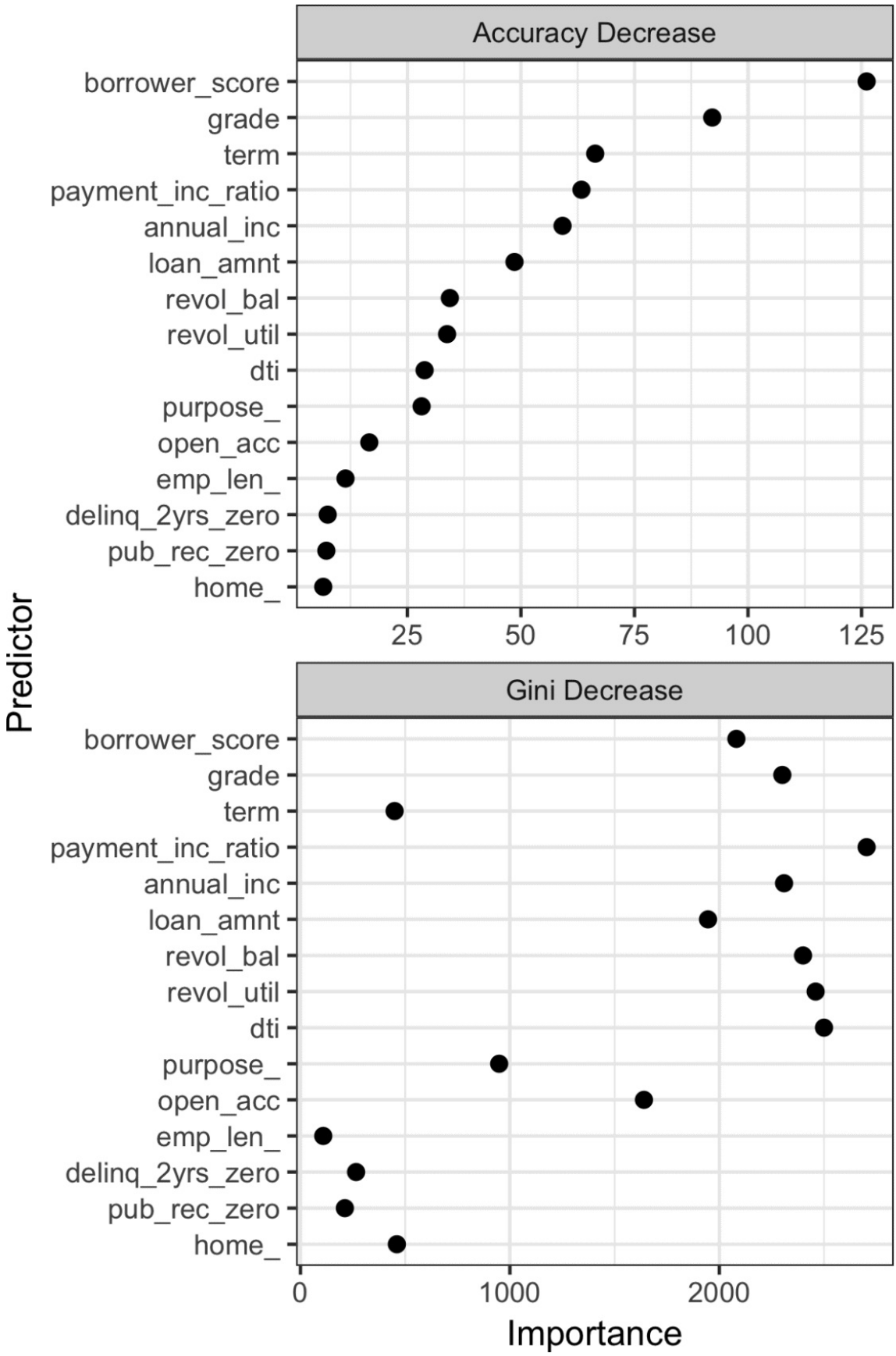

Figure 6-8. The importance of variables for the full model fit to the loan data

There are two ways to measure variable importance:

- By the decrease in accuracy of the model if the values of a variable are randomly permuted (`type=1`). Randomly permuting the values has the effect of removing all predictive power for that variable. The accuracy is computed from the out-of-bag data (so this measure is effectively a cross-validated estimate).

- By the mean decrease in the Gini impurity score (see "Measuring Homogeneity or Impurity") for all of the nodes that were split on a variable (`type=2`). This measures how much improvement to the purity of the nodes that variable contributes. This measure is based on the training set, and therefore less reliable than a measure calculated on out-of-bag data.

The top and bottom panels of Figure 6-8 show variable importance according to the decrease in accuracy and in Gini impurity, respectively. The variables in both panels are ranked by the decrease in accuracy. The variable importance scores produced by these two measures are quite different.

Since the accuracy decrease is a more reliable metric, why should we use the Gini impurity decrease measure? By default, `randomForest` only computes this Gini impurity: Gini impurity is a byproduct of the algorithm, whereas model accuracy by variable requires extra computations (randomly permuting the data and predicting this data). In cases where computational complexity is important, such as in a production setting where thousands of models are being fit, it may not be worth the extra computational effort. In addition, the Gini decrease sheds light on which variables the random forest is using to make its splitting rules (recall that this information, readily visible in a simple tree, is effectively lost in a random forest). Examining the difference between Gini decrease and model accuracy variable importance may suggest ways to improve the model.

Hyperparameters

The random forest, as with many statistical machine learning algorithms, can be considered a black-box algorithm with knobs to adjust how the box works. These knobs are called *hyperparameters*, which are parameters that you need to set before fitting a model; they are not optimized as part of the training process. While traditional statistical models require choices (e.g., the choice of predictors to use in a regression model), the hyperparameters for random forest are more critical, especially to avoid overfitting. In particular, the two most important hyperparemters for the random forest are:

`nodesize`
: The minimum size for terminal nodes (leaves in the tree). The default is 1 for classification and 5 for regression.

`maxnodes`
: The maximum number of nodes in each decision tree. By default, there is no limit and the largest tree will be fit subject to the constraints of `nodesize`.

It may be tempting to ignore these parameters and simply go with the default values. However, using the default may lead to overfitting when you apply the random forest to noisy data. When you increase `nodesize` or set `maxnodes`, the algorithm will fit smaller trees and is less likely to create spurious predictive rules. Cross-validation (see "Cross-Validation") can be used to test the effects of setting different values for hyperparameters.

KEY IDEAS FOR BAGGING AND THE RANDOM FOREST

- Ensemble models improve model accuracy by combining the results from many models.
- Bagging is a particular type of ensemble model based on fitting many models to bootstrapped samples of the data and averaging the models.
- Random forest is a special type of bagging applied to decision trees. In addition to resampling the data, the random forest algorithm samples the predictor variables when splitting the trees.
- A useful output from the random forest is a measure of variable importance that ranks the predictors in terms of their contribution to model accuracy.
- The random forest has a set of hyperparameters that should be tuned using cross-validation to avoid overfitting.

Saishna Budhathoki

Boosting

Ensemble models have become a standard tool for predictive modeling. *Boosting* is a general technique to create an ensemble of models. It was developed around the same time as *bagging* (see "Bagging and the Random Forest"). Like bagging, boosting is most commonly used with decision trees. Despite their similarities, boosting takes a very different approach — one that comes with many more bells and whistles. As a result, while bagging can be done with relatively little tuning, boosting requires much greater care in its application. If these two methods were cars, bagging could be considered a Honda Accord (reliable and steady), whereas boosting could be considered a Porsche (powerful but requires more care).

In linear regression models, the residuals are often examined to see if the fit can be improved (see "Partial Residual Plots and Nonlinearity"). Boosting takes this concept much further and fits a series of models with each successive model fit to minimize the error of the previous models. Several variants of the algorithm are commonly used: *Adaboost*, *gradient boosting*, and *stochastic gradient boosting*. The latter, stochastic gradient boosting, is the most general and widely used. Indeed, with the right choice of parameters, the algorithm can emulate the random forest.

KEY TERMS FOR BOOSTING

Ensemble
: Forming a prediction by using a collection of models.

Synonym
: Model averaging

Boosting
: A general technique to fit a sequence of models by giving more weight to the records with large residuals for each successive round.

Adaboost
: An early version of boosting based on reweighting the data based on the residuals.

Gradient boosting
: A more general form of boosting that is cast in terms of minimizing a cost function.

Stochastic gradient boosting
: The most general algorithm for boosting that incorporates resampling of records and columns in

each round.

Regularization

A technique to avoid overfitting by adding a penalty term to the cost function on the number of parameters in the model.

Hyperparameters

Parameters that need to be set before fitting the algorithm.

The Boosting Algorithm

The basic idea behind the various boosting algorithms is essentially the same. The easiest to understand is Adaboost, which proceeds as follows:

1. Initialize *M*, the maximum number of models to be fit, and set the iteration counter $m = 1$. Initialize the observation weights $w_i = 1 / N$ for $i = 1, 2, \ldots, N$. Initialize the ensemble model $\hat{F}_0 = 0$.

2. Train a model using $\hat{f}_m$ using the observation weights $w_1, w_2, \ldots, w_N$ that minimizes the weighted error e_m defined by summing the weights for the misclassified observations.

3. Add the model to the ensemble: $\hat{F}_m = \hat{F}_{m-1} + \alpha_m \hat{f}_m$ where $\alpha_m = \frac{\log 1 - e_m}{e_m}$.

4. Update the weights $w_1, w_2, \ldots, w_N$ so that the weights are increased for the observations that were misclassfied. The size of the increase depends on α_m with larger values of α_m leading to bigger weights.

5. Increment the model counter $m = m + 1$. If $m \leq M$, go to step 1.

The boosted estimate is given by:

$$\hat{F} = \alpha_1 \hat{f}_1 + \alpha_2 \hat{f}_2 + \cdots + \alpha_M \hat{f}_M$$

By increasing the weights for the observations that were misclassified, the algorithm forces the models to train more heavily on the data for which it performed poorly. The factor α_m ensures that models with lower error have a bigger weight.

Gradient boosting is similar to Adaboost but casts the problem as an optimization of a cost function. Instead of adjusting weights, gradient boosting fits models to a *pseudo-residual*, which has the effect of training more heavily on the larger residuals. In the spirit of the random forest, stochastic gradient boosting adds randomness to the algorithm by sampling observations and predictor variables at each stage.

XGBoost

The most widely used public domain software for boosting is XGBoost, an implementation of stochastic gradient boosting originally developed by Tianqi Chen and Carlos Guestrin at the University of Washington. A computationally efficient implementation with many options, it is available as a package for most major data science software languages. In R, XGBoost is available as the package `xgboost`.

The function `xgboost` has many parameters that can, and should, be adjusted (see "Hyperparameters and Cross-Validation"). Two very important parameters are `subsample`, which controls the fraction of observations that should be sampled at each iteration, and `eta`, a shrinkage factor applied to α_m in the boosting algorithm (see "The Boosting Algorithm"). Using `subsample` makes boosting act like the random forest except that the sampling is done without replacement. The shrinkage parameter `eta` is helpful to prevent overfitting by reducing the change in the weights (a smaller change in the weights means the algorithm is less likely to overfit to the training set). The following applies `xgboost` to the loan data with just two predictor variables:

```
library(xgboost)
predictors <- data.matrix(loan3000[, c('borrower_score',
                                       'payment_inc_ratio')])
label <- as.numeric(loan3000[,'outcome'])-1
xgb <- xgboost(data=predictors, label=label,
               objective = "binary:logistic",
               params=list(subsample=.63, eta=0.1), nrounds=100)
```

Note that `xgboost` does not support the formula syntax, so the predictors need to be converted to a `data.matrix` and the response needs to be converted to 0/1 variables. The `objective` argument tells `xgboost` what type of problem this is; based on this, `xgboost` will choose a metric to optimize.

The predicted values can be obtained from the `predict` function and, since there are only two variables, plotted versus the predictors:

```
pred <- predict(xgb, newdata=predictors)
xgb_df <- cbind(loan3000, pred_default=pred>.5, prob_default=pred)
ggplot(data=xgb_df, aes(x=borrower_score, y=payment_inc_ratio,
                        color=pred_default, shape=pred_default)) +
```

```
        geom_point(alpha=.6, size=2)
```

The result is shown in Figure 6-9. Qualitatively, this is similar to the predictions from the random forest; see Figure 6-7. The predictions are somewhat noisy in that some borrowers with a very high borrower score still end up with a prediction of default.

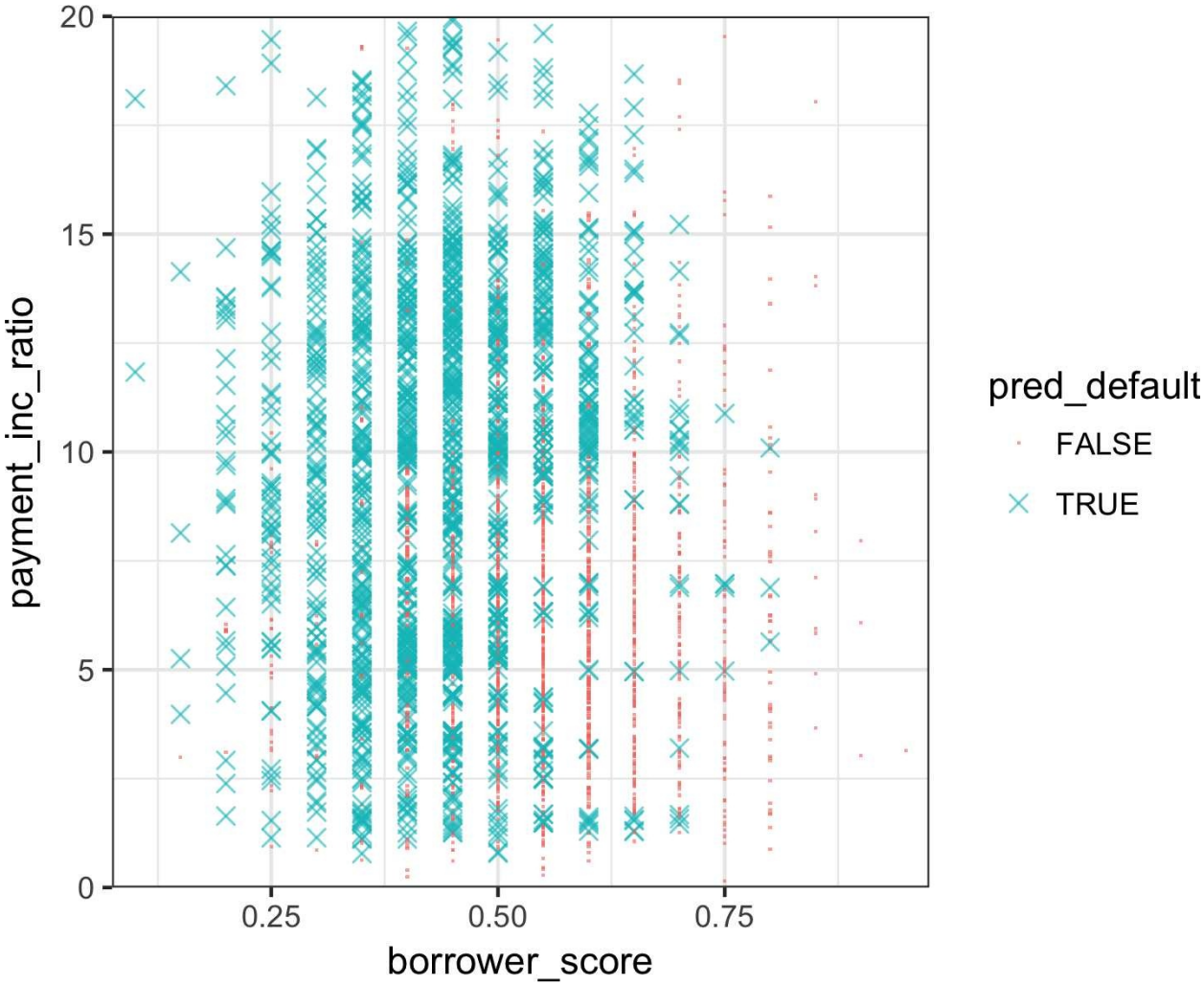

Figure 6-9. The predicted outcomes from XGBoost applied to the loan default data

Regularization: Avoiding Overfitting

Blind application of `xgboost` can lead to unstable models as a result of *overfitting* to the training data. The problem with overfitting is twofold:

- The accuracy of the model on new data not in the training set will be degraded.
- The predictions from the model are highly variable, leading to unstable results.

Any modeling technique is potentially prone to overfitting. For example, if too many variables are included in a regression equation, the model may end up with spurious predictions. However, for most statistical techniques, overfitting can be avoided by a judicious selection of predictor variables. Even the random forest generally produces a reasonable model without tuning the parameters. This, however, is not the case for `xgboost`. Fit `xgboost` to the loan data for a training set with all of the variables included in the model:

```
> predictors <- data.matrix(loan_data[,-which(names(loan_data) %in%
                                        'outcome')])
> label <- as.numeric(loan_data$outcome)-1
> test_idx <- sample(nrow(loan_data), 10000)
> xgb_default <- xgboost(data=predictors[-test_idx,],
                         label=label[-test_idx],
                         objective = "binary:logistic", nrounds=250)
> pred_default <- predict(xgb_default, predictors[test_idx,])
> error_default <- abs(label[test_idx] - pred_default) > 0.5
> xgb_default$evaluation_log[250,]
   iter train_error
1:  250    0.145622
> mean(error_default)
[1] 0.3715
```

The test set consists of 10,000 randomly sampled records from the full data, and the training set consists of the remaining records. Boosting leads to an error rate of only 14.6% for the training set. The test set, however, has a much higher error rate of 36.2%. This is a result of overfitting: while boosting can explain the variability in the training set very well, the prediction rules do not apply to new data.

Boosting provides several parameters to avoid overfitting, including the parameters `eta` and `subsample` (see “XGBoost”). Another approach is

regularization, a technique that modifies the cost function in order to *penalize* the complexity of the model. Decision trees are fit by minimizing cost criteria such as Gini's impurity score (see "Measuring Homogeneity or Impurity"). In `xgboost`, it is possible to modify the cost function by adding a term that measures the complexity of the model.

There are two parameters in `xgboost` to regularize the model: `alpha` and `lambda`, which correspond to Manhattan distance and squared Euclidean distance, respectively (see "Distance Metrics"). Increasing these parameters will penalize more complex models and reduce the size of the trees that are fit. For example, see what happens if we set `lambda` to 1,000:

```
> xgb_penalty <- xgboost(data=predictors[-test_idx,],
                         label=label[-test_idx],
                         params=list(eta=.1, subsample=.63, lambda=1000),
                         objective = "binary:logistic", nrounds=250)
> pred_penalty <- predict(xgb_penalty, predictors[test_idx,])
> error_penalty <- abs(label[test_idx] - pred_penalty) > 0.5
> xgb_penalty$evaluation_log[250,]
   iter train_error
1:  250    0.332405
> mean(error_penalty)
[1] 0.3483
```

Now the training error is only slightly lower than the error on the test set.

The `predict` method offers a convenient argument, `ntreelimit`, that forces only the first *i* trees to be used in the prediction. This lets us directly compare the in-sample versus out-of-sample error rates as more models are included:

```
> error_default <- rep(0, 250)
> error_penalty <- rep(0, 250)
> for(i in 1:250){
  pred_def <- predict(xgb_default, predictors[test_idx,], ntreelimit=i)
  error_default[i] <- mean(abs(label[test_idx] - pred_def) >= 0.5)
  pred_pen <- predict(xgb_penalty, predictors[test_idx,], ntreelimit = i)
  error_penalty[i] <- mean(abs(label[test_idx] - pred_pen) >= 0.5)
}
```

The output from the model returns the error for the training set in the component `xgb_default$evaluation_log`. By combining this with the out-of-sample errors, we can plot the errors versus the number of iterations:

```
> errors <- rbind(xgb_default$evaluation_log,
                  xgb_penalty$evaluation_log,
                  data.frame(iter=1:250, train_error=error_default),
```

```
                data.frame(iter=1:250, train_error=error_penalty))
> errors$type <- rep(c('default train', 'penalty train',
                       'default test', 'penalty test'), rep(250, 4))
> ggplot(errors, aes(x=iter, y=train_error, group=type)) +
  geom_line(aes(linetype=type, color=type))
```

The result, displayed in Figure 6-10, shows how the default model steadily improves the accuracy for the training set but actually gets worse for the test set. The penalized model does not exhibit this behavior.

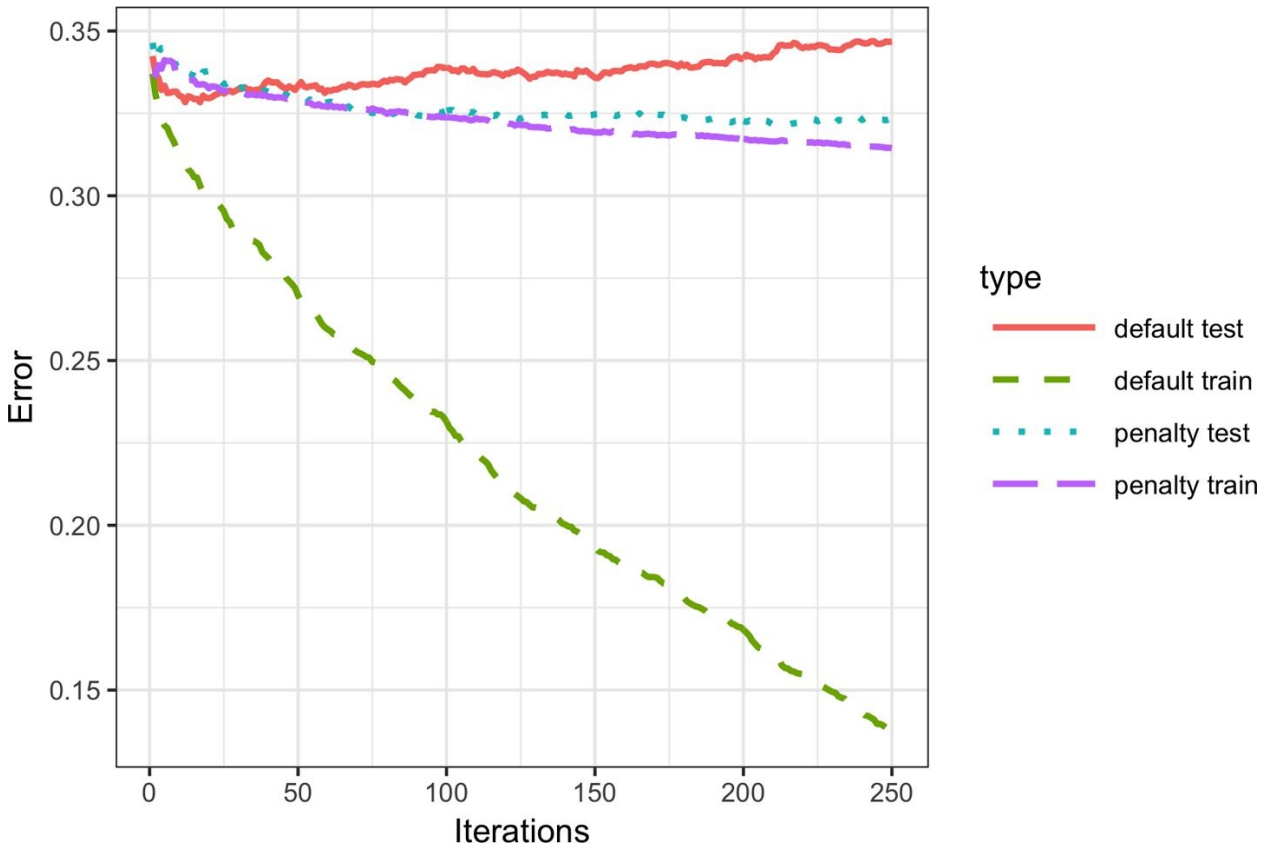

Figure 6-10. The error rate of the default XGBoost versus a penalized version of XGBoost

RIDGE REGRESSION AND THE LASSO

Adding a penalty on the complexity of a model to help avoid overfitting dates back to the 1970s. Least squares regression minimizes the residual sum of squares (RSS); see "Least Squares". *Ridge regression* minimizes the sum of squared residuals plus a penalty on the number and size of the coefficients:

$$\sum_{i=1}^{n}\left(Y_i - \hat{b}_0 - \hat{b}_1 X_i - \cdots \hat{b} X_p\right)^2 + \lambda\left(\hat{b}_1^2 + \cdots + \hat{b}_p^2\right)$$

The value of λ determines how much the coefficients are penalized; larger values produce models that are less likely to overfit the data. The *Lasso* is similar, except that it uses Manhattan distance instead of Euclidean distance as a penalty term:

$$\sum_{i=1}^{n}\left(Y_i - \hat{b}_0 - \hat{b}_1 X_i - \cdots \hat{b} X_p\right)^2 + \alpha\left(|\hat{b}_1| + \cdots + |\hat{b}_p|\right)$$

The `xgboost` parameters `lambda` and `alpha` are acting in a similar mannger.

Hyperparameters and Cross-Validation

`xgboost` has a daunting array of hyperparameters; see "XGBoost Hyperparameters" for a discussion. As seen in "Regularization: Avoiding Overfitting", the specific choice can dramatically change the model fit. Given a huge combination of hyperparameters to choose from, how should we be guided in our choice? A standard solution to this problem is to use *cross-validation*; see "Cross-Validation". Cross-validation randomly splits up the data into *K* different groups, also called *folds*. For each fold, a model is trained on the data not in the fold and then evaluated on the data in the fold. This yields a measure of accuracy of the model on out-of-sample data. The best set of hyperparameters is the one given by the model with the lowest overall error as computed by averaging the errors from each of the folds.

To illustrate the technique, we apply it to parameter selection for `xgboost`. In this example, we explore two parameters: the shrinkage parameter `eta` (see "XGBoost") and the maximum depth of trees `max_depth`. The parameter `max_depth` is the maximum depth of a leaf node to the root of the tree with a default value of 6. This gives us another way to control overfitting: deep trees tend to be more complex and may overfit the data. First we set up the folds and parameter list:

```
> N <- nrow(loan_data)
> fold_number <- sample(1:5, N, replace = TRUE)
> params <- data.frame(eta = rep(c(.1, .5, .9), 3),
                       max_depth = rep(c(3, 6, 12), rep(3,3)))
```

Now we apply the preceding algorithm to compute the error for each model and each fold using five folds:

```
> error <- matrix(0, nrow=9, ncol=5)
> for(i in 1:nrow(params)){
>   for(k in 1:5){
>     fold_idx <- (1:N)[fold_number == k]
>     xgb <- xgboost(data=predictors[-fold_idx,], label=label[-fold_idx],
                     params = list(eta = params[i, 'eta'],
                                   max_depth = params[i, 'max_depth']),
                   objective = "binary:logistic", nrounds=100, verbose=0)
>     pred <- predict(xgb, predictors[fold_idx,])
>     error[i, k] <- mean(abs(label[fold_idx] - pred) >= 0.5)
>   }
> }
```

Since we are fitting 45 total models, this can take a while. The errors are stored as a matrix with the models along the rows and folds along the columns. Using the function `rowMeans`, we can compare the error rate for the different parameter sets:

```
> avg_error <- 100 * rowMeans(error)
> cbind(params, avg_error)
  eta max_depth avg_error
1 0.1         3     35.41
2 0.5         3     35.84
3 0.9         3     36.48
4 0.1         6     35.37
5 0.5         6     37.33
6 0.9         6     39.41
7 0.1        12     36.70
8 0.5        12     38.85
9 0.9        12     40.19
```

Cross-validation suggests that using shallower trees with a smaller value of `eta` yields more accurate results. Since these models are also more stable, the best parameters to use are `eta=0.1` and `max_depth=3` (or possibly `max_depth=6`).

XGBOOST HYPERPARAMETERS

The hyperparameters for `xgboost` are primarily used to balance overfitting with the accuracy and computational complexity. For a complete discussion of the parameters, refer to the `xgboost` documentation.

`eta`

The shrinkage factor between 0 and 1 applied to α in the boosting algorithm. The default is 0.3, but for noisy data, smaller values are recommended (e.g., 0.1).

`nrounds`

The number of boosting rounds. If `eta` is set to a small value, it is important to increase the number of rounds since the algorithm learns more slowly. As long as some parameters are included to prevent overfitting, having more rounds doesn't hurt.

`max_depth`

The maximum depth of the tree (the default is 6). In contrast to the random forest, which fits very deep trees, boosting usually fits shallow trees. This has the advantage of avoiding spurious complex interactions in the model that can arise from noisy data.

`subsample` and `colsample_bytree`

Fraction of the records to sample without replacement and the fraction of predictors to sample for use in fitting the trees. These parameters, which are similar to those in random forests, help avoid overfitting.

`lambda` and `alpha`

The regularization parameters to help control overfitting (see "Regularization: Avoiding Overfitting").

KEY IDEAS FOR BOOSTING

- Boosting is a class of ensemble models based on fitting a sequence of models, with more weight given to records with large errors in successive rounds.
- Stochastic gradient boosting is the most general type of boosting and offers the best performance. The most common form of stochastic gradient boosting uses tree models.
- XGBoost is a popular and computationally efficient software package for stochastic gradient boosting; it is available in all common languages used in data science.
- Boosting is prone to overfitting the data, and the hyperparameters need to be tuned to avoid this.
- Regularization is one way to avoid overfitting by including a penalty term on the number of parameters (e.g., tree size) in a model.
- Cross-validation is especially important for boosting due to the large number of hyperparameters that need to be set.

Summary

This chapter describes two classification and prediction methods that "learn" flexibly and locally from data, rather than starting with a structural model (e.g., a linear regression) that is fit to the entire data set. *K*-Nearest Neighbors is a simple process that simply looks around at similar records and assigns their majority class (or average value) to the record being predicted. Trying various cutoff (split) values of predictor variables, tree models iteratively divide the data into sections and subsections that are increasingly homogeneous with respect to class. The most effective split values form a path, and also a "rule," to a classification or prediction. Tree models are a very powerful and popular predictive tool, often outperforming other methods. They have given rise to various ensemble methods (random forests, boosting, bagging) that sharpen the predictive power of trees.

---

1 This and subsequent sections in this chapter © 2017 Datastats, LLC, Peter Bruce and Andrew Bruce, used by permission.

2 The term CART is a registered trademark of Salford Systems related to their specific implementation of tree models.

3 The term *random forest* is a trademark of Leo Breiman and Adele Cutler and licensed to Salford Systems. There is no standard nontrademark name, and the term random forest is as synonymous with the algorithm as Kleenex is with facial tissues.