that has the largest contribution to R^2, stopping when the contribution is no longer statistically significant. In backward selection, or *backward elimination*, you start with the full model and take away predictors that are not statistically significant until you are left with a model in which all predictors are statistically significant.

Penalized regression is similar in spirit to AIC. Instead of explicitly searching through a discrete set of models, the model-fitting equation incorporates a constraint that penalizes the model for too many variables (parameters). Rather than eliminating predictor variables entirely—as with stepwise, forward, and backward selection—penalized regression applies the penalty by reducing coefficients, in some cases to near zero. Common penalized regression methods are *ridge regression* and *lasso regression*.

Stepwise regression and all subset regression are *in-sample* methods to assess and tune models. This means the model selection is possibly subject to overfitting and may not perform as well when applied to new data. One common approach to avoid this is to use cross-validation to validate the models. In linear regression, overfitting is typically not a major issue, due to the simple (linear) global structure imposed on the data. For more sophisticated types of models, particularly iterative procedures that respond to local data structure, cross-validation is a very important tool; see "Cross-Validation" on page 138 for details.

Weighted Regression

Weighted regression is used by statisticians for a variety of purposes; in particular, it is important for analysis of complex surveys. Data scientists may find weighted regression useful in two cases:

- Inverse-variance weighting when different observations have been measured with different precision.
- Analysis of data in an aggregated form such that the weight variable encodes how many original observations each row in the aggregated data represents.

For example, with the housing data, older sales are less reliable than more recent sales. Using the `DocumentDate` to determine the year of the sale, we can compute a `Weight` as the number of years since 2005 (the beginning of the data).

```
library(lubridate)
house$Year = year(house$DocumentDate)
house$Weight = house$Year - 2005
```

We can compute a weighted regression with the `lm` function using the `weight` argument.

```
house_wt <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
                Bedrooms + BldgGrade,
```

```
                data=house, weight=Weight)
round(cbind(house_lm=house_lm$coefficients,
            house_wt=house_wt$coefficients), digits=3)

                   house_lm    house_wt
(Intercept)    -521924.722 -584265.244
SqFtTotLiving      228.832     245.017
SqFtLot             -0.061      -0.292
Bathrooms       -19438.099  -26079.171
Bedrooms        -47781.153  -53625.404
BldgGrade       106117.210  115259.026
```

The coefficents in the weighted regression are slightly different from the original regression.

Key Ideas

- Multiple linear regression models the relationship between a response variable Y and multiple predictor variables $X_1, ..., X_p$.
- The most important metrics to evaluate a model are root mean squared error (RMSE) and R-squared (R^2).
- The standard error of the coefficients can be used to measure the reliability of a variable's contribution to a model.
- Stepwise regression is a way to automatically determine which variables should be included in the model.
- Weighted regression is used to give certain records more or less weight in fitting the equation.

Prediction Using Regression

The primary purpose of regression in data science is prediction. This is useful to keep in mind, since regression, being an old and established statistical method, comes with baggage that is more relevant to its traditional explanatory modeling role than to prediction.

Key Terms for Prediction Using Regression

Prediction interval
: An uncertainty interval around an individual predicted value.

Extrapolation
: Extension of a model beyond the range of the data used to fit it.

The Dangers of Extrapolation

Regression models should not be used to extrapolate beyond the range of the data. The model is valid only for predictor values for which the data has sufficient values (even in the case that sufficient data is available, there could be other problems: see "Testing the Assumptions: Regression Diagnostics" on page 155). As an extreme case, suppose `model_lm` is used to predict the value of a 5,000-square-foot empty lot. In such a case, all the predictors related to the building would have a value of 0 and the regression equation would yield an absurd prediction of –521,900 + 5,000 × –.0605 = –$522,202. Why did this happen? The data contains only parcels with buildings—there are no records corresponding to vacant land. Consequently, the model has no information to tell it how to predict the sales price for vacant land.

Confidence and Prediction Intervals

Much of statistics involves understanding and measuring variability (uncertainty). The t-statistics and p-values reported in regression output deal with this in a formal way, which is sometimes useful for variable selection (see "Assessing the Model" on page 136). More useful metrics are confidence intervals, which are uncertainty intervals placed around regression coefficients and predictions. An easy way to understand this is via the bootstrap (see "The Bootstrap" on page 57 for more details about the general bootstrap procedure). The most common regression confidence intervals encountered in software output are those for regression parameters (coefficients). Here is a bootstrap algorithm for generating confidence intervals for regression parameters (coefficients) for a data set with P predictors and n records (rows):

1. Consider each row (including outcome variable) as a single "ticket" and place all the n tickets in a box.
2. Draw a ticket at random, record the values, and replace it in the box.
3. Repeat step 2 n times; you now have one bootstrap resample.
4. Fit a regression to the bootstrap sample, and record the estimated coefficients.
5. Repeat steps 2 through 4, say, 1,000 times.
6. You now have 1,000 bootstrap values for each coefficient; find the appropriate percentiles for each one (e.g., 5th and 95th for a 90% confidence interval).

You can use the `Boot` function in R to generate actual bootstrap confidence intervals for the coefficients, or you can simply use the formula-based intervals that are a routine R output. The conceptual meaning and interpretation are the same, and not of central importance to data scientists, because they concern the regression coefficients. Of greater interest to data scientists are intervals around predicted y values ($\hat{Y}_i$). The uncertainty around $\hat{Y}_i$ comes from two sources:

- Uncertainty about what the relevant predictor variables and their coefficients are (see the preceding bootstrap algorithm)
- Additional error inherent in individual data points

The individual data point error can be thought of as follows: even if we knew for certain what the regression equation was (e.g., if we had a huge number of records to fit it), the *actual* outcome values for a given set of predictor values will vary. For example, several houses—each with 8 rooms, a 6,500 square foot lot, 3 bathrooms, and a basement—might have different values. We can model this individual error with the residuals from the fitted values. The bootstrap algorithm for modeling both the regression model error and the individual data point error would look as follows:

1. Take a bootstrap sample from the data (spelled out in greater detail earlier).
2. Fit the regression, and predict the new value.
3. Take a single residual at random from the original regression fit, add it to the predicted value, and record the result.
4. Repeat steps 1 through 3, say, 1,000 times.
5. Find the 2.5th and the 97.5th percentiles of the results.

Prediction Interval or Confidence Interval?

A prediction interval pertains to uncertainty around a single value, while a confidence interval pertains to a mean or other statistic calculated from multiple values. Thus, a prediction interval will typically be much wider than a confidence interval for the same value. We model this individual value error in the bootstrap model by selecting an individual residual to tack on to the predicted value. Which should you use? That depends on the context and the purpose of the analysis, but, in general, data scientists are interested in specific individual predictions, so a prediction interval would be more appropriate. Using a confidence interval when you should be using a prediction interval will greatly underestimate the uncertainty in a given predicted value.

Key Ideas

- Extrapolation beyond the range of the data can lead to error.
- Confidence intervals quantify uncertainty around regression coefficients.
- Prediction intervals quantify uncertainty in individual predictions.

- Most software, R included, will produce prediction and confidence intervals in default or specified output, using formulas.
- The bootstrap can also be used; the interpretation and idea are the same.

Factor Variables in Regression

Factor variables, also termed *categorical* variables, take on a limited number of discrete values. For example, a loan purpose can be "debt consolidation," "wedding," "car," and so on. The binary (yes/no) variable, also called an *indicator* variable, is a special case of a factor variable. Regression requires numerical inputs, so factor variables need to be recoded to use in the model. The most common approach is to convert a variable into a set of binary *dummy* variables.

Key Terms for Factor Variables

Dummy variables
: Binary 0–1 variables derived by recoding factor data for use in regression and other models.

Reference coding
: The most common type of coding used by statisticians, in which one level of a factor is used as a reference and other factors are compared to that level.

Synonyms
: treatment coding

One hot encoder
: A common type of coding used in the machine learning community in which all factors levels are retained. While useful for certain machine learning algorithms, this approach is not appropriate for multiple linear regression.

Deviation coding
: A type of coding that compares each level against the overall mean as opposed to the reference level.

Synonyms
: sum contrasts

Dummy Variables Representation

In the King County housing data, there is a factor variable for the property type; a small subset of six records is shown below.

```
head(house[, 'PropertyType'])
Source: local data frame [6 x 1]

   PropertyType
         (fctr)
1     Multiplex
2 Single Family
3 Single Family
4 Single Family
5 Single Family
6     Townhouse
```

There are three possible values: `Multiplex`, `Single Family`, and `Townhouse`. To use this factor variable, we need to convert it to a set of binary variables. We do this by creating a binary variable for each possible value of the factor variable. To do this in R, we use the `model.matrix` function:[3]

```
prop_type_dummies <- model.matrix(~PropertyType -1, data=house)
head(prop_type_dummies)
  PropertyTypeMultiplex PropertyTypeSingle Family PropertyTypeTownhouse
1                     1                         0                     0
2                     0                         1                     0
3                     0                         1                     0
4                     0                         1                     0
5                     0                         1                     0
6                     0                         0                     1
```

The function `model.matrix` converts a data frame into a matrix suitable to a linear model. The factor variable `PropertyType`, which has three distinct levels, is represented as a matrix with three columns. In the machine learning community, this representation is referred to as *one hot encoding* (see "One Hot Encoder" on page 214). In certain machine learning algorithms, such as nearest neighbors and tree models, one hot encoding is the standard way to represent factor variables (for example, see "Tree Models" on page 219).

In the regression setting, a factor variable with P distinct levels is usually represented by a matrix with only $P - 1$ columns. This is because a regression model typically includes an intercept term. With an intercept, once you have defined the values for $P - 1$ binaries, the value for the Pth is known and could be considered redundant. Adding the Pth column will cause a multicollinearity error (see "Multicollinearity" on page 151).

The default representation in R is to use the first factor level as a *reference* and interpret the remaining levels relative to that factor.

3 The `-1` argument in the `model.matrix` produces one hot encoding representation (by removing the intercept, hence the "-"). Otherwise, the default in R is to produce a matrix with $P - 1$ columns with the first factor level as a reference.

```
lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
+      Bedrooms +  BldgGrade + PropertyType, data=house)

Call:
lm(formula = AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
    Bedrooms + BldgGrade + PropertyType, data = house)

Coefficients:
              (Intercept)             SqFtTotLiving
                -4.469e+05                 2.234e+02
                   SqFtLot                Bathrooms
                -7.041e-02                -1.597e+04
                  Bedrooms                BldgGrade
                -5.090e+04                 1.094e+05
PropertyTypeSingle Family      PropertyTypeTownhouse
                -8.469e+04                -1.151e+05
```

The output from the R regression shows two coefficients corresponding to `PropertyType`: `PropertyTypeSingle Family` and `PropertyTypeTownhouse`. There is no coefficient of `Multiplex` since it is implicitly defined when `PropertyTypeSingle Family == 0` and `PropertyTypeTownhouse == 0`. The coefficients are interpreted as relative to `Multiplex`, so a home that is `Single Family` is worth almost $85,000 less, and a home that is `Townhouse` is worth over $150,000 less.[4]

Different Factor Codings

There are several different ways to encode factor variables, known as *contrast coding* systems. For example, *deviation coding*, also know as *sum contrasts*, compares each level against the overall mean. Another contrast is *polynomial coding*, which is appropriate for ordered factors; see the section "Ordered Factor Variables" on page 149. With the exception of ordered factors, data scientists will generally not encounter any type of coding besides reference coding or one hot encoder.

Factor Variables with Many Levels

Some factor variables can produce a huge number of binary dummies—zip codes are a factor variable and there are 43,000 zip codes in the US. In such cases, it is useful to explore the data, and the relationships between predictor variables and the outcome, to determine whether useful information is contained in the categories. If so, you must further decide whether it is useful to retain all factors, or whether the levels should be consolidated.

4 This is unintuitive, but can be explained by the impact of location as a confounding variable; see "Confounding Variables" on page 152.

In King County, there are 82 zip codes with a house sale:

```
table(house$ZipCode)
```

```
 9800 89118 98001 98002 98003 98004 98005 98006 98007 98008 98010 98011
    1     1   358   180   241   293   133   460   112   291    56   163
98014 98019 98022 98023 98024 98027 98028 98029 98030 98031 98032 98033
   85   242   188   455    31   366   252   475   263   308   121   517
98034 98038 98039 98040 98042 98043 98045 98047 98050 98051 98052 98053
  575   788    47   244   641     1   222    48     7    32   614   499
98055 98056 98057 98058 98059 98065 98068 98070 98072 98074 98075 98077
  332   402     4   420   513   430     1    89   245   502   388   204
98092 98102 98103 98105 98106 98107 98108 98109 98112 98113 98115 98116
  289   106   671   313   361   296   155   149   357     1   620   364
98117 98118 98119 98122 98125 98126 98133 98136 98144 98146 98148 98155
  619   492   260   380   409   473   465   310   332   287    40   358
98166 98168 98177 98178 98188 98198 98199 98224 98288 98354
  193   332   216   266   101   225   393     3     4     9
```

`ZipCode` is an important variable, since it is a proxy for the effect of location on the value of a house. Including all levels requires 81 coefficients corresponding to 81 degrees of freedom. The original model `house_lm` has only 5 degress of freedom; see "Assessing the Model" on page 136. Moreover, several zip codes have only one sale. In some problems, you can consolidate a zip code using the first two or three digits, corresponding to a submetropolitan geographic region. For King County, almost all of the sales occur in 980xx or 981xx, so this doesn't help.

An alternative approach is to group the zip codes according to another variable, such as sale price. Even better is to form zip code groups using the residuals from an initial model. The following `dplyr` code consolidates the 82 zip codes into five groups based on the median of the residual from the `house_lm` regression:

```
zip_groups <- house %>%
  mutate(resid = residuals(house_lm)) %>%
  group_by(ZipCode) %>%
  summarize(med_resid = median(resid),
            cnt = n()) %>%
  arrange(med_resid) %>%
  mutate(cum_cnt = cumsum(cnt),
         ZipGroup = ntile(cum_cnt, 5))
house <- house %>%
  left_join(select(zip_groups, ZipCode, ZipGroup), by='ZipCode')
```

The median residual is computed for each zip and the `ntile` function is used to split the zip codes, sorted by the median, into five groups. See "Confounding Variables" on page 152 for an example of how this is used as a term in a regression improving upon the original fit.

The concept of using the residuals to help guide the regression fitting is a fundamental step in the modeling process; see "Testing the Assumptions: Regression Diagnostics" on page 155.

Ordered Factor Variables

Some factor variables reflect levels of a factor; these are termed *ordered factor variables* or *ordered categorical variables*. For example, the loan grade could be A, B, C, and so on—each grade carries more risk than the prior grade. Ordered factor variables can typically be converted to numerical values and used as is. For example, the variable `BldgGrade` is an ordered factor variable. Several of the types of grades are shown in Table 4-1. While the grades have specific meaning, the numeric value is ordered from low to high, corresponding to higher-grade homes. With the regression model `house_lm`, fit in "Multiple Linear Regression" on page 134, `BldgGrade` was treated as a numeric variable.

Table 4-1. A typical data format

Value	Description
1	Cabin
2	Substandard
5	Fair
10	Very good
12	Luxury
13	Mansion

Treating ordered factors as a numeric variable preserves the information contained in the ordering that would be lost if it were converted to a factor.

Key Ideas

- Factor variables need to be converted into numeric variables for use in a regression.
- The most common method to encode a factor variable with P distinct values is to represent them using P-1 dummy variables.
- A factor variable with many levels, even in very big data sets, may need to be consolidated into a variable with fewer levels.
- Some factors have levels that are ordered and can be represented as a single numeric variable.

Interpreting the Regression Equation

In data science, the most important use of regression is to predict some dependent (outcome) variable. In some cases, however, gaining insight from the equation itself to understand the nature of the relationship between the predictors and the outcome can be of value. This section provides guidance on examining the regression equation and interpreting it.

Key Terms for Interpreting the Regression Equation

Correlated variables
When the predictor variables are highly correlated, it is difficult to interpret the individual coefficients.

Multicollinearity
When the predictor variables have perfect, or near-perfect, correlation, the regression can be unstable or impossible to compute.

Synonyms
collinearity

Confounding variables
An important predictor that, when omitted, leads to spurious relationships in a regression equation.

Main effects
The relationship between a predictor and the outcome variable, independent from other variables.

Interactions
An interdependent relationship between two or more predictors and the response.

Correlated Predictors

In multiple regression, the predictor variables are often correlated with each other. As an example, examine the regression coefficients for the model `step_lm`, fit in "Model Selection and Stepwise Regression" on page 139:

```
step_lm$coefficients
              (Intercept)             SqFtTotLiving
             6.227632e+06              1.865012e+02
                Bathrooms                  Bedrooms
             4.472172e+04             -4.980718e+04
                BldgGrade PropertyTypeSingle Family
             1.391792e+05              2.332869e+04
  PropertyTypeTownhouse           SqFtFinBasement
```

```
 9.221625e+04              9.039911e+00
      YrBuilt
-3.592468e+03
```

The coefficient for `Bedrooms` is negative! This implies that adding a bedroom to a house will reduce its value. How can this be? This is because the predictor variables are correlated: larger houses tend to have more bedrooms, and it is the size that drives house value, not the number of bedrooms. Consider two homes of the exact same size: it is reasonable to expect that a home with more, but smaller, bedrooms would be considered less desirable.

Having correlated predictors can make it difficult to interpret the sign and value of regression coefficients (and can inflate the standard error of the estimates). The variables for bedrooms, house size, and number of bathrooms are all correlated. This is illustrated by the following example, which fits another regression removing the variables `SqFtTotLiving`, `SqFtFinBasement`, and `Bathrooms` from the equation:

```
update(step_lm, . ~ . -SqFtTotLiving - SqFtFinBasement - Bathrooms)

Call:
lm(formula = AdjSalePrice ~ Bedrooms + BldgGrade + PropertyType +
    YrBuilt, data = house0, na.action = na.omit)

Coefficients:
             (Intercept)                  Bedrooms
                 4834680                     27657
              BldgGrade  PropertyTypeSingle Family
                  245709                    -17604
    PropertyTypeTownhouse                   YrBuilt
                  -47477                     -3161
```

The `update` function can be used to add or remove variables from a model. Now the coefficient for bedrooms is positive—in line with what we would expect (though it is really acting as a proxy for house size, now that those variables have been removed).

Correlated variables are only one issue with interpreting regression coefficients. In `house_lm`, there is no variable to account for the location of the home, and the model is mixing together very different types of regions. Location may be a *confounding* variable; see "Confounding Variables" on page 152 for further discussion.

Multicollinearity

An extreme case of correlated variables produces multicollinearity—a condition in which there is redundance among the predictor variables. Perfect multicollinearity occurs when one predictor variable can be expressed as a linear combination of others. Multicollinearity occurs when:

- A variable is included multiple times by error.
- *P* dummies, instead of *P* – 1 dummies, are created from a factor variable (see "Factor Variables in Regression" on page 145).
- Two variables are nearly perfectly correlated with one another.

Multicollinearity in regression must be addressed—variables should be removed until the multicollinearity is gone. A regression does not have a well-defined solution in the presence of perfect multicollinearity. Many software packages, including R, automatically handle certain types of multicolliearity. For example, if `SqFtTotLiving` is included twice in the regression of the `house` data, the results are the same as for the `house_lm` model. In the case of nonperfect multicollinearity, the software may obtain a solution but the results may be unstable.

Multicollinearity is not such a problem for nonregression methods like trees, clustering, and nearest-neighbors, and in such methods it may be advisable to retain *P* dummies (instead of *P* – 1). That said, even in those methods, nonredundancy in predictor variables is still a virtue.

Confounding Variables

With correlated variables, the problem is one of commission: including different variables that have a similar predictive relationship with the response. With *confounding variables*, the problem is one of omission: an important variable is not included in the regression equation. Naive interpretation of the equation coefficients can lead to invalid conclusions.

Take, for example, the King County regression equation `house_lm` from "Example: King County Housing Data" on page 135. The regression coefficients of `SqFtLot`, `Bathrooms`, and `Bedrooms` are all negative. The original regression model does not contain a variable to represent location—a very important predictor of house price. To model location, include a variable `ZipGroup` that categorizes the zip code into one of five groups, from least expensive (1) to most expensive (5).[5]

5 There are 82 zip codes in King County, several with just a handful of sales. An alternative to directly using zip code as a factor variable, `ZipGroup` clusters similar zip codes into a single group. See "Factor Variables with Many Levels" on page 147 for details.

```
lm(AdjSalePrice ~  SqFtTotLiving + SqFtLot +
     Bathrooms + Bedrooms +
     BldgGrade + PropertyType + ZipGroup,
   data=house, na.action=na.omit)

Coefficients:
               (Intercept)              SqFtTotLiving
                -6.709e+05                  2.112e+02
                   SqFtLot                  Bathrooms
                 4.692e-01                  5.537e+03
                  Bedrooms                  BldgGrade
                -4.139e+04                  9.893e+04
PropertyTypeSingle Family      PropertyTypeTownhouse
                 2.113e+04                 -7.741e+04
                 ZipGroup2                  ZipGroup3
                 5.169e+04                  1.142e+05
                 ZipGroup4                  ZipGroup5
                 1.783e+05                  3.391e+05
```

`ZipGroup` is clearly an important variable: a home in the most expensive zip code group is estimated to have a higher sales price by almost $340,000. The coefficients of `SqFtLot` and `Bathrooms` are now positive and adding a bathroom increases the sale price by $7,500.

The coefficient for `Bedrooms` is still negative. While this is unintuitive, this is a well-known phenomenon in real estate. For homes of the same livable area and number of bathrooms, having more, and therefore smaller, bedrooms is associated with less valuable homes.

Interactions and Main Effects

Statisticians like to distinguish between *main effects*, or independent variables, and the *interactions* between the main effects. Main effects are what are often referred to as the *predictor variables* in the regression equation. An implicit assumption when only main effects are used in a model is that the relationship between a predictor variable and the response is independent of the other predictor variables. This is often not the case.

For example, the model fit to the King County Housing Data in "Confounding Variables" on page 152 includes several variables as main effects, including `ZipCode`. Location in real estate is everything, and it is natural to presume that the relationship between, say, house size and the sale price depends on location. A big house built in a low-rent district is not going to retain the same value as a big house built in an expensive area. You include interactions between variables in R using the `*` operator. For the King County data, the following fits an interaction between `SqFtTotLiving` and `ZipGroup`:

```
lm(AdjSalePrice ~  SqFtTotLiving*ZipGroup + SqFtLot +
     Bathrooms + Bedrooms + BldgGrade + PropertyType,
   data=house, na.action=na.omit)

 Coefficients:
               (Intercept)              SqFtTotLiving
                -4.919e+05                  1.176e+02
                 ZipGroup2                  ZipGroup3
                -1.342e+04                  2.254e+04
                 ZipGroup4                  ZipGroup5
                 1.776e+04                 -1.555e+05
                   SqFtLot                  Bathrooms
                 7.176e-01                 -5.130e+03
                  Bedrooms                  BldgGrade
                -4.181e+04                  1.053e+05
PropertyTypeSingle Family      PropertyTypeTownhouse
                 1.603e+04                 -5.629e+04
  SqFtTotLiving:ZipGroup2    SqFtTotLiving:ZipGroup3
                 3.165e+01                  3.893e+01
  SqFtTotLiving:ZipGroup4    SqFtTotLiving:ZipGroup5
                 7.051e+01                  2.298e+02
```

The resulting model has four new terms: `SqFtTotLiving:ZipGroup2`, `SqFtTotLiving:ZipGroup3`, and so on.

Location and house size appear to have a strong interaction. For a home in the lowest `ZipGroup`, the slope is the same as the slope for the main effect `SqFtTotLiving`, which is $177 per square foot (this is because R uses *reference* coding for factor variables; see "Factor Variables in Regression" on page 145). For a home in the highest `ZipGroup`, the slope is the sum of the main effect plus `SqFtTotLiving:ZipGroup5`, or $177 + $230 = $447 per square foot. In other words, adding a square foot in the most expensive zip code group boosts the predicted sale price by a factor of almost 2.7, compared to the boost in the least expensive zip code group.

Model Selection with Interaction Terms

In problems involving many variables, it can be challenging to decide which interaction terms should be included in the model. Several different approaches are commonly taken:

- In some problems, prior knowledge and intuition can guide the choice of which interaction terms to include in the model.
- Stepwise selection (see "Model Selection and Stepwise Regression" on page 139) can be used to sift through the various models.
- Penalized regression can automatically fit to a large set of possible interaction terms.
- Perhaps the most common approach is the use *tree models*, as well as their descendents, *random forest* and *gradient boosted trees*. This class of models automatically searches for optimal interaction terms; see "Tree Models" on page 219.

Key Ideas

- Because of correlation between predictors, care must be taken in the interpretation of the coefficients in multiple linear regression.
- Multicollinearity can cause numerical instability in fitting the regression equation.
- A confounding variable is an important predictor that is omitted from a model and can lead to a regression equation with spurious relationships.
- An interaction term between two variables is needed if the relationship between the variables and the response is interdependent.

Testing the Assumptions: Regression Diagnostics

In explanatory modeling (i.e., in a research context), various steps, in addition to the metrics mentioned previously (see "Assessing the Model" on page 136), are taken to assess how well the model fits the data. Most are based on analysis of the residuals, which can test the assumptions underlying the model. These steps do not directly address predictive accuracy, but they can provide useful insight in a predictive setting.

Key Terms for Regression Diagnostics

Standardized residuals
: Residuals divided by the standard error of the residuals.

Outliers
: Records (or outcome values) that are distant from the rest of the data (or the predicted outcome).

Influential value
: A value or record whose presence or absence makes a big difference in the regression equation.

Leverage
: The degree of influence that a single record has on a regression equation.

Synonyms
: hat-value

Non-normal residuals
: Non-normally distributed residuals can invalidate some technical requirements of regression, but are usually not a concern in data science.

Heteroskedasticity
: When some ranges of the outcome experience residuals with higher variance (may indicate a predictor missing from the equation).

Partial residual plots
: A diagnostic plot to illuminate the relationship between the outcome variable and a single predictor.

Synonyms
: added variables plot

Outliers

Generally speaking, an extreme value, also called an *outlier*, is one that is distant from most of the other observations. Just as outliers need to be handled for estimates of location and variability (see "Estimates of Location" on page 8 and "Estimates of Variability" on page 13), outliers can cause problems with regression models. In regression, an outlier is a record whose actual y value is distant from the predicted value. You can detect outliers by examining the *standardized residual*, which is the residual divided by the standard error of the residuals.

There is no statistical theory that separates outliers from nonoutliers. Rather, there are (arbitrary) rules of thumb for how distant from the bulk of the data an observa-

tion needs to be in order to be called an outlier. For example, with the boxplot, outliers are those data points that are too far above or below the box boundaries (see "Percentiles and Boxplots" on page 20), where "too far" = "more than 1.5 times the inter-quartile range." In regression, the standardized residual is the metric that is typically used to determine whether a record is classified as an outlier. Standardized residuals can be interpreted as "the number of standard errors away from the regression line."

Let's fit a regression to the King County house sales data for all sales in zip code 98105:

```
house_98105 <- house[house$ZipCode == 98105,]
lm_98105 <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
                Bedrooms + BldgGrade, data=house_98105)
```

We extract the standardized residuals using the `rstandard` function and obtain the index of the smallest residual using the `order` function:

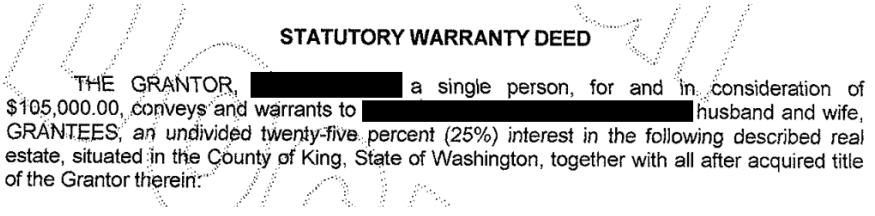
```
sresid <- rstandard(lm_98105)
idx <- order(sresid)
sresid[idx[1]]
    20431
-4.326732
```

The biggest overestimate from the model is more than four standard errors above the regression line, corresponding to an overestimate of $757,753. The original data record corresponding to this outlier is as follows:

```
house_98105[idx[1], c('AdjSalePrice', 'SqFtTotLiving', 'SqFtLot',
              'Bathrooms', 'Bedrooms', 'BldgGrade')]

AdjSalePrice SqFtTotLiving SqFtLot Bathrooms Bedrooms BldgGrade
         (dbl)         (int)   (int)     (dbl)    (int)     (int)
1       119748          2900    7276         3        6         7
```

In this case, it appears that there is something wrong with the record: a house of that size typically sells for much more than $119,748 in that zip code. Figure 4-4 shows an excerpt from the statuatory deed from this sale: it is clear that the sale involved only partial interest in the property. In this case, the outlier corresonds to a sale that is anomalous and should not be included in the regression. Outliers could also be the result of other problems, such as a "fat-finger" data entry or a mismatch of units (e.g., reporting a sale in thousands of dollars versus simply dollars).



STATUTORY WARRANTY DEED

THE GRANTOR, [redacted] a single person, for and in consideration of $105,000.00, conveys and warrants to [redacted] husband and wife, GRANTEES, an undivided twenty-five percent (25%) interest in the following described real estate, situated in the County of King, State of Washington, together with all after acquired title of the Grantor therein:

Figure 4-4. Statutory warrant of deed for the largest negative residual

For big data problems, outliers are generally not a problem in fitting the regression to be used in predicting new data. However, outliers are central to anomaly detection, where finding outliers is the whole point. The outlier could also correspond to a case of fraud or an accidental action. In any case, detecting outliers can be a critical business need.

Influential Values

A value whose absence would significantly change the regression equation is termed an *influential observation*. In regression, such a value need not be associated with a large residual. As an example, consider the regression lines in Figure 4-5. The solid line corresponds to the regression with all the data, while the dashed line corresonds to the regression with the point in the upper-right removed. Clearly, that data value has a huge influence on the regression even though it is not associated with a large outlier (from the full regression). This data value is considered to have high *leverage* on the regression.

In addition to standardized residuals (see "Outliers" on page 156), statisticians have developed several metrics to determine the influence of a single record on a regression. A common measure of leverage is the *hat-value*; values above $2(P + 1)/n$ indicate a high-leverage data value.[6]

6 The term *hat-value* comes from the notion of the hat matrix in regression. Multiple linear regression can be expressed by the formula $\hat{Y} = HY$ where H is the hat matrix. The hat-values correspond to the diagonal of H.

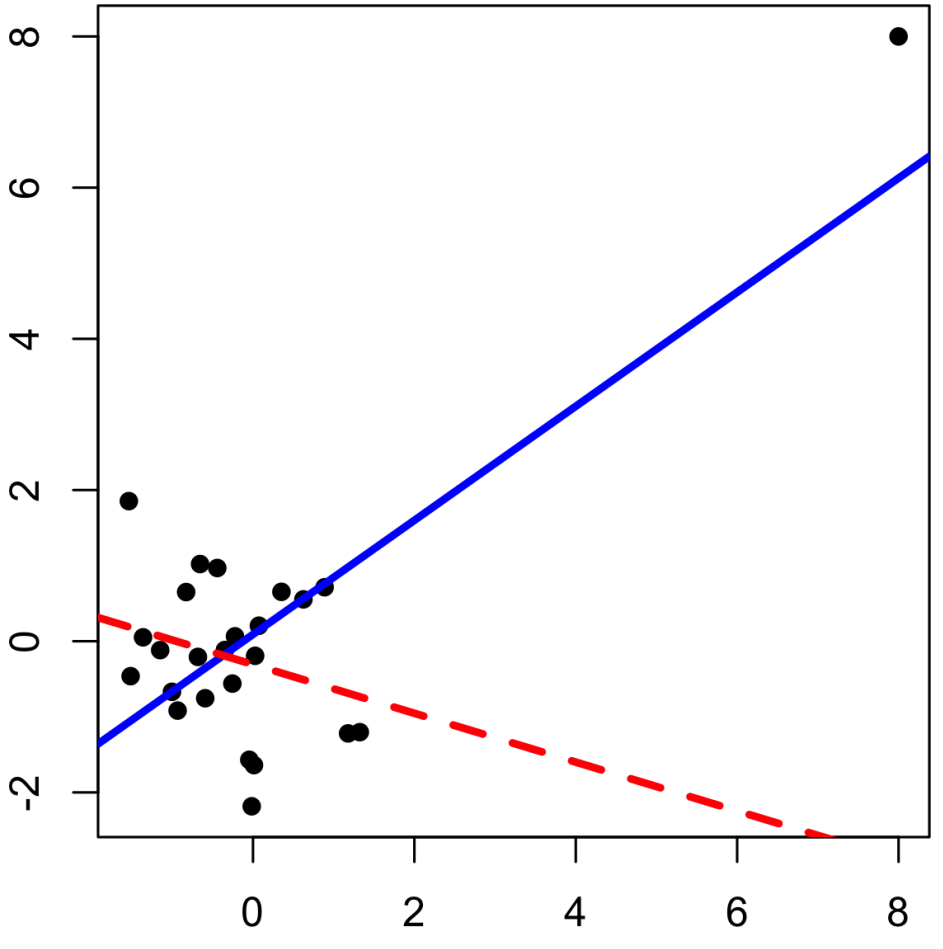


Figure 4-5. An example of an influential data point in regression

Another metric is *Cook's distance*, which defines influence as a combination of leverage and residual size. A rule of thumb is that an observation has high influence if Cook's distance exceeds $4/(n - P - 1)$.

An *influence plot* or *bubble plot* combines standardized residuals, the hat-value, and Cook's distance in a single plot. Figure 4-6 shows the influence plot for the King County house data, and can be created by the following R code.

```
std_resid <- rstandard(lm_98105)
cooks_D <- cooks.distance(lm_98105)
hat_values <- hatvalues(lm_98105)
plot(hat_values, std_resid, cex=10*sqrt(cooks_D))
abline(h=c(-2.5, 2.5), lty=2)
```

There are apparently several data points that exhibit large influence in the regression. Cook's distance can be computed using the function `cooks.distance`, and you can use `hatvalues` to compute the diagnostics. The hat values are plotted on the x-axis, the residuals are plotted on the y-axis, and the size of the points is related to the value of Cook's distance.

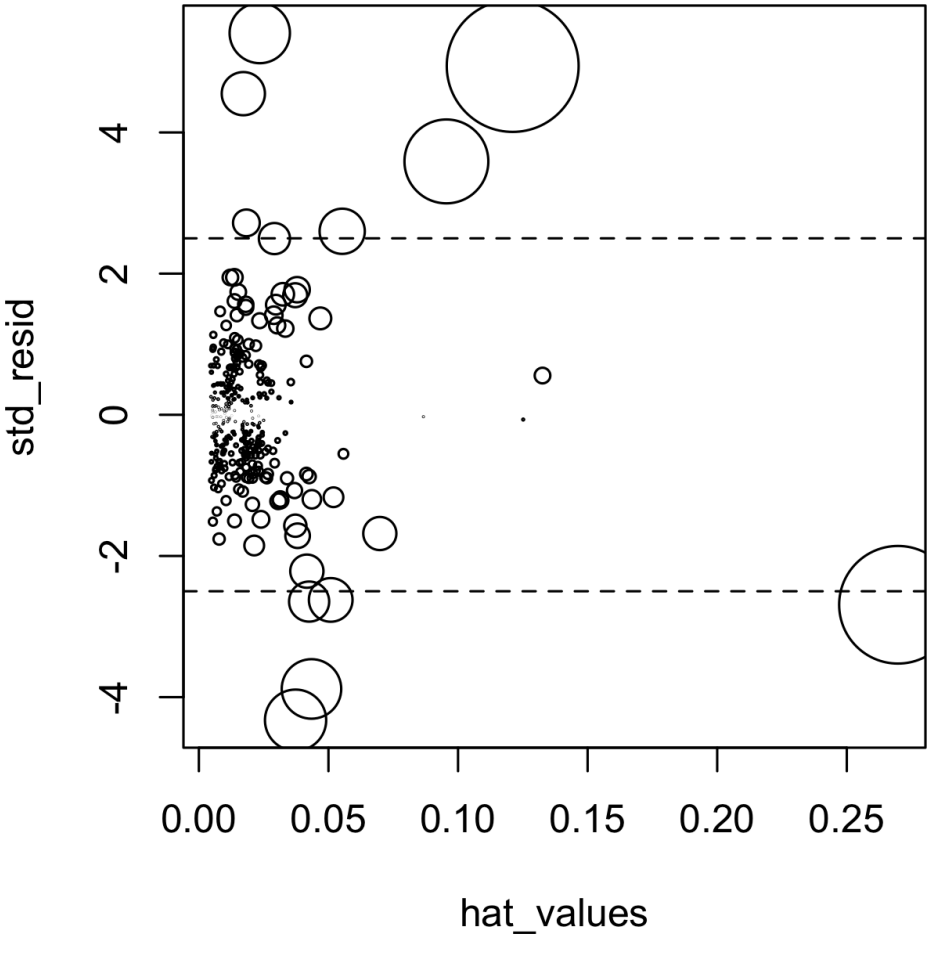


Figure 4-6. A plot to determine which observations have high influence

Table 4-2 compares the regression with the full data set and with highly influential data points removed. The regression coefficient for `Bathrooms` changes quite dramatically.[7]

Table 4-2. Comparison of regression coefficients with the full data and with influential data removed

	Original	Influential removed
(Intercept)	–772550	–647137
SqFtTotLiving	210	230
SqFtLot	39	33
Bathrooms	2282	–16132
Bedrooms	–26320	–22888
BldgGrade	130000	114871

For purposes of fitting a regression that reliably predicts future data, identifying influential observations is only useful in smaller data sets. For regressions involving many records, it is unlikely that any one observation will carry sufficient weight to cause extreme influence on the fitted equation (although the regression may still have big outliers). For purposes of anomaly detection, though, identifying influential observations can be very useful.

Heteroskedasticity, Non-Normality and Correlated Errors

Statisticians pay considerable attention to the distribution of the residuals. It turns out that ordinary least squares (see "Least Squares" on page 132) are unbiased, and in some cases the "optimal" estimator, under a wide range of distributional assumptions. This means that in most problems, data scientists do not need to be too concerned with the distribution of the residuals.

The distribution of the residuals is relevant mainly for the validity of formal statistical inference (hypothesis tests and p-values), which is of minimal importance to data scientists concerned mainly with predictive accuracy. For formal inference to be fully valid, the residuals are assumed to be normally distributed, have the same variance, and be independent. One area where this may be of concern to data scientists is the standard calculation of confidence intervals for predicted values, which are based upon the assumptions about the residuals (see "Confidence and Prediction Intervals" on page 143).

7 The coefficient for `Bathrooms` becomes negative, which is unintuitive. Location has not been taken into account and the zip code 98105 contains areas of disparate types of homes. See "Confounding Variables" on page 152 for a discussion of confounding variables.

Heteroskedasticity is the lack of constant residual variance across the range of the predicted values. In other words, errors are greater for some portions of the range than for others. The `ggplot2` package has some convenient tools to analyze residuals.

The following code plots the absolute residuals versus the predicted values for the `lm_98105` regression fit in "Outliers" on page 156.

```
df <- data.frame(
  resid = residuals(lm_98105),
  pred = predict(lm_98105))
ggplot(df, aes(pred, abs(resid))) +
  geom_point() +
  geom_smooth()
```

Figure 4-7 shows the resulting plot. Using `geom_smooth`, it is easy to superpose a smooth of the absolute residuals. The function calls the `loess` method to produce a visual smooth to estimate the relationship between the variables on the x-axis and y-axis in a scatterplot (see Scatterplot Smoothers on page 164).

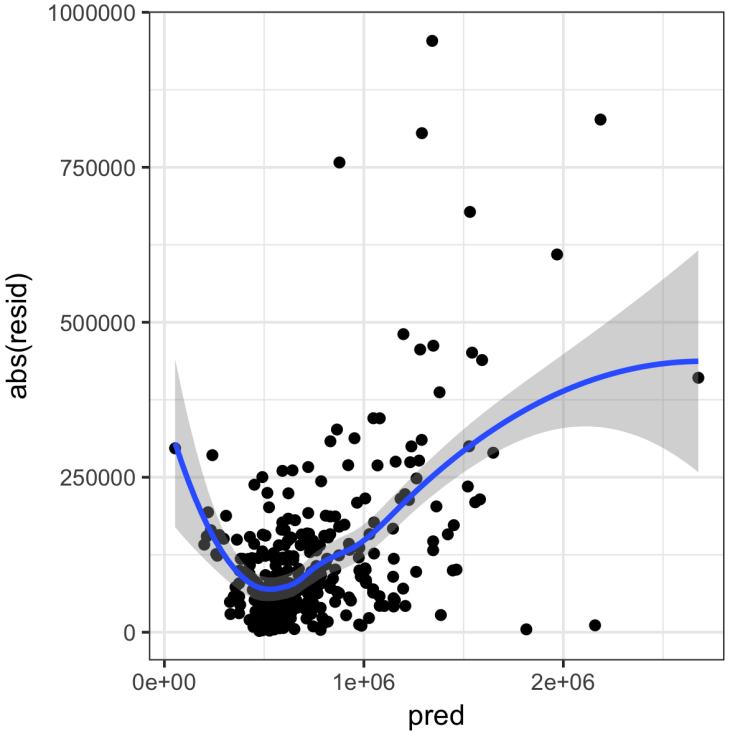


Figure 4-7. A plot of the absolute value of the residuals versus the predicted values

Evidently, the variance of the residuals tends to increase for higher-valued homes, but is also large for lower-valued homes. This plot indicates that `lm_98105` has *heteroskedastic* errors.

Why Would a Data Scientist Care about Heteroskedasticity?

Heteroskedasticity indicates that prediction errors differ for different ranges of the predicted value, and may suggest an incomplete model. For example, the heteroskedasticity in `lm_98105` may indicate that the regression has left something unaccounted for in high- and low-range homes.

Figure 4-8 is a histogram of the standarized residuals for the `lm_98105` regression. The distribution has decidely longer tails than the normal distribution, and exhibits mild skewness toward larger residuals.

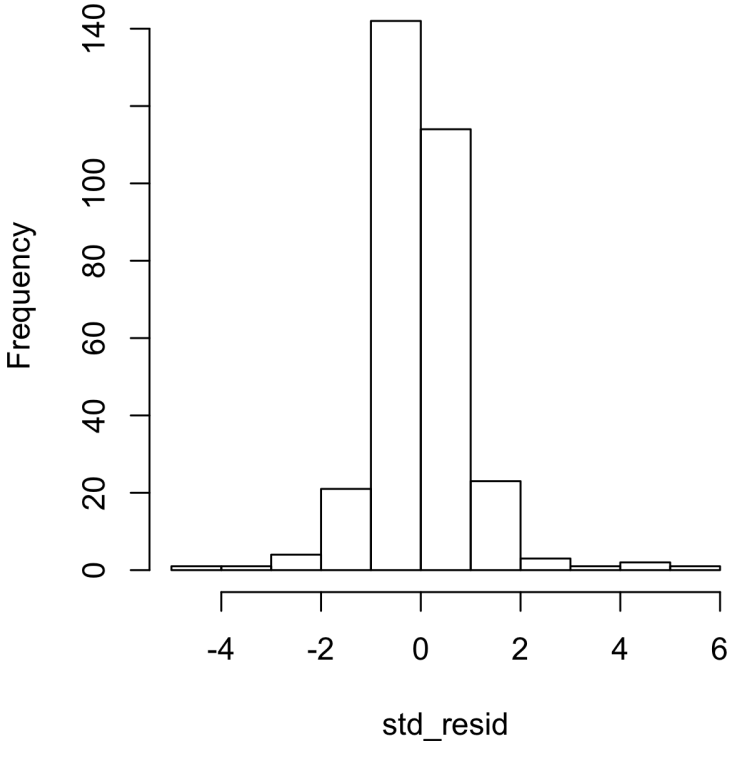


Figure 4-8. A histogram of the residuals from the regression of the housing data

Statisticians may also check the assumption that the errors are independent. This is particularly true for data that is collected over time. The *Durbin-Watson* statistic can

be used to detect if there is significant autocorrelation in a regression involving time series data.

Even though a regression may violate one of the distributional assumptions, should we care? Most often in data science, the interest is primarily in predictive accuracy, so some review of heteroskedasticity may be in order. You may discover that there is some signal in the data that your model has not captured. Satisfying distributional assumptions simply for the sake of validating formal statistical inference (p-values, F-statistics, etc.), however, is not that important for the data scientist.

Scatterplot Smoothers

Regression is about modeling the relationship between the response and predictor variables. In evaluating a regression model, it is useful to use a *scatterplot smoother* to visually highlight relationships between two variables.

For example, in Figure 4-7, a smooth of the relationship between the absolute residuals and the predicted value shows that the variance of the residuals depends on the value of the residual. In this case, the `loess` function was used; `loess` works by repeatedly fitting a series of local regressions to contiguous subsets to come up with a smooth. While `loess` is probably the most commonly used smoother, other scatterplot smoothers are available in R, such as super smooth (`supsmu`) and kernal smoothing (`ksmooth`). For the purposes of evaluating a regression model, there is typically no need to worry about the details of these scatterplot smooths.

Partial Residual Plots and Nonlinearity

Partial residual plots are a way to visualize how well the estimated fit explains the relationship between a predictor and the outcome. Along with detection of outliers, this is probably the most important diagnostic for data scientists. The basic idea of a partial residual plot is to isolate the relationship between a predictor variable and the response, *taking into account all of the other predictor variables*. A partial residual might be thought of as a "synthetic outcome" value, combining the prediction based on a single predictor with the actual residual from the full regression equation. A partial residual for predictor X_i is the ordinary residual plus the regression term associated with X_i:

$$\text{Partial residual} = \text{Residual} + \hat{b}_i X_i$$

where $\hat{b}_i$ is the estimated regression coefficient. The `predict` function in R has an option to return the individual regression terms $\hat{b}_i X_i$:

```
terms <- predict(lm_98105, type='terms')
partial_resid <- resid(lm_98105) + terms
```

The partial residual plot displays the X_i on the x-axis and the partial residuals on the y-axis. Using `ggplot2` makes it easy to superpose a smooth of the partial residuals.

```
df <- data.frame(SqFtTotLiving = house_98105[, 'SqFtTotLiving'],
                 Terms = terms[, 'SqFtTotLiving'],
                 PartialResid = partial_resid[, 'SqFtTotLiving'])
ggplot(df, aes(SqFtTotLiving, PartialResid)) +
  geom_point(shape=1) + scale_shape(solid = FALSE) +
  geom_smooth(linetype=2) +
  geom_line(aes(SqFtTotLiving, Terms))
```

The resulting plot is shown in Figure 4-9. The partial residual is an estimate of the contribution that `SqFtTotLiving` adds to the sales price. The relationship between `SqFtTotLiving` and the sales price is evidently nonlinear. The regression line underestimates the sales price for homes less than 1,000 square feet and overestimates the price for homes between 2,000 and 3,000 square feet. There are too few data points above 4,000 square feet to draw conclusions for those homes.

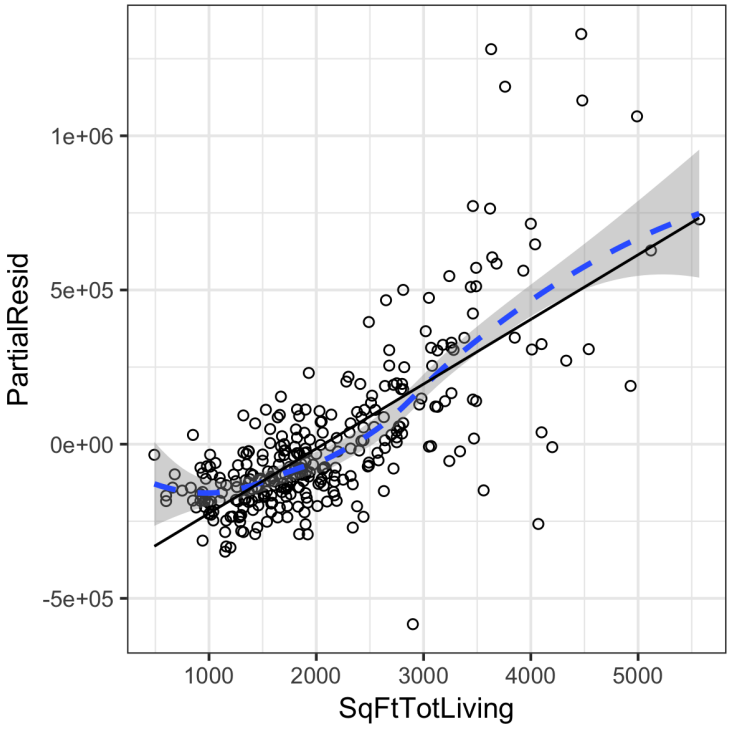


Figure 4-9. A partial residual plot for the variable SqFtTotLiving

This nonlinearity makes sense in this case: adding 500 feet in a small home makes a much bigger difference than adding 500 feet in a large home. This suggests that, instead of a simple linear term for `SqFtTotLiving`, a nonlinear term should be considered (see "Polynomial and Spline Regression" on page 166).

Key Ideas

- While outliers can cause problems for small data sets, the primary interest with outliers is to identify problems with the data, or locate anomalies.
- Single records (including regression outliers) can have a big influence on a regression equation with small data, but this effect washes out in big data.
- If the regression model is used for formal inference (p-values and the like), then certain assumptions about the distribution of the residuals should be checked. In general, however, the distribution of residuals is not critical in data science.
- The partial residuals plot can be used to qualitatively assess the fit for each regression term, possibly leading to alternative model specification.

Polynomial and Spline Regression

The relationship between the response and a predictor variable is not necessarily linear. The response to the dose of a drug is often nonlinear: doubling the dosage generally doesn't lead to a doubled response. The demand for a product is not a linear function of marketing dollars spent since, at some point, demand is likely to be saturated. There are several ways that regression can be extended to capture these nonlinear effects.

Key Terms for Nonlinear Regression

Polynomial regression
: Adds polynomial terms (squares, cubes, etc.) to a regression.

Spline regression
: Fitting a smooth curve with a series of polynomial segments.

Knots
: Values that separate spline segments.

Generalized additive models
: Spline models with automated selection of knots.

Synonyms
GAM

Nonlinear Regression

When statisticians talk about *nonlinear regression*, they are referring to models that can't be fit using least squares. What kind of models are nonlinear? Essentially all models where the response cannot be expressed as a linear combination of the predictors or some transform of the predictors. Nonlinear regression models are harder and computationally more intensive to fit, since they require numerical optimization. For this reason, it is generally preferred to use a linear model if possible.

Polynomial

Polynomial regression involves including polynomial terms to a regression equation. The use of polynomial regression dates back almost to the development of regression itself with a paper by Gergonne in 1815. For example, a quadratic regression between the response Y and the predictor X would take the form:

$$Y = b_0 + b_1 X + b_2 X^2 + e$$

Polynomial regression can be fit in R through the `poly` function. For example, the following fits a quadratic polynomial for `SqFtTotLiving` with the King County housing data:

```
lm(AdjSalePrice ~  poly(SqFtTotLiving, 2) + SqFtLot +
                BldgGrade +  Bathrooms +  Bedrooms,
                    data=house_98105)

Call:
lm(formula = AdjSalePrice ~ poly(SqFtTotLiving, 2) + SqFtLot +
    BldgGrade + Bathrooms + Bedrooms, data = house_98105)

Coefficients:
            (Intercept)  poly(SqFtTotLiving, 2)1
              -402530.47               3271519.49
poly(SqFtTotLiving, 2)2                   SqFtLot
               776934.02                    32.56
              BldgGrade                Bathrooms
               135717.06                 -1435.12
               Bedrooms
                -9191.94
```

There are now two coefficients associated with `SqFtTotLiving`: one for the linear term and one for the quadratic term.

The partial residual plot (see "Partial Residual Plots and Nonlinearity" on page 164) indicates some curvature in the regression equation associated with `SqFtTotLiving`. The fitted line more closely matches the smooth (see "Splines" on page 168) of the partial residuals as compared to a linear fit (see Figure 4-10).

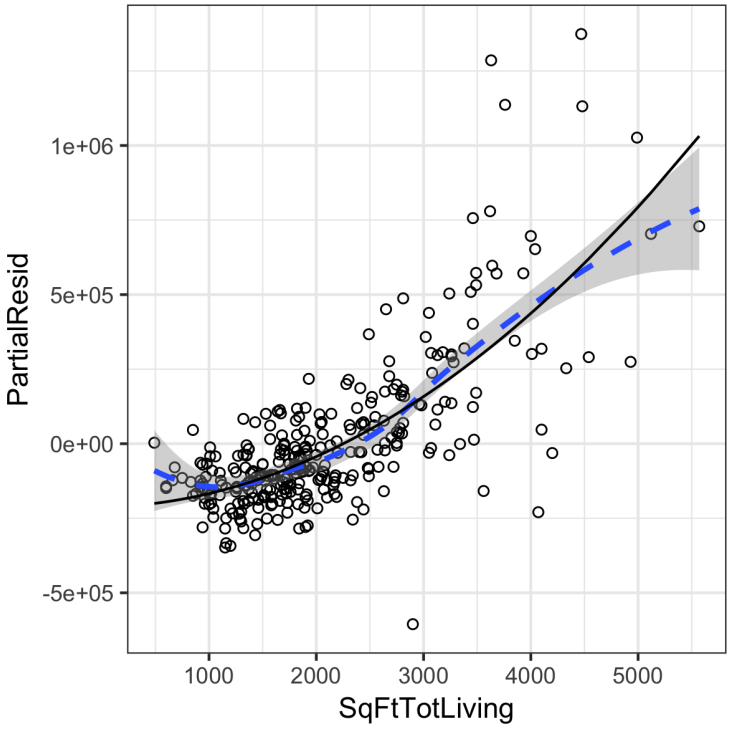


Figure 4-10. A polynomial regression fit for the variable SqFtTotLiving (solid line) versus a smooth (dashed line; see the following section about splines)

Splines

Polynomial regression only captures a certain amount of curvature in a nonlinear relationship. Adding in higher-order terms, such as a cubic quartic polynomial, often leads to undesirable "wiggliness" in the regression equation. An alternative, and often superior, approach to modeling nonlinear relationships is to use *splines*. *Splines* provide a way to smoothly interpolate between fixed points. Splines were originally used by draftsmen to draw a smooth curve, particularly in ship and aircraft building.

The splines were created by bending a thin piece of wood using weights, referred to as "ducks"; see Figure 4-11.

Figure 4-11. Splines were originally created using bendable wood and "ducks," and were used as a draftsman tool to fit curves. Photo courtesy Bob Perry.

The technical definition of a spline is a series of piecewise continuous polynomials. They were first developed during World War II at the US Aberdeen Proving Grounds by I. J. Schoenberg, a Romanian mathematician. The polynomial pieces are smoothly connected at a series of fixed points in a predictor variable, referred to as *knots*. Formulation of splines is much more complicated than polynomial regression; statistical software usually handles the details of fitting a spline. The R package `splines` includes the function `bs` to create a *b-spline* term in a regression model. For example, the following adds a b-spline term to the house regression model:

```
library(splines)
knots <- quantile(house_98105$SqFtTotLiving, p=c(.25, .5, .75))
lm_spline <- lm(AdjSalePrice ~ bs(SqFtTotLiving, knots=knots, degree=3) +
  SqFtLot + Bathrooms + Bedrooms + BldgGrade,  data=house_98105)
```

Two parameters need to be specified: the degree of the polynomial and the location of the knots. In this case, the predictor `SqFtTotLiving` is included in the model using a cubic spline (`degree=3`). By default, `bs` places knots at the boundaries; in addition, knots were also placed at the lower quartile, the median quartile, and the upper quartile.

In contrast to a linear term, for which the coefficient has a direct meaning, the coefficients for a spline term are not interpretable. Instead, it is more useful to use the visual display to reveal the nature of the spline fit. Figure 4-12 displays the partial residual plot from the regression. In contrast to the polynomial model, the spline model more closely matches the smooth, demonstrating the greater flexibility of splines. In this case, the line more closely fits the data. Does this mean the spline regression is a better model? Not necessarily: it doesn't make economic sense that very small homes (less than 1,000 square feet) would have higher value than slightly larger homes. This is possibly an artifact of a confounding variable; see "Confounding Variables" on page 152.

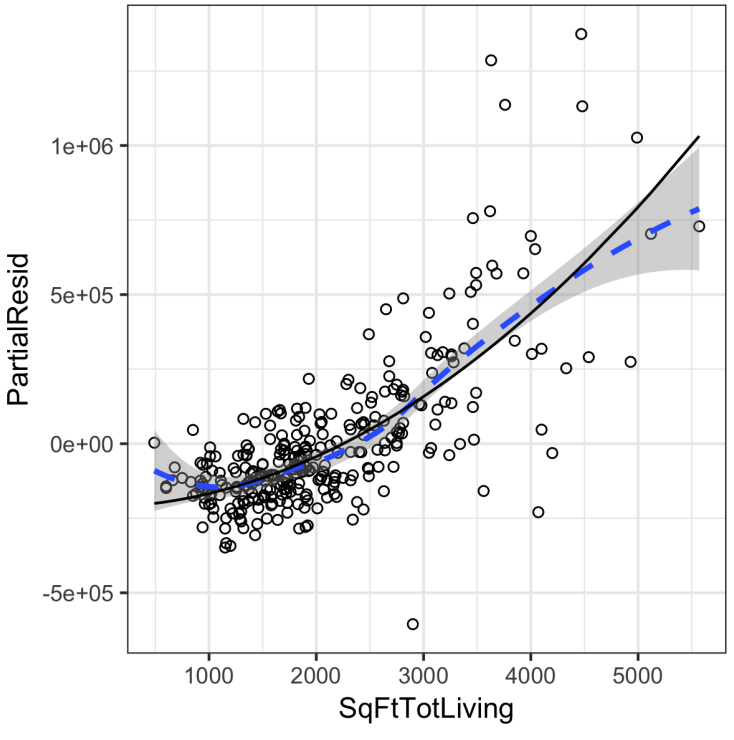


Figure 4-12. A spline regression fit for the variable SqFtTotLiving (solid line) compared to a smooth (dashed line)

Generalized Additive Models

Suppose you suspect a nonlinear relationship between the response and a predictor variable, either by a priori knowledge or by examining the regression diagnostics. Polynomial terms may not flexible enough to capture the relationship, and spline terms require specifying the knots. *Generalized additive models*, or *GAM*, are a technique to automatically fit a spline regression. The `gam` package in R can be used to fit a GAM model to the housing data:

```
library(mgcv)
lm_gam <- gam(AdjSalePrice ~ s(SqFtTotLiving) + SqFtLot +
                    Bathrooms +  Bedrooms + BldgGrade,
                data=house_98105)
```

The term `s(SqFtTotLiving)` tells the `gam` function to find the "best" knots for a spline term (see Figure 4-13).

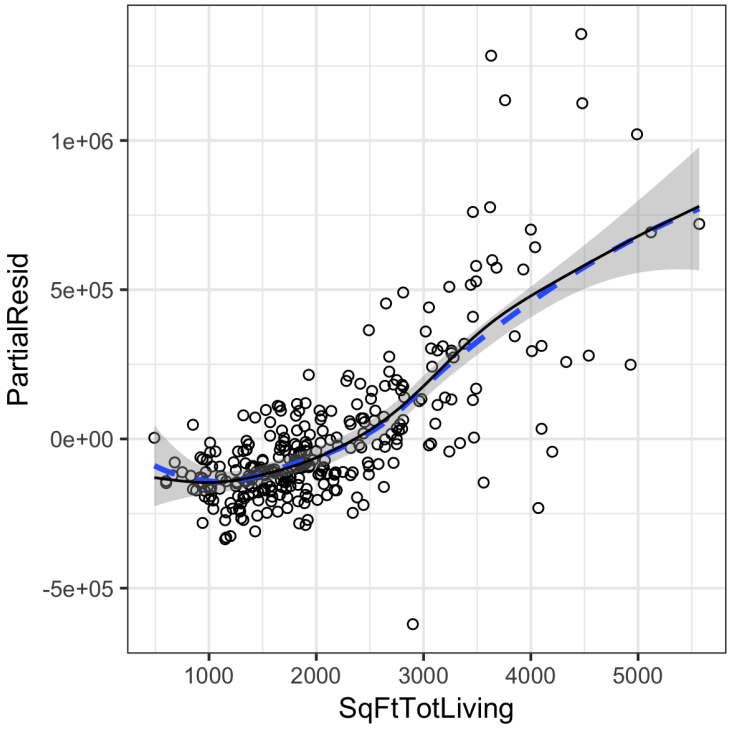


Figure 4-13. A GAM regression fit for the variable SqFtTotLiving (solid line) compared to a smooth (dashed line)

Key Ideas

- Outliers in a regression are records with a large residual.
- Multicollinearity can cause numerical instability in fitting the regression equation.
- A confounding variable is an important predictor that is omitted from a model and can lead to a regression equation with spurious relationships.
- An interaction term between two variables is needed if the effect of one variable depends on the *level* of the other.
- Polynomial regression can fit nonlinear relationships between predictors and the outcome variable.
- Splines are series of polynomial segments strung together, joining at knots.
- Generalized additive models (GAM) automate the process of specifying the knots in splines.

Further Reading

For more on spline models and GAMS, see *The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman, and its shorter cousin based on R, *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani; both are Springer books.

Summary

Perhaps no other statistical method has seen greater use over the years than regression—the process of establishing a relationship between multiple predictor variables and an outcome variable. The fundamental form is linear: each predictor variable has a coefficient that describes a linear relationship between the predictor and the outcome. More advanced forms of regression, such as polynomial and spline regression, permit the relationship to be nonlinear. In classical statistics, the emphasis is on finding a good fit to the observed data to explain or describe some phenomenon, and the strength of this fit is how traditional ("in-sample") metrics are used to assess the model. In data science, by contrast, the goal is typically to predict values for new data, so metrics based on predictive accuracy for out-of-sample data are used. Variable selection methods are used to reduce dimensionality and create more compact models.

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]

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

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 $P(X_i | Y_i)$.

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.

Predictor variables must be categorical (factor) variables in the standard naive Bayes algorithm. See "Numeric Predictor Variables" on page 178 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.

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:

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

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

$P\left(X_1, X_2, \cdots X_p \middle| Y = i\right)$, 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 $P\left(X_j \middle| Y = i\right)$. 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" on page 64)—to estimate the conditional probability $P(X_j \mid Y = i)$.

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" on page 250). In addition, discriminant analysis can provide a measure of predictor importance, and it is used as a computationally efficient method of feature selection.

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.

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.

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" on page 9). 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?" on page 16).

As with the correlation coefficient (see "Correlation" on page 29), 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.

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

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 on page 214) 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.

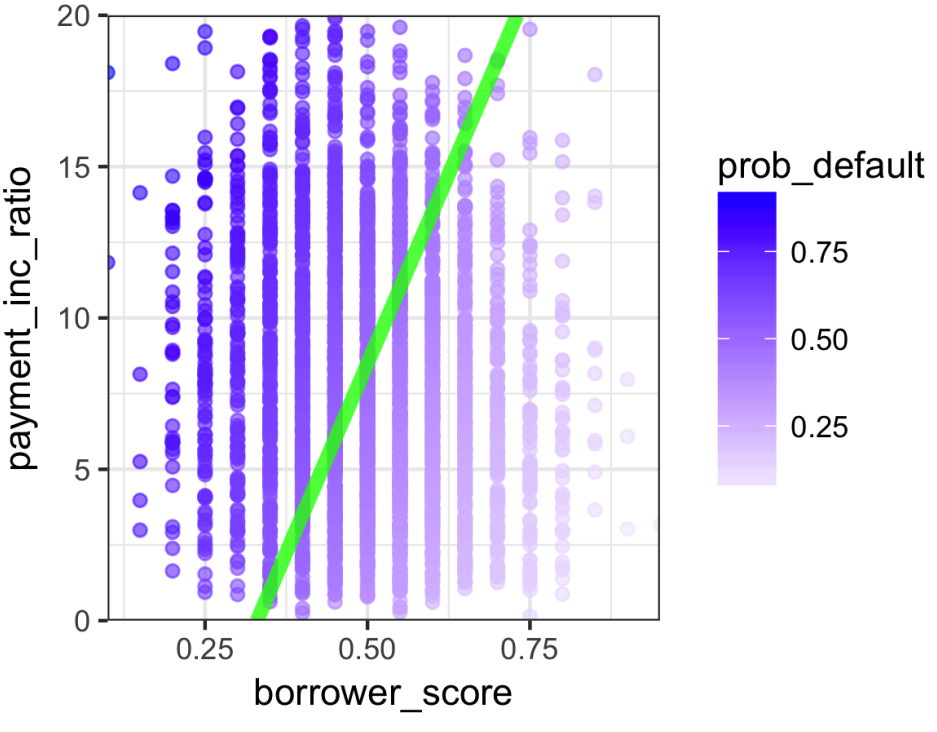


Figure 5-1. LDA prediction of loan default using two variables: a score of the borrower's creditworthiness and the payment to income ratio.

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:

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

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.

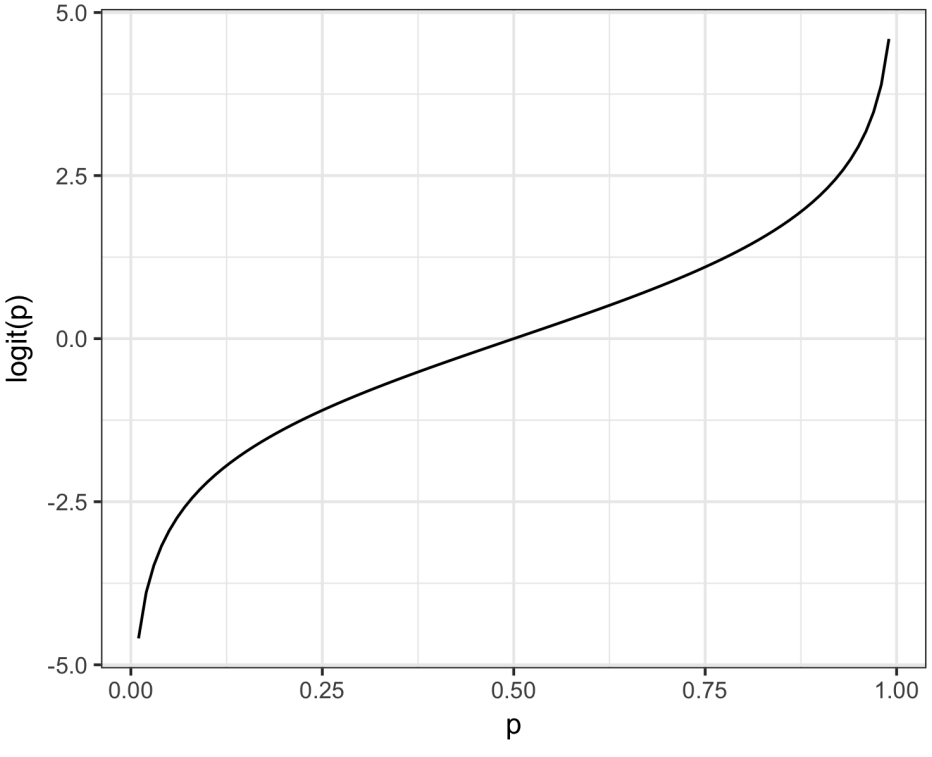


Figure 5-2. The function that maps a probability to a scale suitable for a linear model (logit)

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" on page 210.

```
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" on page 145). 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" on page 218.

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" on page 196).

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" on page 186, 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 $exp(1.21226) \approx 3.4$. 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 $exp(1) \approx 2.72$ in the odds ratio.

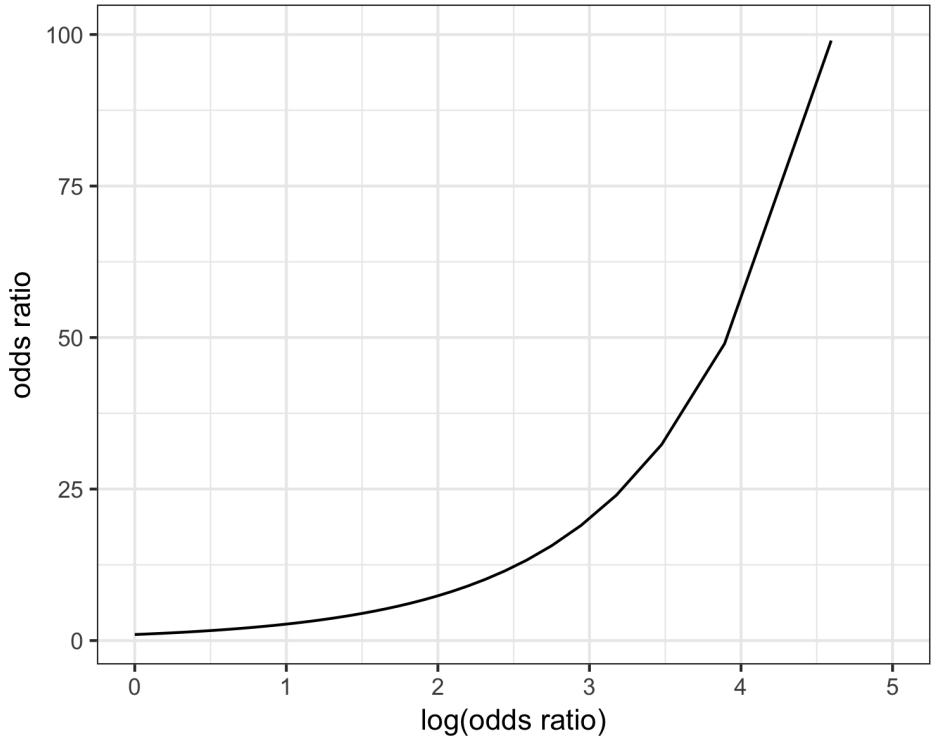


Figure 5-3. The relationship between the odds ratio and the log-odds ratio

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 $\mathscr{P}_\theta(X_1, X_2, \cdots, X_n)$ 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 $\mathscr{P}_\theta(X_1, X_2, \cdots, X_n)$; that is, it maximizes the probability of observing $(X_1, X_2, \cdots, X_n)$ given the model $\mathscr{P}$..] In the fitting process, the model is evaluated using a metric called *deviance*:

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

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" on page 145. 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" on page 214).

Assessing the Model

Like other classification methods, logistic regression is assessed by how accurately the model classifies new data (see "Evaluating Classification Models" on page 194). 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" on page 136) 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" on page 194.

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" on page 150). You can fit generalized additive models (see "Generalized Additive Models" on page 170) 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.

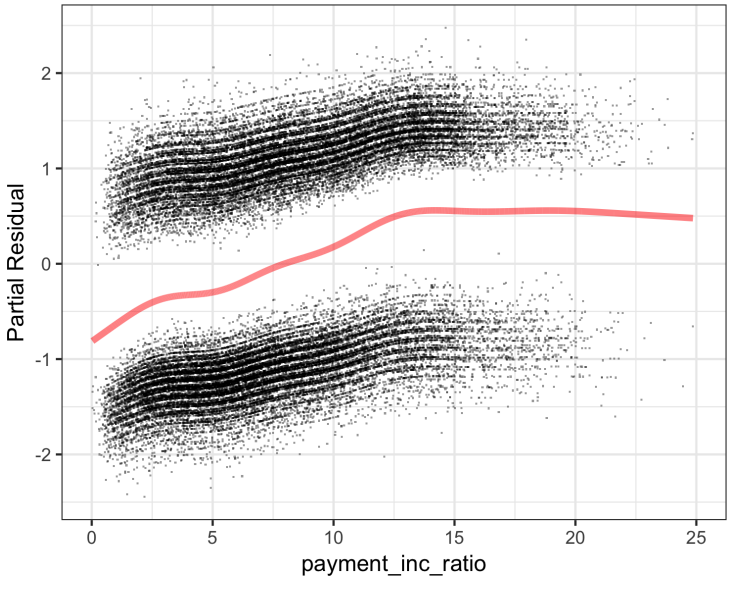


Figure 5-4. Partial residuals from logistic regression

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" on page 190.

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)
```

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.

```
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" on page 197 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.

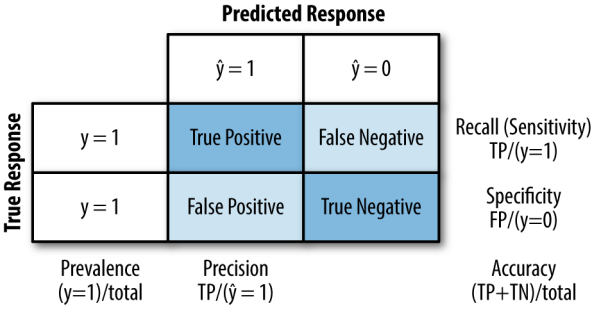


Figure 5-5. Confusion matrix for a binary response and various metrics

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)) +
```

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.

```
    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%.

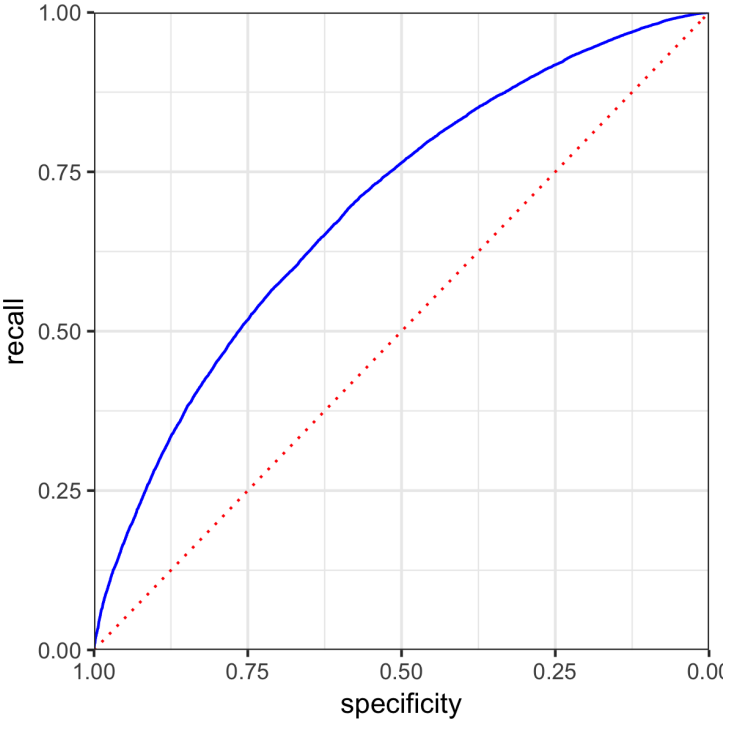


Figure 5-6. ROC curve for the loan data

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.

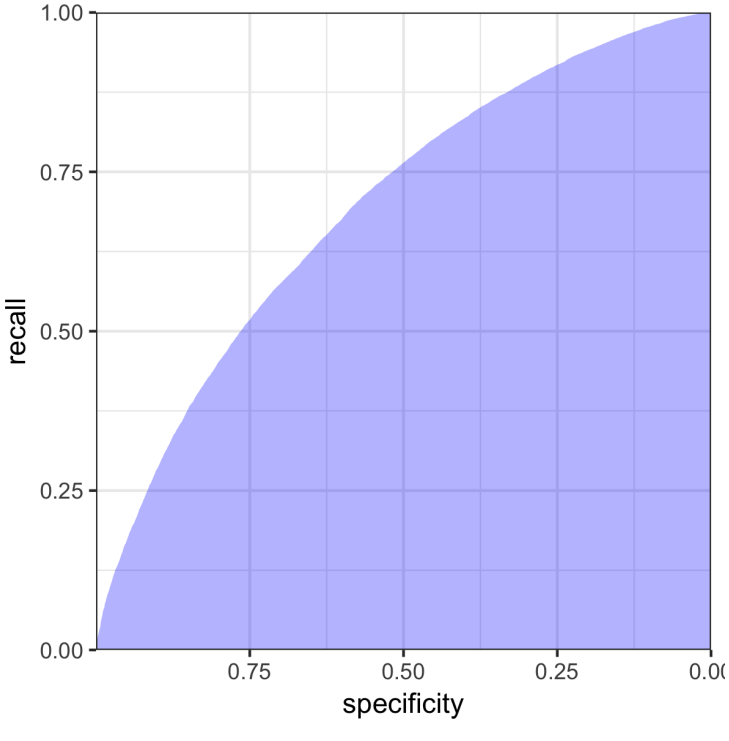


Figure 5-7. Area under the ROC curve for the loan data

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" on page 184 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" on page 204) 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" on page 210) 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" on page 219). 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.

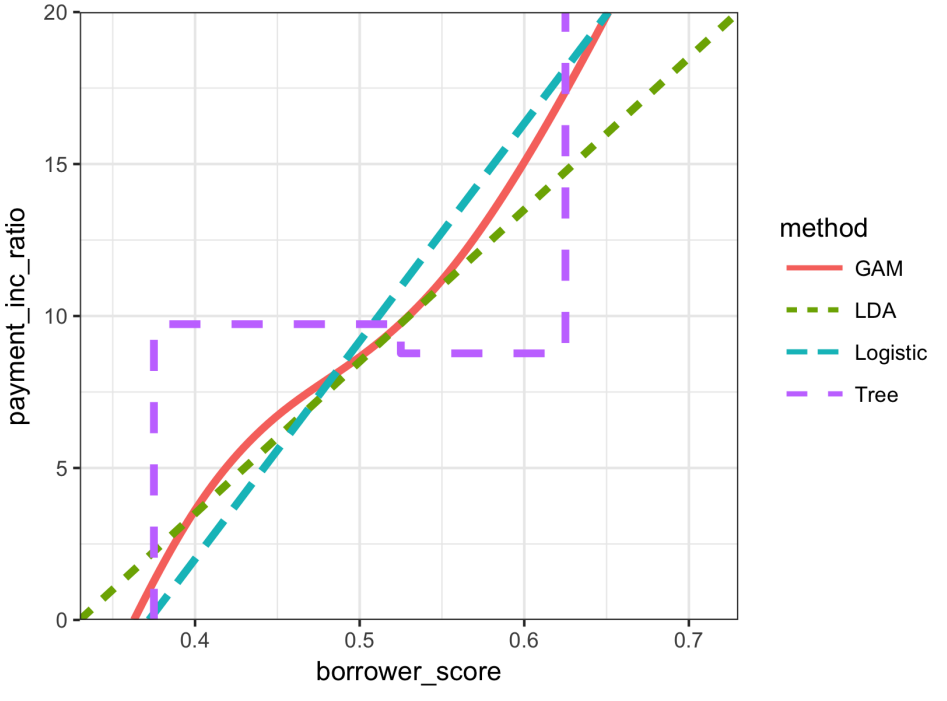


Figure 5-8. Comparison of the classification rules for four different methods

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.

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" on page 228), grew up firmly in the statistics camp. Boosting (see "Boosting" on page 237), 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.

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

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.

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" on page 203. 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.

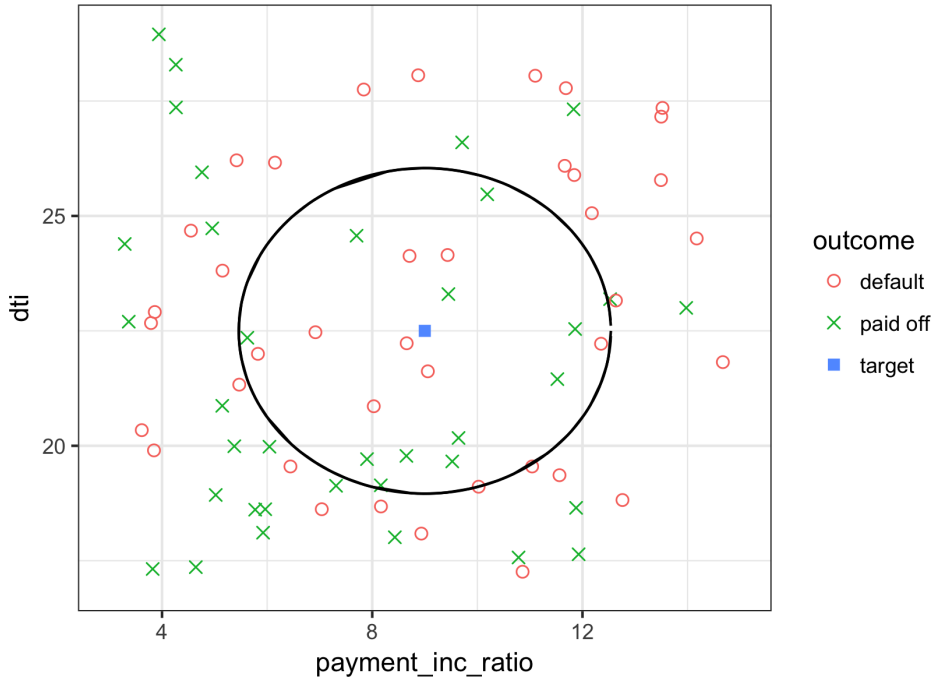


Figure 6-2. KNN prediction of loan default using two variables: debt-to-income ratio and loan payment-to-income ratio

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)" on page 215.

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" on page 180.

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

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

In linear and logistic regression, one hot encoding causes problems with multicollinearity; see "Multicollinearity" on page 151. 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.

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" on page 211) 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.

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.

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" on page 64).

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" on page 138 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" on page 151). 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.

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" on page 228 and "Boosting" on page 237), form the basis for the most widely used and powerful predictive modeling tools in data science for both regression and classification.

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

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.

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" on page 210 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.

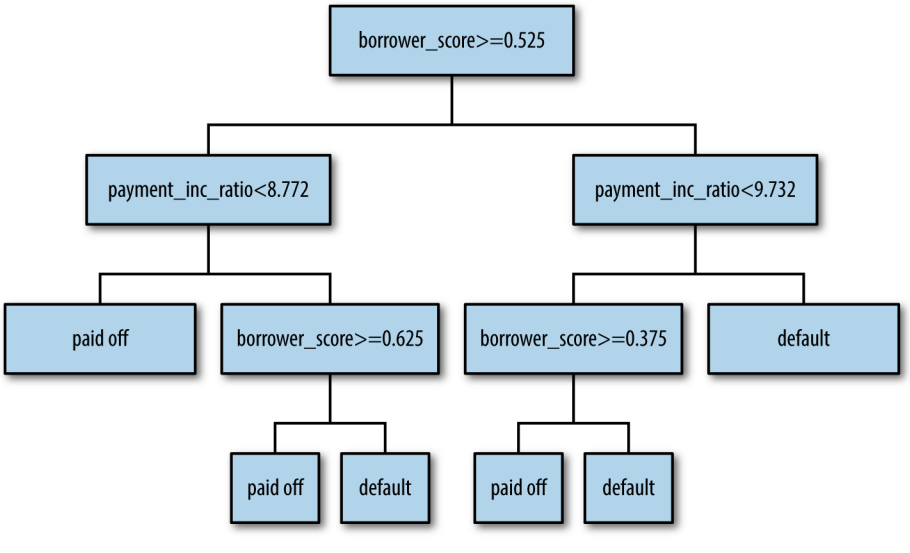


Figure 6-3. The rules for a simple tree model fit to the loan data

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.

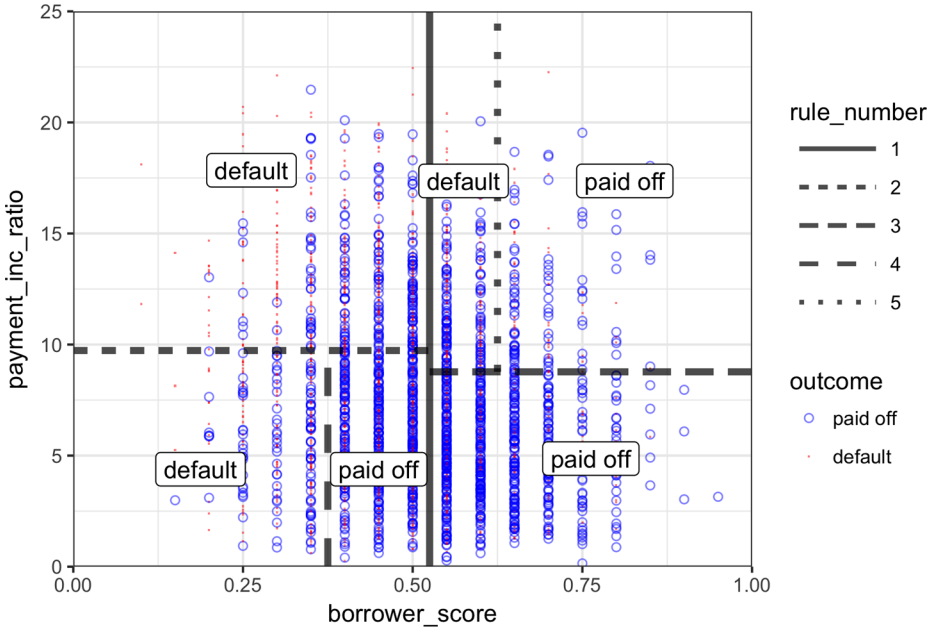


Figure 6-4. The rules for a simple tree model fit to the loan data

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.

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 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.

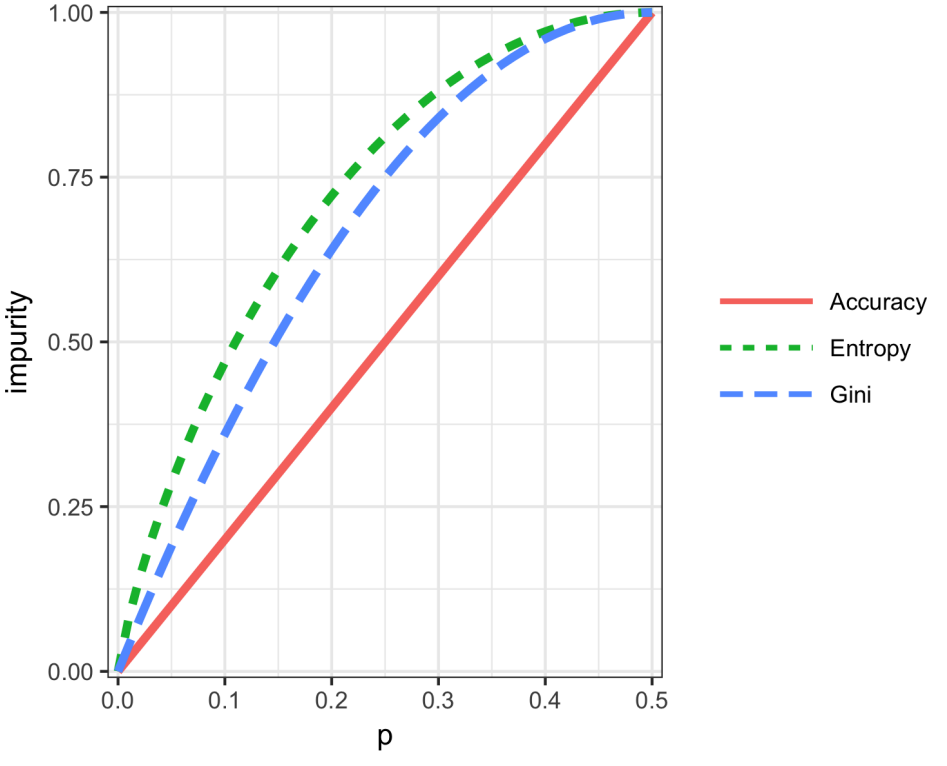


Figure 6-5. Gini impurity and entropy measures

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" on page 200).

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 on page 217) 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" on page 228), 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 on page 217). The most common way to estimate a good value of `cp` is via cross-validation (see "Cross-Validation" on page 138):

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 `cp`s 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" on page 136) 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" on page 228 and "Boosting" on page 237), 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 Surowiecki 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" on page 237.

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" on page 224). 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" on page 222), the random forest algorithm adds two more steps: the bagging discussed earlier (see "Bagging and the Random Forest" on page 228), and the bootstrap sampling of variables at each split:

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.

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 $X_{j(1)}, X_{j(2)}, ..., X_{j(p)}$, 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" on page 210 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:

```
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))
```

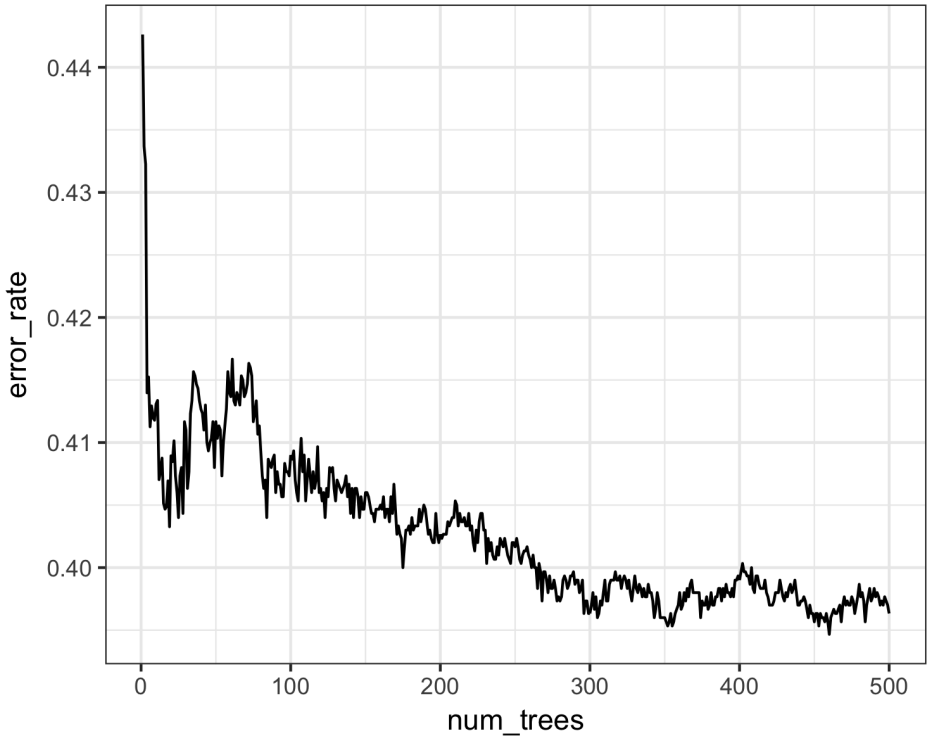


Figure 6-6. The improvement in accuracy of the random forest with the addition of more trees

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

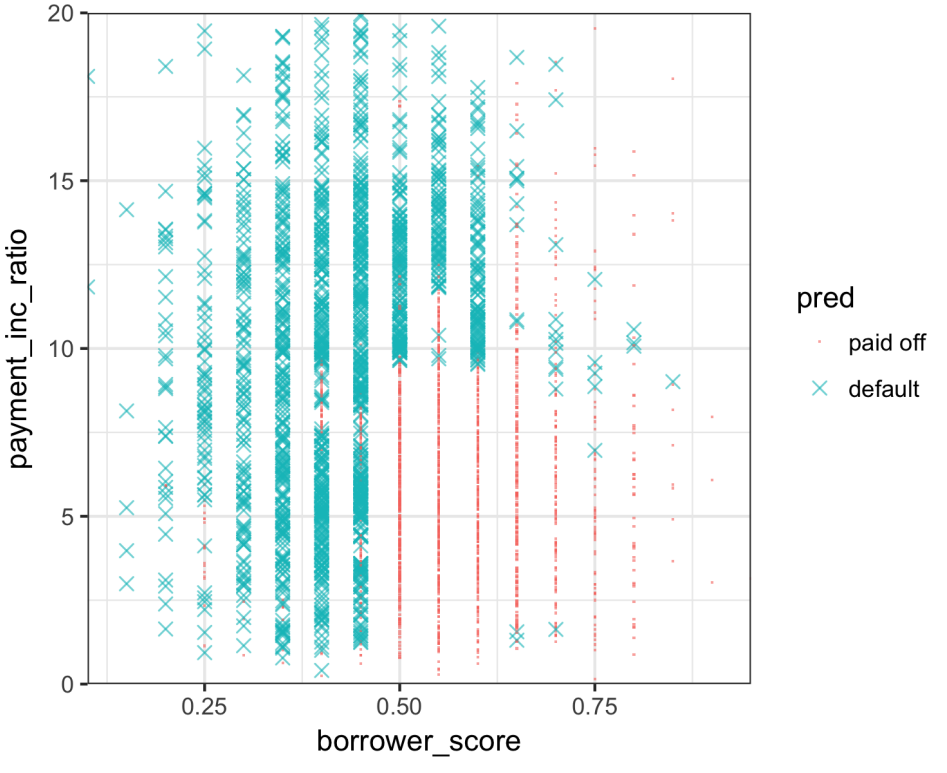


Figure 6-7. The predicted outcomes from the random forest applied to the loan default data

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 on page 217).

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" on page 153). 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.

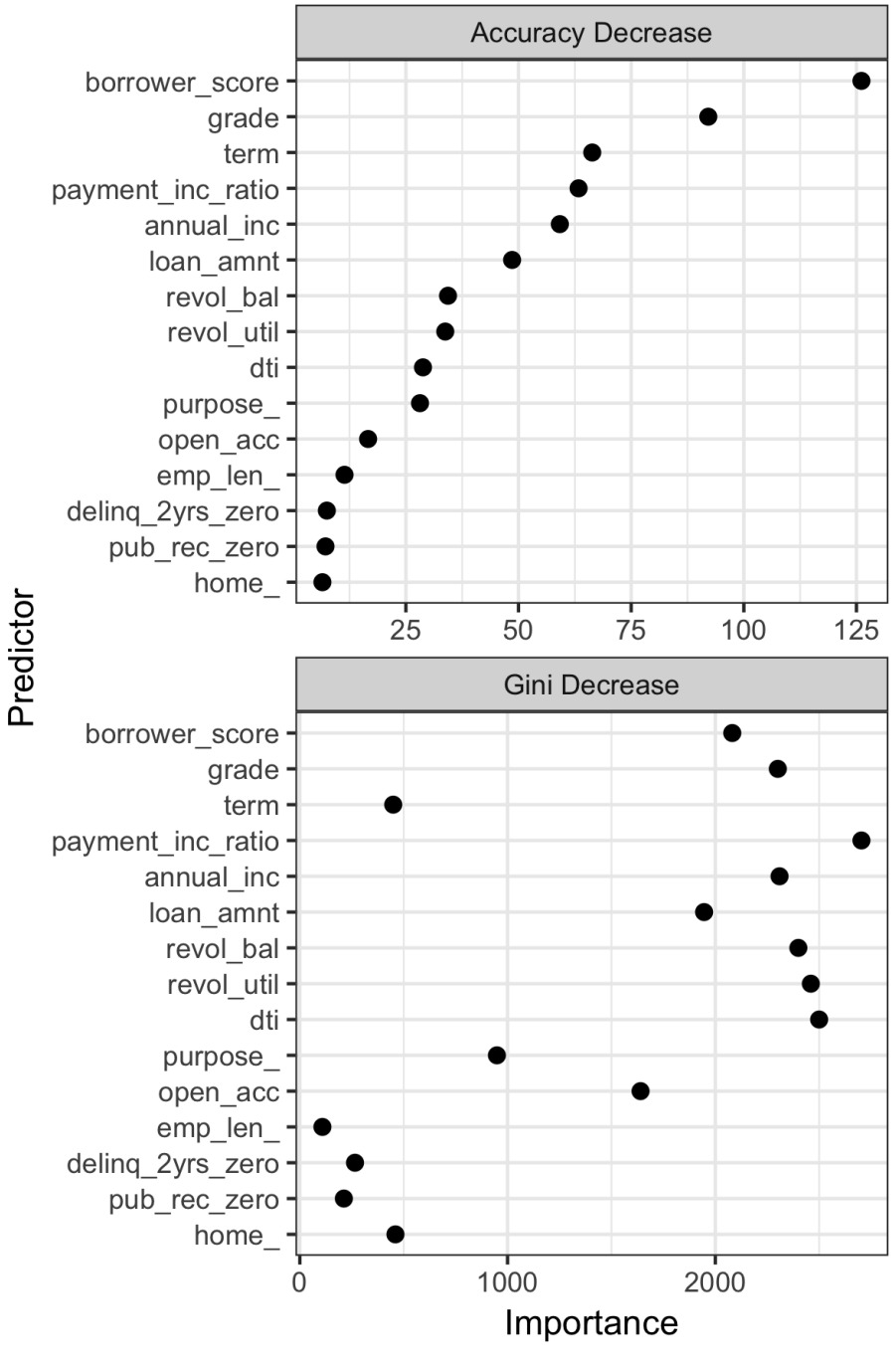


Figure 6-8. The importance of variables for the full model fit to the loan data

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" on page 224) 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" on page 138) 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.

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" on page 228). 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" on page 164). 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 $i = 1, 2, ..., N$. Initialize the ensemble model $\hat{F}_0 = 0$.

2. Train a model using $\hat{f}_m$ using the observation weights $w_1, w_2, ..., w_N$ 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 $w_1, w_2, ..., w_N$ 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" on page 245). 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" on page 238). 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.

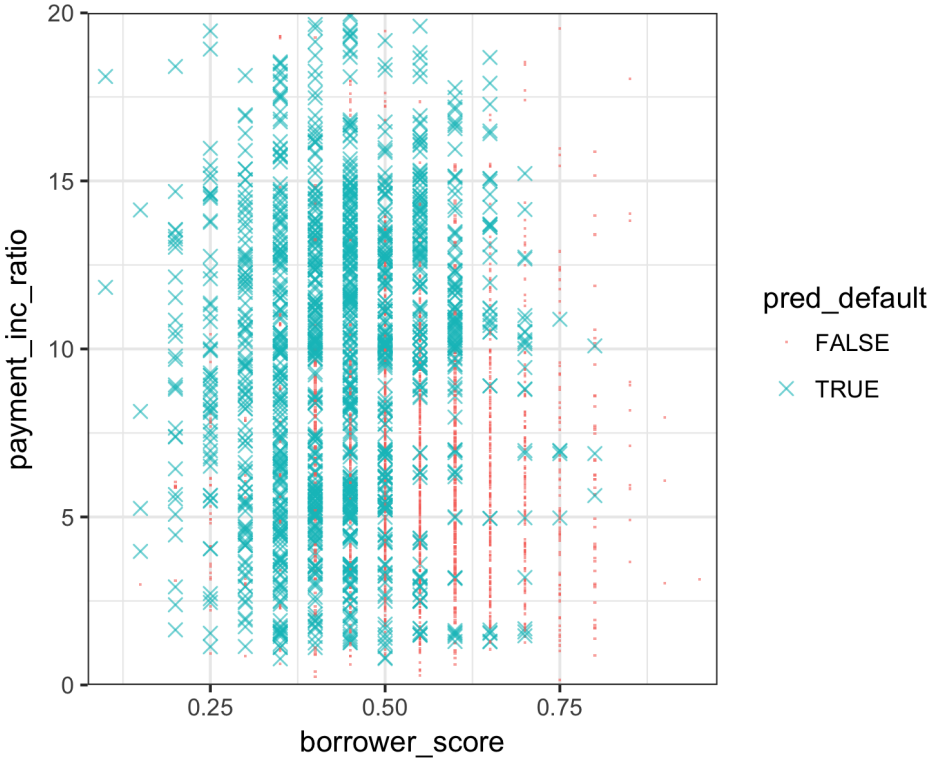


Figure 6-9. The predicted outcomes from XGBoost applied to the loan default data

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" on page 239). 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" on page 224). 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" on page 213). 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.

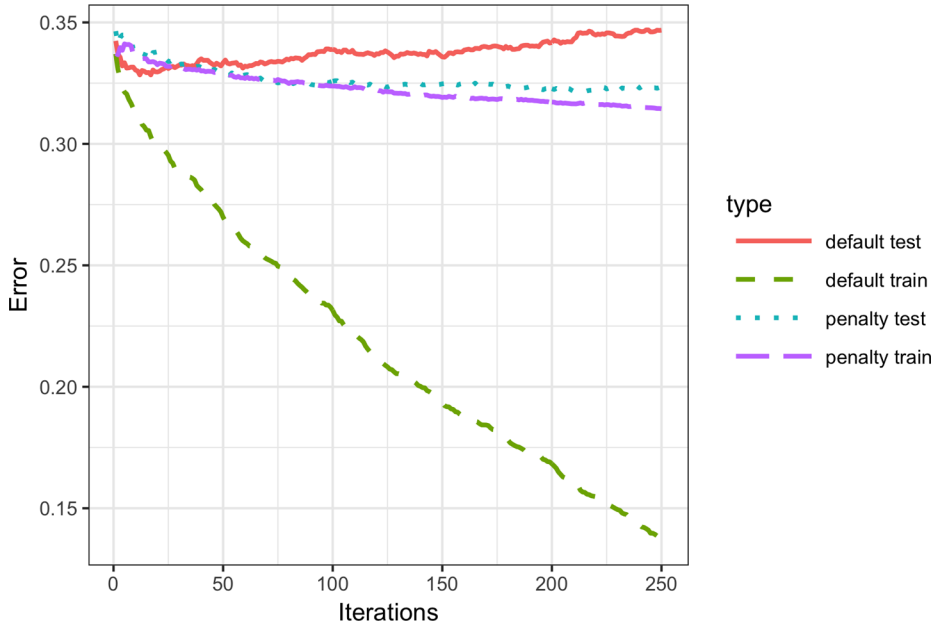


Figure 6-10. The error rate of the default XGBoost versus a penalized version of XGBoost

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" on page 132. *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:

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

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" on page 246 for a discussion. As seen in "Regularization: Avoiding Overfitting" on page 241, 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" on page 138. 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" on page 239) 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" on page 241).

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.

CHAPTER 7
Unsupervised Learning

The term *unsupervised learning* refers to statistical methods that extract meaning from data without training a model on labeled data (data where an outcome of interest is known). In Chapters 4 and 5, the goal is to build a model (set of rules) to predict a response from a set of predictor variables. Unsupervised learning also constructs a model of the data, but does not distinguish between a response variable and predictor variables.

Unsupervised learning can have different possible goals. In some cases, it can be used to create a predictive rule in the absence of a labeled response. *Clustering* methods can be used to identify meaningful groups of data. For example, using the web clicks and demographic data of a user on a website, we may be able to group together different types of users. The website could then be personalized to these different types.

In other cases, the goal may be to *reduce the dimension* of the data to a more manageable set of variables. This reduced set could then be used as input into a predictive model, such as regression or classification. For example, we may have thousands of sensors to monitor an industrial process. By reducing the data to a smaller set of features, we may be able to build a more powerful and interpretable model to predict process failure than by including data streams from thousands of sensors.

Finally, unsupervised learning can be viewed as an extension of the exploratory data analysis (see Chapter 1) to situations where you are confronted with a large number of variables and records. The aim is to gain insight into a set of data and how the different variables relate to each other. Unsupervised techniques give ways to sift through and analyze these variables and discover relationships.

Unsupervised Learning and Prediction

Unsupervised learning can play an important role for prediction, both for regression and classification problems. In some cases, we want to predict a category in the absence of any labeled data. For example, we might want to predict the type of vegetation in an area from a set of satellite sensory data. Since we don't have a response variable to train a model, clustering gives us a way to identify common patterns and categorize the regions.

Clustering is an especially important tool for the "cold-start problem." In these types of problems, such as launching a new marketing campaign or identifying potential new types of fraud or spam, we initially may not have any response to train a model. Over time, as data is collected, we can learn more about the system and build a traditional predictive model. But clustering helps us start the learning process more quickly by identifying population segments.

Unsupervised learning is also important as a building block for regression and classification techniques. With big data, if a small subpopulation is not well represented in the overall population, the trained model may not perform well for that subpopulation. With clustering, it is possible to identify and label subpopulations. Separate models can then be fit to the different subpopulations. Alternatively, the subpopulation can be represented with its own feature, forcing the overall model to explicitly consider subpopulation identity as a predictor.

Principal Components Analysis

Often, variables will vary together (covary), and some of the variation in one is actually duplicated by variation in another. Principal components analysis (PCA) is a technique to discover the way in which numeric variables covary.[1]

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

Key Terms for Principal Components Analysis

Principal component
A linear combination of the predictor variables.

Loadings
The weights that transform the predictors into the components.

Synonym
Weights

Screeplot
A plot of the variances of the components, showing the relative importance of the components.

The idea in PCA is to combine multiple numeric predictor variables into a smaller set of variables, which are weighted linear combinations of the original set. The smaller set of variables, the *principal components*, "explains" most of the variability of the full set of variables, reducing the dimension of the data. The weights used to form the principal components reveal the relative contributions of the original variables to the new principal components.

PCA was first proposed by Karl Pearson. In what was perhaps the first paper on unsupervised learning, Pearson recognized that in many problems there is variability in the predictor variables, so he developed PCA as a technique to model this variability. PCA can be viewed as the unsupervised version of linear discriminant analysis; see"Discriminant Analysis" on page 179.

A Simple Example

For two variables, X_1 and X_2, there are two principal components Z_i (i = 1 or 2):

$$Z_i = w_{i,1}X_1 + w_{i,2}X_2$$

The weights $\left(w_{i,1}, w_{i,2}\right)$ are known as the component *loadings*. These transform the original variables into the principal components. The first principal component, Z_1, is the linear combination that best explains the total variation. The second principal component, Z_2, explains the remaining variation (it is also the linear combination that is the worst fit).

It is also common to compute principal components on deviations from the means of the predictor variables, rather than on the values themselves.

You can compute principal components in R using the `princomp` function. The following performs a PCA on the stock price returns for Chevron (CVX) and ExxonMobil (XOM):

```
oil_px <- sp500_px[, c('CVX', 'XOM')]
pca <- princomp(oil_px)
pca$loadings

Loadings:
    Comp.1 Comp.2
CVX -0.747  0.665
XOM -0.665 -0.747
```

The weights for CVX and XOM for the first principal component are –0.747 and –0.665 and for the second principal component they are 0.665 and –0.747. How to interpret this? The first principal component is essentially an average of CVX and XOM, reflecting the correlation between the two energy companies. The second principal component measures when the stock prices of CVX and XOM diverge.

It is instructive to plot the principal components with the data:

```
loadings <- pca$loadings
ggplot(data=oil_px, aes(x=CVX, y=XOM)) +
  geom_point(alpha=.3) +
  stat_ellipse(type='norm', level=.99) +
  geom_abline(intercept = 0, slope = loadings[2,1]/loadings[1,1]) +
  geom_abline(intercept = 0, slope = loadings[2,2]/loadings[1,2])
```

The result is shown in Figure 7-1.

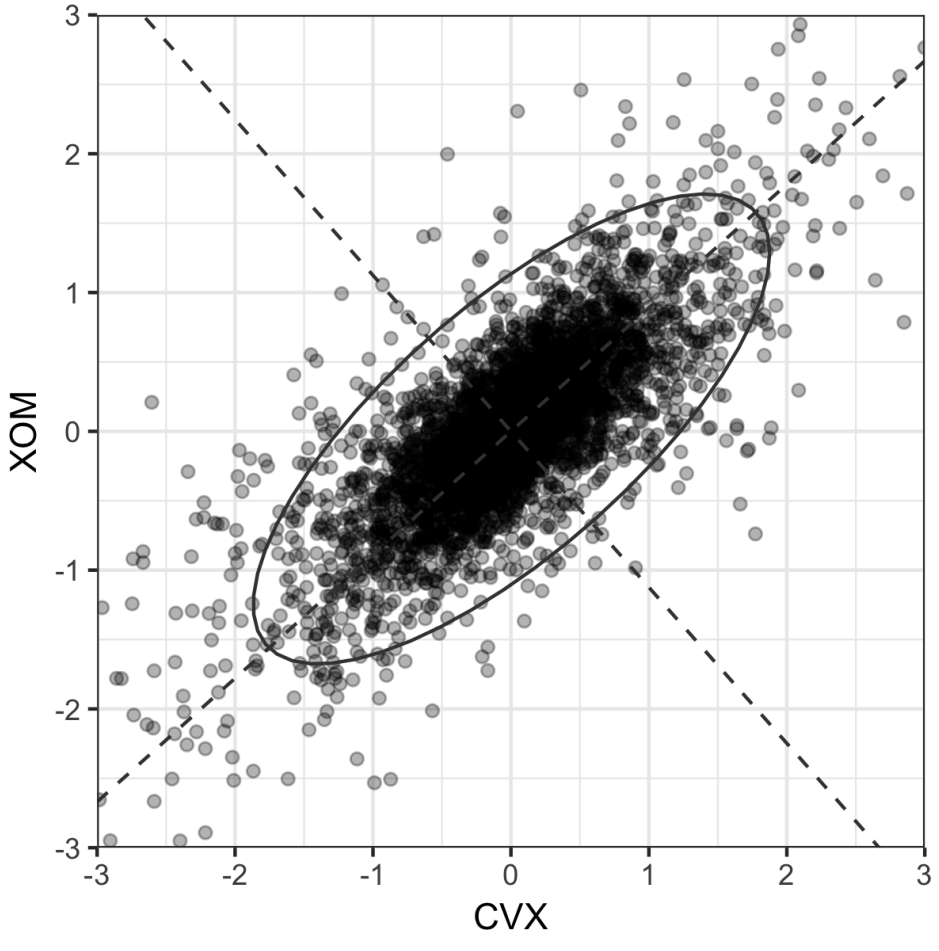


Figure 7-1. The principal components for the stock returns for Chevron and ExxonMobil

The solid dashed lines show the two principal components: the first one is along the long axis of the ellipse and the second one is along the short axis. You can see that a majority of the variability in the two stock returns is explained by the first principal component. This makes sense since energy stock prices tend to move as a group.

The weights for the first principal component are both negative, but reversing the sign of all the weights does not change the principal component. For example, using weights of 0.747 and 0.665 for the first principal component is equivalent to the negative weights, just as an infinite line defined by the origin and 1,1 is the same as one defined by the origin and –1, –1.

Computing the Principal Components

Going from two variables to more variables is straightforward. For the first component, simply include the additional predictor variables in the linear combination, assigning weights that optimize the collection of the covariation from all the predictor variables into this first principal component (*covariance* is the statistical term; see "Covariance Matrix" on page 180). Calculation of principal components is a classic statistical method, relying on either the correlation matrix of the data or the covariance matrix, and it executes rapidly, not relying on iteration. As noted earlier, it works only with numeric variables, not categorical ones. The full process can be described as follows:

1. In creating the first principal component, PCA arrives at the linear combination of predictor variables that maximizes the percent of total variance explained.
2. This linear combination then becomes the first "new" predictor, Z_1.
3. PCA repeats this process, using the same variables, with different weights to create a second new predictor, Z_2. The weighting is done such that Z_1 and Z_2 are uncorrelated.
4. The process continues until you have as many new variables, or components, Z_i as original variables X_i.
5. Choose to retain as many components as are needed to account for most of the variance.
6. The result so far is a set of weights for each component. The final step is to convert the original data into new principal component scores by applying the weights to the original values. These new scores can then be used as the reduced set of predictor variables.

Interpreting Principal Components

The nature of the principal components often reveals information about the structure of the data. There are a couple of standard visualization displays to help you glean insight about the principal components. One such method is a *Screeplot* to visualize the relative importance of principal components (the name derives from the resemblance of the plot to a scree slope). The following is an example for a few top companies in the S&P 500:

```
syms <- c( 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM',
   'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
top_sp <- sp500_px[row.names(sp500_px)>='2005-01-01', syms]
sp_pca <- princomp(top_sp)
screeplot(sp_pca)
```

As seen in Figure 7-2, the variance of the first principal component is quite large (as is often the case), but the other top principal components are significant.

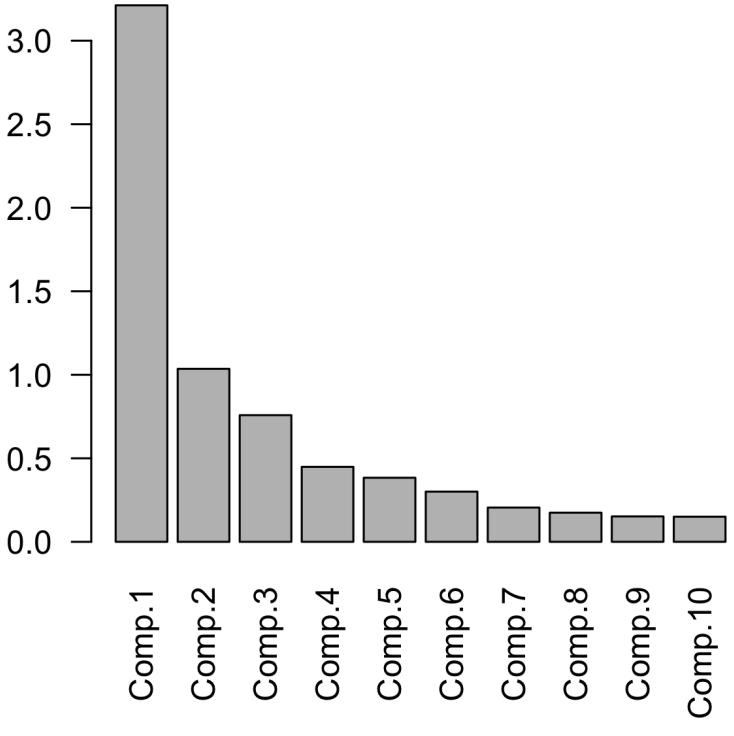


Figure 7-2. A screeplot for a PCA of top stocks from the S&P 500

It can be especially revealing to plot the weights of the top principal components. One way to do this is to use the `gather` function from the `tidyr` package in conjunction with `ggplot`:

```
library(tidyr)
loadings <- sp_pca$loadings[,1:5]
loadings$Symbol <- row.names(loadings)
loadings <- gather(loadings, "Component", "Weight", -Symbol)
ggplot(loadings, aes(x=Symbol, y=Weight)) +
  geom_bar(stat='identity') +
  facet_grid(Component ~ ., scales='free_y')
```

The loadings for the top five components are shown in Figure 7-3. The loadings for the first principal component have the same sign: this is typical for data in which all the columns share a common factor (in this case, the overall stock market trend). The second component captures the price changes of energy stocks as compared to the other stocks. The third component is primarily a contrast in the movements of Apple and CostCo. The fourth component contrasts the movements of Schlumberger to the

other energy stocks. Finally, the fifth component is mostly dominated by financial companies.

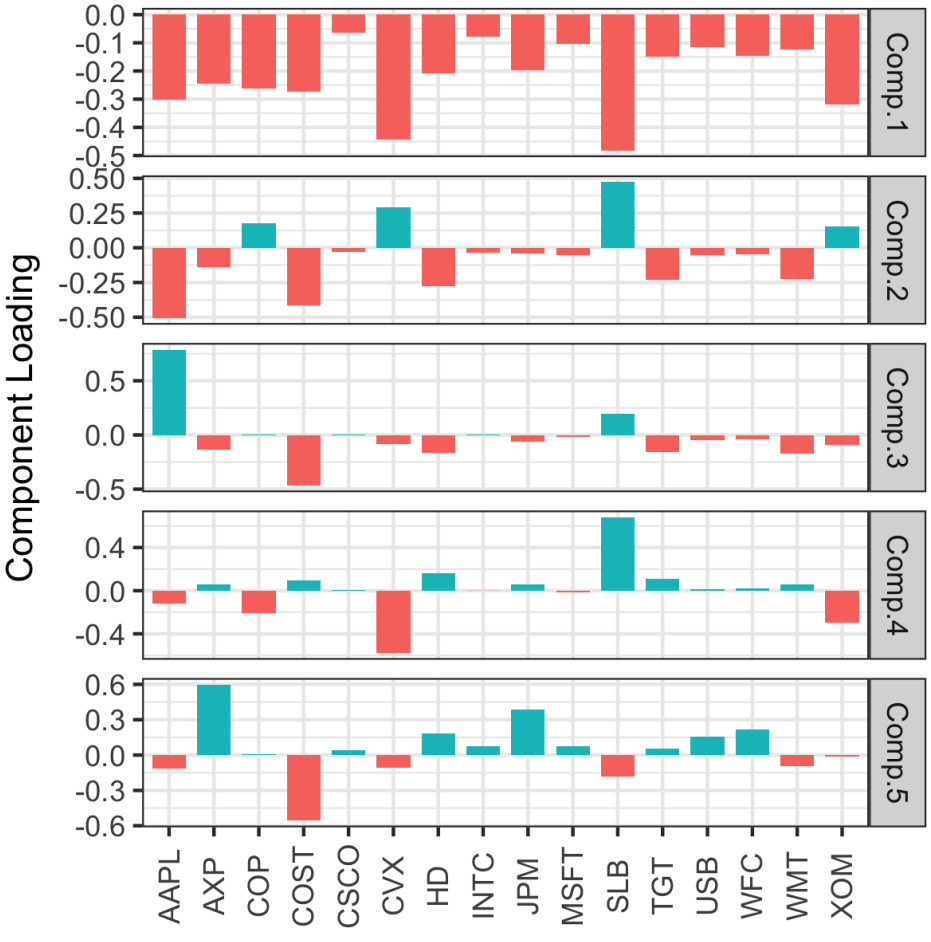


Figure 7-3. The loadings for the top five principal components of stock price returns

How Many Components to Choose?

If your goal is to reduce the dimension of the data, you must decide how many principal components to select. The most common approach is to use an ad hoc rule to select the components that explain "most" of the variance. You can do this visually through the screeplot; for example, in Figure 7-2, it would be natural to restrict the analysis to the top five components. Alternatively, you could select the top components such that the cumulative variance exceeds a threshold, such as 80%. Also, you can inspect the loadings to determine if the component has an intuitive interpretation. Cross-validation provides a more formal method to select the number of significant components (see "Cross-Validation" on page 138 for more).

Key Ideas for Principal Components

- Principal components are linear combinations of the predictor variables (numeric data only).
- They are calculated so as to minimize correlation between components, reducing redundancy.
- A limited number of components will typically explain most of the variance in the outcome variable.
- The limited set of principal components can then be used in place of the (more numerous) original predictors, reducing dimensionality.

Further Reading

For a detailed look at the use of cross-validation in principal components, see Rasmus Bro, K. Kjeldahl, A.K. Smilde, and Henk A. L. Kiers, "Cross-Validation of Component Models: A Critical Look at Current Methods", *Analytical and Bioanalytical Chemistry* 390, no. 5 (2008).

K-Means Clustering

Clustering is a technique to divide data into different groups, where the records in each group are similar to one another. A goal of clustering is to identify significant and meaningful groups of data. The groups can be used directly, analyzed in more depth, or passed as a feature or an outcome to a predictive regression or classification model. *K-means* is the first clustering method to be developed; it is still widely used,

owing its popularity to the relative simplicity of the algorithm and its ability to scale to large data sets.

Key Terms for K-Means Clustering

Cluster
A group of records that are similar.

Cluster mean
The vector of variable means for the records in a cluster.

K
The number of clusters.

K-means divides the data into *K* clusters by minimizing the sum of the squared distances of each record to the *mean* of its assigned cluster. The is referred to as the *within-cluster sum of squares* or *within-cluster SS*. *K*-means does not ensure the clusters will have the same size, but finds the clusters that are the best separated.

Normalization

It is typical to normalize (standardize) continuous variables by subtracting the mean and dividing by the standard deviation. Otherwise, variables with large scale will dominate the clustering process (see "Standardization (Normalization, Z-Scores)" on page 215).

A Simple Example

Start by considering a data set with n records and just two variables, x and y. Suppose we want to split the data into $K = 4$ clusters. This means assigning each record (x_i, y_i) to a cluster k. Given an assignment of n_k records to cluster k, the center of the cluster $(\bar{x}_k, \bar{y}_k)$ is the mean of the points in the cluster:

$$\bar{x}_k = \frac{1}{n_k} \sum_{i \in \text{Cluster } k} x_i$$

$$\bar{y}_k = \frac{1}{n_k} \sum_{i \in \text{Cluster } k} y_i$$

Cluster Mean

In clustering records with multiple variables (the typical case), the term *cluster mean* refers not to a single number, but to the vector of means of the variables.

The sum of squares within a cluster is given by:

$$SS_k = \sum_{i \in \text{Cluster } k} (x_i - \bar{x}_k)^2 + (y_i - \bar{y}_k)^2$$

K-means finds the assignment of records that minimizes within-cluster sum of squares across all four clusters $SS_1 + SS_2 + SS_3 + SS_4$.

$$\sum_{k=1}^{4} SS_i$$

K-means clustering can be used to gain insight into how the price movements of stocks tend to cluster. Note that stock returns are reported in a fashion that is, in effect, standardized, so we do not need to normalize the data. In R, *K*-means clustering can be performed using the `kmeans` function. For example, the following finds four clusters based on two variables: the stock returns for ExxonMobil (`XOM`) and Chevron (`CVX`):

```
df <- sp500_px[row.names(sp500_px)>='2011-01-01', c('XOM', 'CVX')]
km <- kmeans(df, centers=4)
```

The cluster assignment for each record is returned as the `cluster` component:

```
> df$cluster <- factor(km$cluster)
> head(df)
                 XOM        CVX cluster
2011-01-03 0.73680496  0.2406809       2
2011-01-04 0.16866845 -0.5845157       1
2011-01-05 0.02663055  0.4469854       2
2011-01-06 0.24855834 -0.9197513       1
2011-01-07 0.33732892  0.1805111       2
2011-01-10 0.00000000 -0.4641675       1
```

The first six records are assigned to either cluster 1 or cluster 2. The means of the clusters are also returned:

```
> centers <- data.frame(cluster=factor(1:4), km$centers)
> centers
  cluster        XOM        CVX
1       1 -0.3284864 -0.5669135
2       2  0.2410159  0.3342130
3       3 -1.1439800 -1.7502975
4       4  0.9568628  1.3708892
```

Clusters 1 and 3 represent "down" markets, while clusters 2 and 4 represent "up markets." In this example, with just two variables, it is straightforward to visualize the clusters and their means:

```
ggplot(data=df, aes(x=XOM, y=CVX, color=cluster, shape=cluster)) +
  geom_point(alpha=.3) +
  geom_point(data=centers,  aes(x=XOM, y=CVX), size=3, stroke=2)
```

The resulting plot, given by Figure 7-4, shows the cluster assignments and the cluster means.

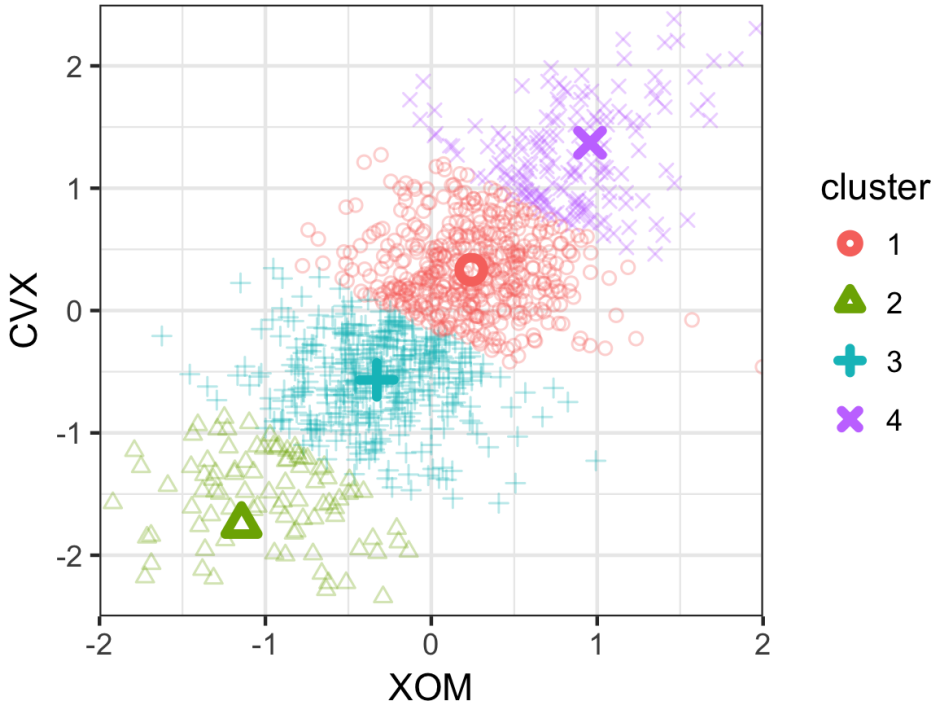


Figure 7-4. The clusters of K-means applied to stock price data for ExxonMobil and Chevron (the two cluster centers in the dense area are hard to distinguish)

K-Means Algorithm

In general, *K*-means can be applied to a data set with p variables $X_1, ..., X_p$. While the exact solution to *K*-means is computationally very difficult, heuristic algorithms provide an efficient way to compute a locally optimal solution.

The algorithm starts with a user-specified K and an initial set of cluster means, then iterates the following steps:

1. Assign each record to the nearest cluster mean as measured by squared distance.
2. Compute the new cluster means based on the assignment of records.

The algorithm converges when the assignment of records to clusters does not change.

For the first iteration, you need to specify an initial set of cluster means. Usually you do this by randomly assigning each record to one of the *K* clusters, then finding the means of those clusters.

Since this algorithm isn't guaranteed to find the best possible solution, it is recommended to run the algorithm several times using different random samples to initialize the algorithm. When more than one set of iterations is used, the *K*-means result is given by the iteration that has the lowest within-cluster sum of squares.

The `nstart` parameter to the R function `kmeans` allows you to specify the number of random starts to try. For example, the following code runs *K*-means to find 5 clusters using 10 different starting cluster means:

```
syms <- c( 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM', 'SLB', 'COP',
           'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
df <- sp500_px[row.names(sp500_px)>='2011-01-01', syms]
km <- kmeans(df, centers=5, nstart=10)
```

The function automatically returns the best solution out of the 10 different starting points. You can use the argument `iter.max` to set the maximum number of iterations the algorithm is allowed for each random start.

Interpreting the Clusters

An important part of cluster analysis can involve the interpretation of the clusters. The two most important outputs from `kmeans` are the sizes of the clusters and the cluster means. For the example in the previous subsection, the sizes of resulting clusters are given by this R command:

```
km$size
[1] 186 106 285 288 266
```

The cluster sizes are relatively balanced. Imbalanced clusters can result from distant outliers, or groups of records very distinct from the rest of the data—both may warrant further inspection.

You can plot the centers of the clusters using the `gather` function in conjunction with `ggplot`:

```
centers <- as.data.frame(t(centers))
names(centers) <- paste("Cluster", 1:5)
centers$Symbol <- row.names(centers)
centers <- gather(centers, "Cluster", "Mean", -Symbol)
centers$Color = centers$Mean > 0
ggplot(centers, aes(x=Symbol, y=Mean, fill=Color)) +
  geom_bar(stat='identity', position = "identity", width=.75) +
  facet_grid(Cluster ~ ., scales='free_y')
```

The resulting plot is shown in Figure 7-5 and reveals the nature of each cluster. For example, clusters 1 and 2 correspond to days on which the market is down and up,

respectively. Clusters 3 and 5 are characterized by up-market days for consumer stocks and down-market days for energy stocks, respectively. Finally, cluster 4 captures the days in which energy stocks were up and consumer stocks were down.

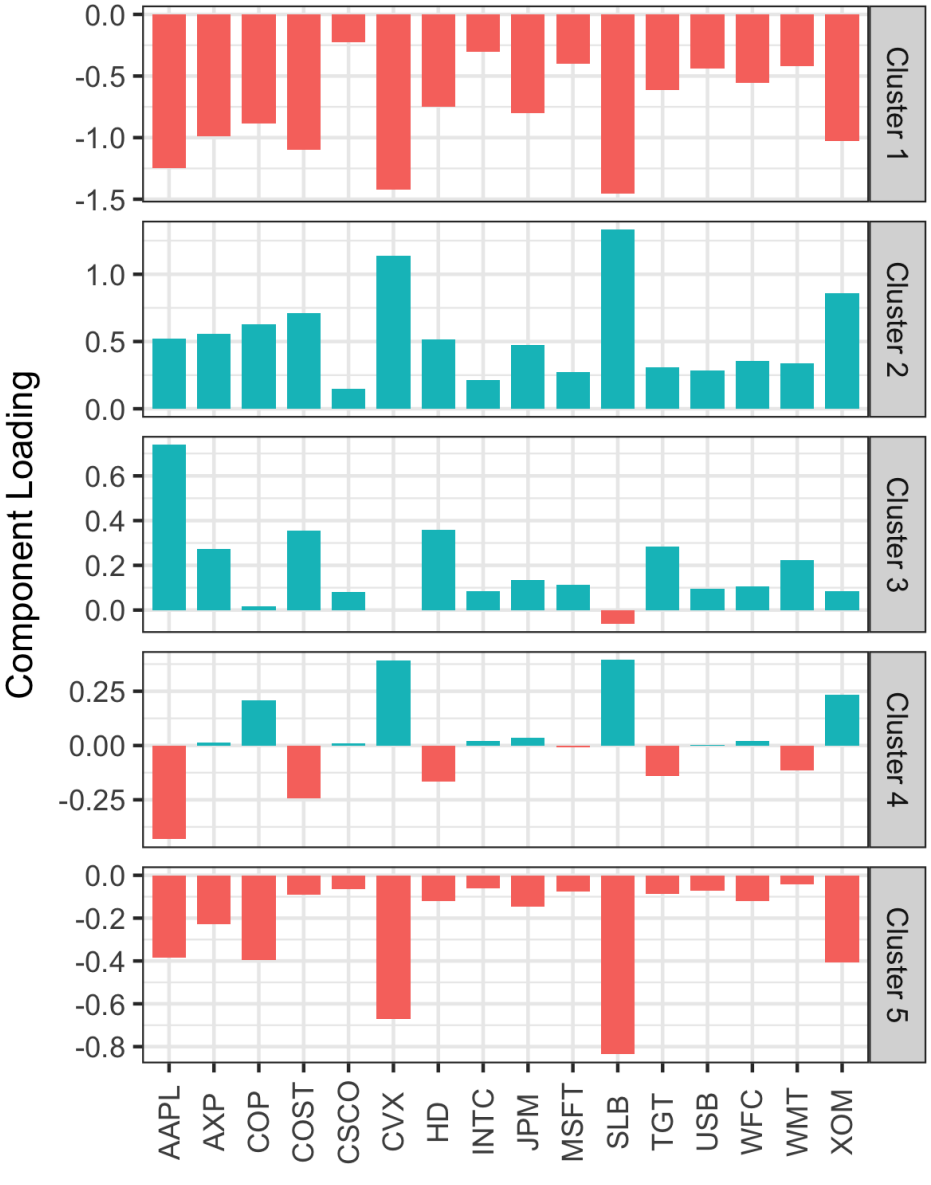


Figure 7-5. The means of the variables in each cluster ("cluster means")

Cluster Analysis versus PCA

The plot of cluster means is similar in spirit to looking at the loadings for principal component analysis (PCA); see "Interpreting Principal Components" on page 254. A major distinction is that unlike with PCA, the sign of the cluster means is meaningful. PCA identifies principal directions of variation, whereas cluster analysis finds groups of records located near one another.

Selecting the Number of Clusters

The *K*-means algorithm requires that you specify the number of clusters *K*. Sometimes the number of clusters is driven by the application. For example, a company managing a sales force might want to cluster customers into "personas" to focus and guide sales calls. In such a case, managerial considerations would dictate the number of desired customer segments—for example, two might not yield useful differentiation of customers, while eight might be too many to manage.

In the absence of a cluster number dictated by practical or managerial considerations, a statistical approach could be used. There is no single standard method to find the "best" number of clusters.

A common approach, called the *elbow method*, is to identify when the set of clusters explains "most" of the variance in the data. Adding new clusters beyond this set contributes relatively little incremental contribution in the variance explained. The elbow is the point where the cumulative variance explained flattens out after rising steeply, hence the name of the method.

Figure 7-6 shows the cumulative percent of variance explained for the default data for the number of clusters ranging from 2 to 15. Where is the elbow in this example? There is no obvious candidate, since the incremental increase in variance explained drops gradually. This is fairly typical in data that does not have well-defined clusters. This is perhaps a drawback of the elbow method, but it does reveal the nature of the data.

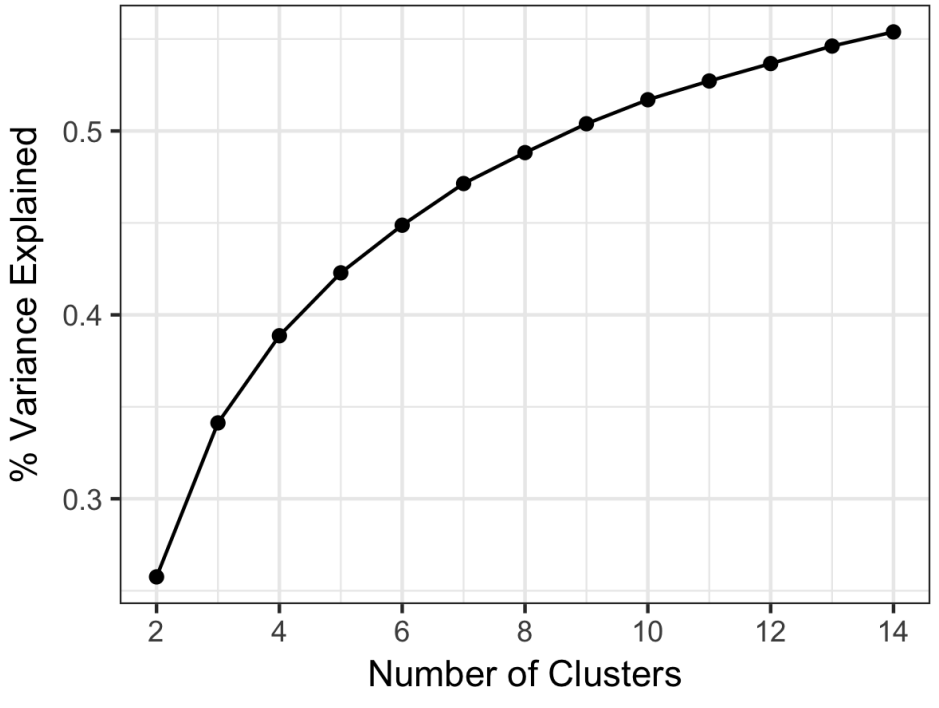


Figure 7-6. The elbow method applied to the stock data

In R, the `kmeans` function doesn't provide a single command for applying the elbow method, but it can be readily applied from the output of `kmeans` as shown here:

```
pct_var <- data.frame(pct_var = 0,
                      num_clusters=2:14)
totalss <- kmeans(df, centers=14, nstart=50, iter.max = 100)$totss
for(i in 2:14){
  pct_var[i-1, 'pct_var'] <- kmeans(df, centers=i, nstart=50, iter.max = 100)
    $betweenss/totalss
}
```

In evaluating how many clusters to retain, perhaps the most important test is this: how likely are the clusters to be replicated on new data? Are the clusters interpretable, and do they relate to a general characteristic of the data, or do they just reflect a specific instance? You can assess this, in part, using cross-validation; see "Cross-Validation" on page 138.

In general, there is no single rule that will reliably guide how many clusters to produce.

There are several more formal ways to determine the number of clusters based on statistical or information theory. For example, Robert Tibshirani, Guenther Walther, and Trevor Hastie (*http://www.stanford.edu/~hastie/Papers/gap.pdf*) propose a "gap" statistic based on statistical theory to identify the elbow. For most applications, a theoretical approach is probably not necessary, or even appropriate.

Key Ideas for K-Means Clustering

- The number of desired clusters, *K*, is chosen by the user.
- The algorithm develops clusters by iteratively assigning records to the nearest cluster mean until cluster assignments do not change.
- Practical considerations usually dominate the choice of *K*; there is no statistically determined optimal number of clusters.

Hierarchical Clustering

Hierarchical clustering is an alternative to *K*-means that can yield very different clusters. Hierarchical clustering is more flexible than *K*-means and more easily accommodates non-numerical variables. It is more sensitive in discovering outlying or aberrant groups or records. Hierarchical clustering also lends itself to an intuitive graphical display, leading to easier interpretation of the clusters.

Key Terms for Hierarchical Clustering

Dendrogram
: A visual representation of the records and the hierarchy of clusters to which they belong.

Distance
: A measure of how close one *record* is to another.

Dissimilarity
: A measure of how close one *cluster* is to another.

Hierarchical clustering's flexibility comes with a cost, and hierarchical clustering does not scale well to large data sets with millions of records. For even modest-sized data with just tens of thousands of records, hierarchical clustering can require intensive computing resources. Indeed, most of the applications of hierarchical clustering are focused on relatively small data sets.

A Simple Example

Hierarchical clustering works on a data set with n records and p variables and is based on two basic building blocks:

- A distance metric $d_{i,j}$ to measure the distance beween two records i and j.
- A dissimilarity metric $D_{A,B}$ to measure the difference between two clusters A and B based on the distances $d_{i,j}$ between the members of each cluster.

For applications involving numeric data, the most importance choice is the dissimilarity metric. Hierarchical clustering starts by setting each record as its own cluster and iterates to combine the least dissimilar clusters.

In R, the `hclust` function can be used to perform hierarchical clustering. One big difference with `hclust` versus `kmeans` is that it operates on the pairwise distances $d_{i,j}$ rather than the data itself. You can compute these using the `dist` function. For example, the following applies hierarchical clustering to the stock returns for a set of companies:

```
syms1 <- c('GOOGL', 'AMZN', 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX',
           'XOM', 'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP',
           'WMT', 'TGT', 'HD', 'COST')
# take transpose: to cluster companies, we need the stocks along the rows
df <- t(sp500_px[row.names(sp500_px)>='2011-01-01', syms1])
d <- dist(df)
hcl <- hclust(d)
```

Clustering algorithms will cluster the records (rows) of a data frame. Since we want to cluster the companies, we need to *transpose* the data frame and put the stocks along the rows and the dates along the columns.

The Dendrogram

Hierarchical clustering lends itself to a natural graphical display as a tree, referred to as a *dendrogram*. The name comes from the Greek words *dendro* (tree) and *gramma* (drawing). In R, you can easily produce this using the `plot` command:

```
plot(hcl)
```

The result is shown in Figure 7-7. The leaves of the tree correspond to the records. The length of the branch in the tree indicates the degree of dissimilarity between corresponding clusters. The returns for Google and Amazon are quite dissimilar to the returns for the other stocks. The other stocks fall into natural groups: energy stocks, financial stocks, and consumer stocks are all separated into their own subtrees.

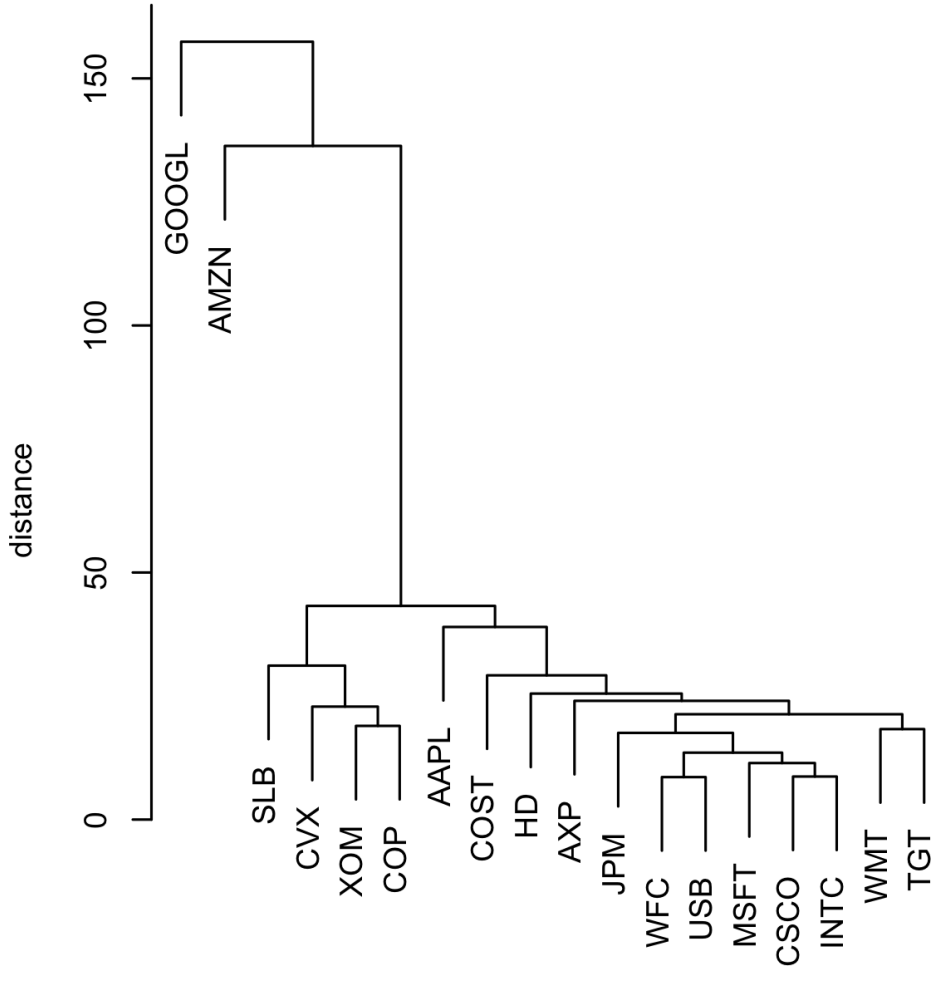


Figure 7-7. A dendogram of stocks

In contrast to *K*-means, it is not necessary to prespecify the number of clusters. To extract a specific number of clusters, you can use the `cutree` function:

```
cutree(hcl, k=4)
GOOGL  AMZN  AAPL  MSFT  CSCO  INTC   CVX   XOM   SLB   COP   JPM   WFC
    1     2     3     3     3     3     4     4     4     4     3     3
  USB   AXP   WMT   TGT    HD  COST
    3     3     3     3     3     3
```

The number of clusters to extract is set to 4, and you can see that Google and Amazon each belong to their own cluster. The oil stocks (XOM, CVS, SLB, COP) all belong to another cluster. The remaining stocks are in the fourth cluster.

The Agglomerative Algorithm

The main algorithm for hierarchical clustering is the *agglomerative* algorithm, which iteratively merges similar clusters. The agglomerative algorithm begins with each record constituting its own single-record cluster, then builds up larger and larger clusters. The first step is to calculate distances between all pairs of records.

For each pair of records $\left(x_1, x_2, ..., x_p\right)$ and $\left(y_1, y_2, ..., y_p\right)$, we measure the distance between the two records, $d_{x,y}$, using a distance metric (see "Distance Metrics" on page 213). For example, we can use Euclidian distance:

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + \left(x_p - y_p\right)^2}$$

We now turn to inter-cluster distance. Consider two clusters A and B, each with a distinctive set of records, $A = (a_1, a_2, ..., a_m)$ and $B = \left(b_1, b_2, ..., b_q\right)$. We can measure the dissimilarity between the clusters $D(A, B)$ by using the distances between the members of A and the members of B.

One measure of dissimilarity is the *complete-linkage* method, which is the maximum distance across all pairs of records between A and B:

$$D(A,B) = \max d\left(a_i, b_j\right) \text{ for all pairs } i, j$$

This defines the dissimilarity as the biggest difference between all pairs.

The main steps of the agglomerative algorithm are:

1. Create an initial set of clusters with each cluster consisting of a single record for all records in the data.
2. Compute the dissimilarity $D\left(C_k, C_\ell\right)$) between all pairs of clusters k, ℓ.
3. Merge the two clusters C_k and C_ℓ that are least dissimilar as measured by $D\left(C_k, C_\ell\right)$).
4. If we have more than one cluster remaining, return to step 2. Otherwise, we are done.

Measures of Dissimilarity

There are four common measures of dissimilarity: *complete linkage*, *single linkage*, *average linkage*, and *minimum variance*. These (plus other measures) are all supported by most hierarchical clustering software, including `hclust`. The complete

linkage method defined earlier tends to produce clusters with members that are similar. The single linkage method is the minimum distance between the records in two clusters:

$$D(A, B) = \min d\left(a_i, b_j\right) \text{ for all pairs } i, j$$

This is a "greedy" method and produces clusters that can contain quite disparate elements. The average linkage method is the average of all distance pairs and represents a compromise between the single and complete linkage methods. Finally, the minimum variance method, also referred to as *Ward's* method, is similar to *K*-means since it minimizes the within-cluster sum of squares (see "K-Means Clustering" on page 257).

Figure 7-8 applies hierarchical clustering using the four measures to the ExxonMobil and Chevron stock returns. For each measure, four clusters are retained.

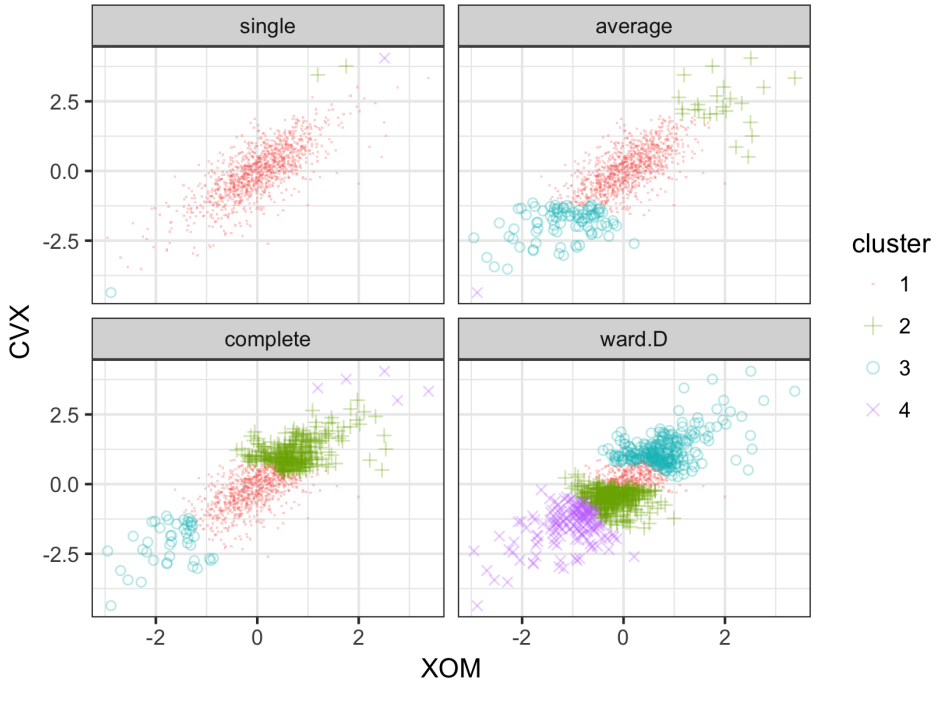


Figure 7-8. A comparison of measures of dissimilarity applied to stock data

The results are strikingly different: the single linkage measure assigns almost all of the points to a single cluster. Except for the minimum variance method (`Ward.D`), all measures end up with at least one cluster with just a few outlying points. The mini-

mum variance method is most similar to the *K*-means cluster; compare with Figure 7-4.

Key Ideas for Hierarchical Clustering

- Start with every record in its own cluster.
- Progressively, clusters are joined to nearby clusters until all records belong to a single cluster (the agglomerative algorithm).
- The agglomeration history is retained and plotted, and the user (without specifying the number of clusters beforehand) can visualize the number and structure of clusters at different stages.
- Inter-cluster distances are computed in different ways, all relying on the set of all inter-record distances.

Model-Based Clustering

Clustering methods such as hierarchical clustering and *K*-means are based on heuristics and rely primarily on finding clusters whose members are close to one another, as measured directly with the data (no probability model involved). In the past 20 years, significant effort has been devoted to developing *model-based clustering* methods. Adrian Raftery and other researchers at the University of Washington made critical contributions to model-based clustering, including both theory and software. The techniques are grounded in statistical theory and provide more rigorous ways to determine the nature and number of clusters. They could be used, for example, in cases where there might be one group of records that are similar to one another but not necessarily close to one another (e.g., tech stocks with high variance of returns), and another group of records that is similar, and also close (e.g., utility stocks with low variance).

Multivariate Normal Distribution

The most widely used model-based clustering methods rest on the *multivariate normal* distribution. The multivariate normal distribution is a generalization of the normal distribution to set of p variables $X_1, X_2, ..., X_p$. The distribution is defined by a set of means $\mu = \mu_\mathbf{1}, \mu_\mathbf{2}, ..., \mu_\mathbf{p}$ and a covariance matrix Σ. The covariance matrix is a measure of how the variables correlate with each other (see "Covariance Matrix" on page 180 for details on the covariance). The covariance matrix Σ consists of p variances $\sigma_1^2, \sigma_2^2, ..., \sigma_p^2$ and covariances $\sigma_{i,j}$ for all pairs of variables $i \neq j$. With the variables put along the rows and duplicated along the columns, the matrix looks like this:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \cdots & \sigma_{1,p} \\ \sigma_{2,1} & \sigma_2^2 & \cdots & \sigma_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p,1} & \sigma_{p,2}^2 & \cdots & \sigma_p^2 \end{bmatrix}$$

Since a covariance matrix is symmetric, and $\sigma_{i,j} = \sigma_{j,i}$, there are only $p \times (p-1) - p$ covariance terms. In total, the covariance matrix has $p \times (p-1)$ parameters. The distribution is denoted by:

$$\left(X_1, X_2, ..., X_p\right) \widetilde{N}_p(\mu, \Sigma)$$

This is a symbolic way of saying that the variables are all normally distributed, and the overall distribution is fully described by the vector of variable means and the covariance matrix.

Figure 7-9 shows the probability contours for a multivariate normal distribution for two variables X and Y (the 0.5 probability contour, for example, contains 50% of the distribution).

The means are $\mu_x = 0.5$ and $\mu_y = -0.5$ and the covariance matrix is:

$$\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

Since the covariance σ_{xy} is positive, X and Y are positively correlated.

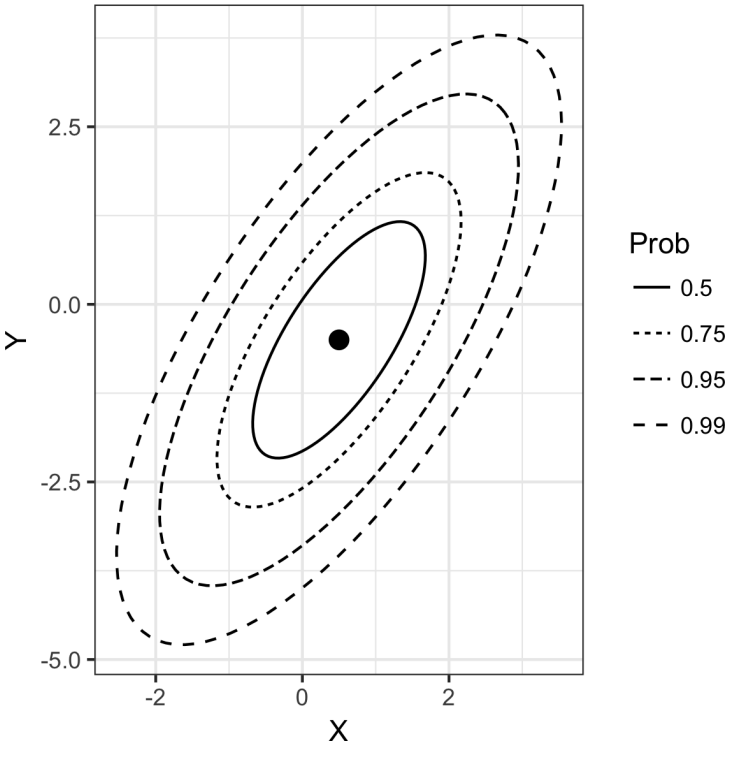


Figure 7-9. Probability contours for a two-dimensional normal distribution

Mixtures of Normals

The key idea behind model-based clustering is that each record is assumed to be distributed as one of K multivariate-normal distributions, where K is the number of clusters. Each distribution has a different mean μ and covariance matrix Σ. For example, if you have two variables, X and Y, then each row (X_i, Y_i) is modeled as having been sampled from one of K distributions $N_1(\mu_1), \Sigma_1), N_1(\mu_2), \Sigma_2), ..., N_1(\mu_K), \Sigma_K)$.

R has a very rich package for model-based clustering called `mclust`, originally developed by Chris Fraley and Adrian Raftery. With this package, we can apply model-based clustering to the stock return data we previously analyzed using K-means and hierarchical clustering:

```
> library(mclust)
> df <- sp500_px[row.names(sp500_px)>='2011-01-01', c('XOM', 'CVX')]
> mcl <- Mclust(df)
> summary(mcl)
Mclust VEE (ellipsoidal, equal shape and orientation) model with 2 components:
```

```
 log.likelihood    n df       BIC       ICL
      -2255.134 1131  9 -4573.546 -5076.856

Clustering table:
  1   2
963 168
```

If you execute this code, you will notice that the computation takes significiantly longer than other procedures. Extracting the cluster assignments using the `predict` function, we can visualize the clusters:

```
cluster <- factor(predict(mcl)$classification)
ggplot(data=df, aes(x=XOM, y=CVX, color=cluster, shape=cluster)) +
  geom_point(alpha=.8)
```

The resulting plot is shown in Figure 7-10. There are two clusters: one cluster in the middle of the data, and a second cluster in the outer edge of the data. This is very different from the clusters obtained using *K*-means (Figure 7-4) and hierarchical clustering (Figure 7-8), which find clusters that are compact.

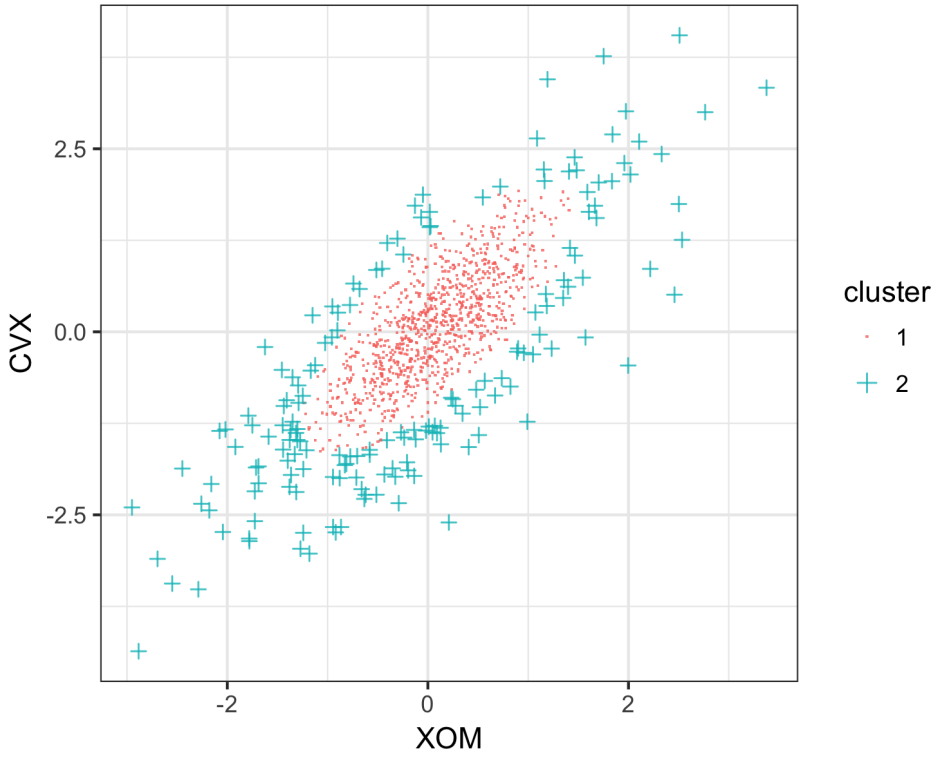


Figure 7-10. Two clusters are obtained for stock return data using mclust

You can extract the parameters to the normal distributions using the `summary` function:

```
> summary(mcl, parameters=TRUE)$mean
          [,1]        [,2]
XOM 0.05783847 -0.04374944
CVX 0.07363239 -0.21175715
> summary(mcl, parameters=TRUE)$variance
, , 1
          XOM       CVX
XOM 0.3002049 0.3060989
CVX 0.3060989 0.5496727
, , 2

         XOM      CVX
XOM 1.046318 1.066860
CVX 1.066860 1.915799
```

The distributions have similar means and correlations, but the second distribution has much larger variances and covariances.

The clusters from `mclust` may seem surprising, but in fact, they illustrate the statistical nature of the method. The goal of model-based clustering is to find the best-fitting set of multivariate normal distributions. The stock data appears to have a normal-looking shape: see the contours of Figure 7-9. In fact, though, stock returns have a longer-tailed distribution than a normal distribution. To handle this, `mclust` fits a distribution to the bulk of the data, but then fits a second distribution with a bigger variance.

Selecting the Number of Clusters

Unlike *K*-means and hierarchical clustering, `mclust` automatically selects the number of clusters (in this case, two). It does this by choosing the number of clusters for which the *Bayesian Information Criteria* (*BIC*) has the largest value. BIC (similar to AIC) is a general tool to find the best model amongst a candidate set of models. For example, AIC (or BIC) is commonly used to select a model in stepwise regression; see "Model Selection and Stepwise Regression" on page 139. BIC works by selecting the best-fitting model with a penalty for the number of parameters in the model. In the case of model-based clustering, adding more clusters will always improve the fit at the expense of introducing additional parameters in the model.

You can plot the BIC value for each cluster size using a function in `hclust`:

```
plot(mcl, what='BIC', ask=FALSE)
```

The number of clusters—or number of different multivariate normal models (components)—is shown on the x-axis (see Figure 7-11).

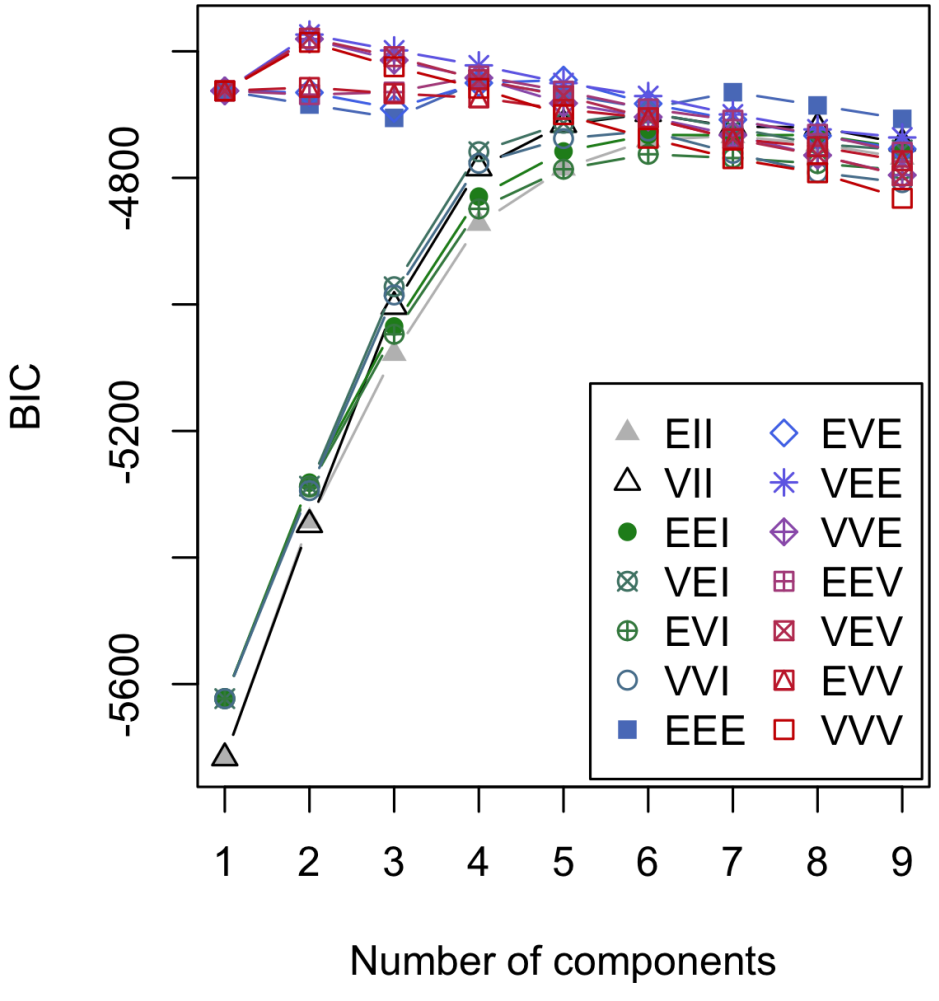


Figure 7-11. BIC scores for the stock return data for different numbers of clusters (components)

This plot is similar to the elbow plot used to identify the number of clusters to choose for *K*-means, except the value being plotted is BIC instead of percent of variance explained (see Figure 7-6). One big difference is that instead of one line, `mclust` shows 14 different lines! This is because `mclust` is actually fitting 14 different models for each cluster size, and ultimately it chooses the best-fitting model.

Why does `mclust` fit so many models to determine the best set of multivariate normals? It's because there are different ways to parameterize the covariance matrix Σ for fitting a model. For the most part, you do not need to worry about the details of the

models and can simply use the model chosen by `mclust`. In this example, according to BIC, three different models (called VEE, VEV, and VVE) give the best fit using two components.

Model-based clustering is a rich and rapidly developing area of study, and the coverage in this text only spans a small part of the field. Indeed, the `mclust` help file is currently 154 pages long. Navigating the nuances of model-based clustering is probably more effort than is needed for most problems encountered by data scientists.

Model-based clustering techniques do have some limitations. The methods require an underlying assumption of a model for the data, and the cluster results are very dependent on that assumption. The computations requirements are higher than even hierarchical clustering, making it difficult to scale to large data. Finally, the algorithm is more sophisticated and less accessible than that of other methods.

Key Ideas for Model-Based Clustering

- Clusters are assumed to derive from different data-generating processes with different probability distributions.
- Different models are fit, assuming different numbers of (typically normal) distributions.
- The method chooses the model (and the associated number of clusters) that fits the data well without using too many parameters (i.e., overfitting).

Further Reading

For more detail on model-based clustering, see the `mclust` documentation.

Scaling and Categorical Variables

Unsupervised learning techniques generally require that the data be appropriately scaled. This is different from many of the techniques for regression and classification in which scaling is not important (an exception is *K*-nearest neighbors; see "K-Nearest Neighbors" on page 210).

Key Terms for Scaling Data

Scaling
: Squashing or expanding data, usually to bring multiple variables to the same scale.

Normalization
: One method of scaling—subtracting the mean and dividing by the standard deviation.

Synonym
: Standardization

Gower's distance
: A scaling algorithm applied to mixed numeric and categoprical data to bring all variables to a 0–1 range.

For example, with the personal loan data, the variables have widely different units and magnitude. Some variables have relatively small values (e.g., number of years employed), while others have very large values (e.g., loan amount in dollars). If the data is not scaled, then the PCA, *K*-means, and other clustering methods will be dominated by the variables with large values and ignore the variables with small values.

Categorical data can pose a special problem for some clustering procedures. As with *K*-nearest neighbors, unordered factor variables are generally converted to a set of binary (0/1) variables using one hot encoding (see "One Hot Encoder" on page 214). Not only are the binary variables likely on a different scale from other data, the fact that binary variables have only two values can prove problematic with techniques such as PCA and *K*-means.

Scaling the Variables

Variables with very different scale and units need to be normalized appropriately before you apply a clustering procedure. For example, let's apply `kmeans` to a set of data of loan defaults without normalizing:

```
df <- defaults[, c('loan_amnt', 'annual_inc', 'revol_bal', 'open_acc',
                   'dti', 'revol_util')]
km <- kmeans(df, centers=4, nstart=10)
centers <- data.frame( size=km$size, km$centers)
round(centers, digits=2)
  size loan_amnt annual_inc revol_bal open_acc   dti revol_util
1    55  23157.27  491522.49  83471.07    13.35  6.89      58.74
2  1218  21900.96  165748.53  38299.44    12.58 13.43      63.58
```

```
3  7686  18311.55   83504.68  19685.28    11.68 16.80      62.18
4 14177  10610.43   42539.36  10277.97     9.60 17.73      58.05
```

The variables `annual_inc` and `revol_bal` dominate the clusters, and the clusters have very different sizes. Cluster 1 has only 55 members with comparatively high income and revolving credit balance.

A common approach to scaling the variables is to convert them to *z*-scores by subtracting the mean and dividing by the standard deviation. This is termed standardization or normalization (see "Standardization (Normalization, Z-Scores)" on page 215 for more discussion about using *z*-scores):

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

See what happens to the clusters when `kmeans` is applied to the normalized data:

```
df0 <- scale(df)
km0 <- kmeans(df0, centers=4, nstart=10)
centers0 <-scale(km0$centers, center=FALSE,
                 scale=1/attr(df0, 'scaled:scale'))
centers0 <- scale(centers0, center=-attr(df0, 'scaled:center'), scale=F)
data.frame(size=km0$size, centers0)
  size loan_amnt annual_inc revol_bal open_acc   dti revol_util
1 5429  10393.60   53689.54   6077.77     8.69 11.35      30.69
2 6396  13310.43   55522.76  16310.95    14.25 24.27      59.57
3 7493  10482.19   51216.95  11530.17     7.48 15.79      77.68
4 3818  25933.01  116144.63  32617.81    12.44 16.25      66.01
```

The cluster sizes are more balanced, and the clusters are not just dominated by `annual_inc` and `revol_bal`, revealing more interesting structure in the data. Note that the centers are rescaled to the original units in the preceding code. If we had left them unscaled, the resulting values would be in terms of *z*-scores, and therefore less interpretable.

Scaling is also important for PCA. Using the *z*-scores is equivalent to using the correlation matrix (see "Correlation" on page 29) instead of the covariance matrix in computing the principal components. Software to compute PCA usually has an option to use the correlation matrix (in R, the `princomp` function has the argument `cor`).

Dominant Variables

Even in cases where the variables are measured on the same scale and accurately reflect relative importance (e.g., movement to stock prices), it can sometimes be useful to rescale the variables.

Suppose we add Alphabet (GOOGL) and Amazon (AMZN) to the analysis in "Interpreting Principal Components" on page 254.

```
syms <- c('AMZN', 'GOOGL' 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM',
   'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
top_sp1 <- sp500_px[row.names(sp500_px)>='2005-01-01', syms]
sp_pca1 <- princomp(top_sp1)
screeplot(sp_pca1)
```

The screeplot displays the variances for the top principal components. In this case, the screeplot in Figure 7-12 reveals that the variances of the first and second components are much larger than the others. This often indicates that one or two variables dominate the loadings. This is, indeed, the case in this example:

```
round(sp_pca1$loadings[,1:2], 3)
      Comp.1 Comp.2
GOOGL  0.781  0.609
AMZN   0.593 -0.792
AAPL   0.078  0.004
MSFT   0.029  0.002
CSCO   0.017 -0.001
INTC   0.020 -0.001
CVX    0.068 -0.021
XOM    0.053 -0.005
...
```

The first two principal components are almost completely dominated by GOOGL and AMZN. This is because the stock price movements of GOOGL and AMZN dominate the variability.

To handle this situation, you can either include them as is, rescale the variables (see "Scaling the Variables" on page 277), or exclude the dominant variables from the analysis and handle them separately. There is no "correct" approach, and the treatment depends on the application.

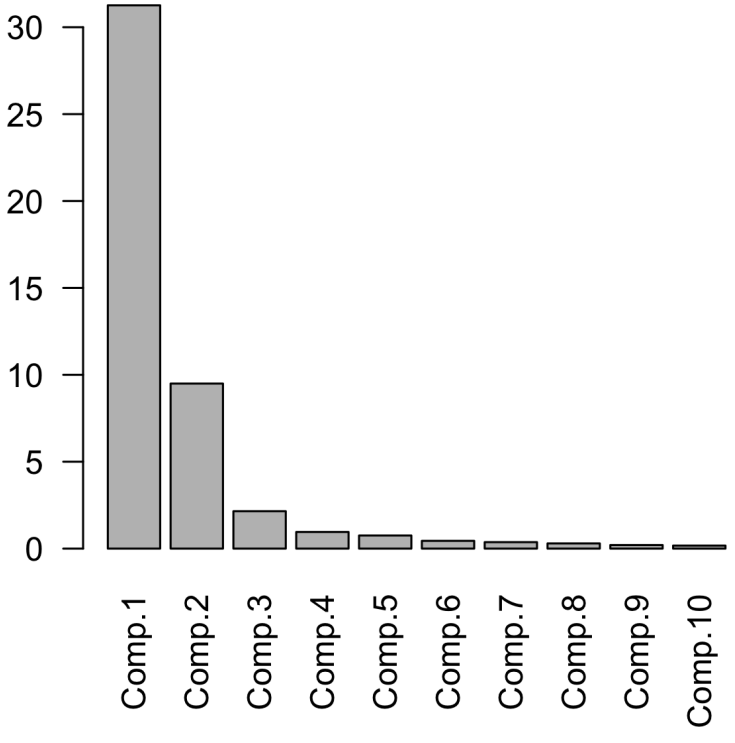


Figure 7-12. A screeplot for a PCA of top stocks from the S&P 500 including GOOGL and AMZN

Categorical Data and Gower's Distance

In the case of categorical data, you must convert it to numeric data, either by ranking (for an ordered factor) or by encoding as a set of binary (dummy) variables. If the data consists of mixed continuous and binary variables, you will usually want to scale the variables so that the ranges are similar; see "Scaling the Variables" on page 277. One popular method is to use *Gower's distance*.

The basic idea behind Gower's distance is to apply a different distance metric to each variable depending on the type of data:

- For numeric variables and ordered factors, distance is calculated as the absolute value of the difference between two records (*Manhattan distance*).
- For categorical variables, the distance is 1 if the categories between two records are different and the distance is 0 if the categories are the same.

Gower's distance is computed as follows:

1. Compute the distance $d_{i,j}$ for all pairs of variables i and j for each record.
2. Scale each pair $d_{i,j}$ so the minimum is 0 and the maximum is 1.
3. Add the pairwise scaled distances between variables together, either using a simple or weighted mean, to create the distance matrix.

To illustrate Gower's distance, take a few rows from the loan data:

```
> x = defaults[1:5, c('dti', 'payment_inc_ratio', 'home', 'purpose')]
> x
# A tibble: 5 × 4
    dti payment_inc_ratio   home            purpose
  <dbl>             <dbl> <fctr>             <fctr>
1  1.00           2.39320   RENT                car
2  5.55           4.57170    OWN     small_business
3 18.08           9.71600   RENT              other
4 10.08          12.21520   RENT debt_consolidation
5  7.06           3.90888   RENT              other
```

The function `daisy` in the `cluster` package can be used to compute Gower's distance:

```
> library(cluster)
> daisy(x, metric='gower')
Dissimilarities :
          1         2         3         4
2 0.6220479
3 0.6863877 0.8143398
4 0.6329040 0.7608561 0.4307083
5 0.3772789 0.5389727 0.3091088 0.5056250
```

All distances are between 0 and 1. The pair of records with the biggest distance is 2 and 3: neither has the same values for `home` or `purpose` and they have very different levels of `dti` (debt-to-income) and `payment_inc_ratio`. Records 3 and 5 have the smallest distance because they share the same values for `home` or `purpose`.

You can apply hierarchical clustering (see "Hierarchical Clustering" on page 265) to the resulting distance matrix using `hclust` to the output from `daisy`:

```
df <- defaults[sample(nrow(defaults), 250),
               c('dti', 'payment_inc_ratio', 'home', 'purpose')]
d = daisy(df, metric='gower')
hcl <- hclust(d)
dnd <- as.dendrogram(hcl)
plot(dnd, leaflab='none')
```

The resulting dendrogram is shown in Figure 7-13. The individual records are not distinguishable on the x-axis, but we can examine the records in one of the subtrees (on the left, using a "cut" of 0.5), with this code:

```
> df[labels(dnd_cut$lower[[1]]),]
# A tibble: 9 × 4
```

```
     dti payment_inc_ratio   home purpose
   <dbl>             <dbl> <fctr>  <fctr>
1 24.57           0.83550   RENT   other
2 34.95           5.02763   RENT   other
3  1.51           2.97784   RENT   other
4  8.73          14.42070   RENT   other
5 12.05           9.96750   RENT   other
6 10.15          11.43180   RENT   other
7 19.61          14.04420   RENT   other
8 20.92           6.90123   RENT   other
9 22.49           9.36000   RENT   other
```

This subtree entirely consists of renters with a loan purpose labeled as "other." While strict separation is not true of all subtrees, this illustrates that the categorical variables tend to be grouped together in the clusters.

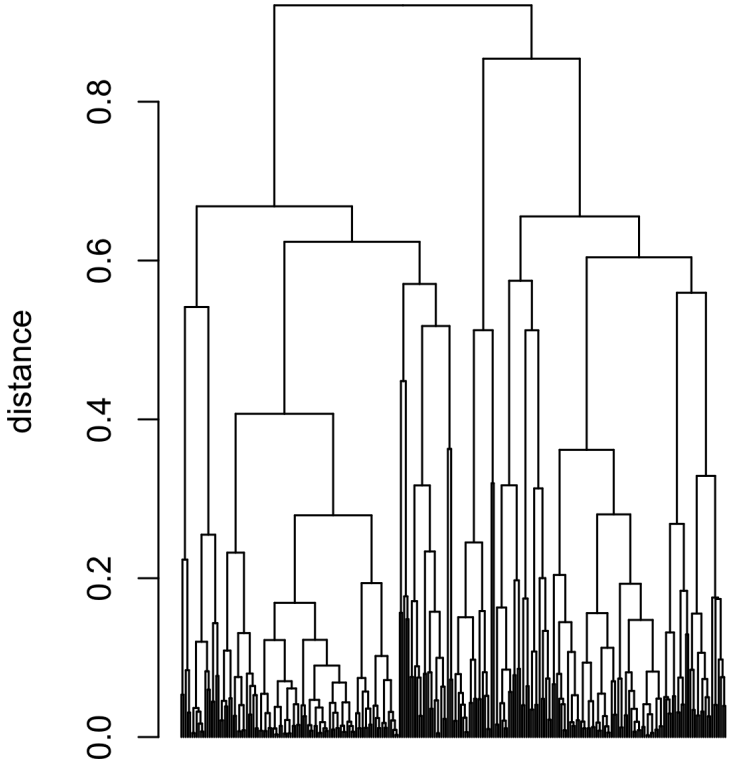


Figure 7-13. A dendrogram of hclust applied to a sample of loan default data with mixed variable types

Problems with Clustering Mixed Data

K-means and PCA are most appropriate for continuous variables. For smaller data sets, it is better to use hierarchical clustering with Gower's distance. In principle, there is no reason why *K*-means can't be applied to binary or categorical data. You would usually use the "one hot encoder" representation (see "One Hot Encoder" on page 214) to convert the categorical data to numeric values. In practice, however, using *K*-means and PCA with binary data can be difficult.

If the standard *z*-scores are used, the binary variables will dominate the definition of the clusters. This is because 0/1 variables take on only two values and *K*-means can obtain a small within-cluster sum-of-squares by assigning all the records with a 0 or 1 to a single cluster. For example, apply `kmeans` to loan default data including factor variables `home` and `pub_rec_zero`:

```
df <- model.matrix(~ -1 + dti + payment_inc_ratio + home + pub_rec_zero,
                   data=defaults)
df0 <- scale(df)
km0 <- kmeans(df0, centers=4, nstart=10)
centers0 <-scale(km0$centers, center=FALSE,
                 scale=1/attr(df0, 'scaled:scale'))
scale(centers0, center=-attr(df0, 'scaled:center'), scale=F)
    dti payment_inc_ratio homeMORTGAGE homeOWN homeRENT pub_rec_zero
1 17.02              9.10         0.00       0     1.00         1.00
2 17.47              8.43         1.00       0     0.00         1.00
3 17.23              9.28         0.00       1     0.00         0.92
4 16.50              8.09         0.52       0     0.48         0.00
```

The top four clusters are essentially proxies for the different levels of the factor variables. To avoid this behavior, you could scale the binary variables to have a smaller variance than other variables. Alternatively, for very large data sets, you could apply clustering to different subsets of data taking on specific categorical values. For example, you could apply clustering separately to those loans made to someone who has a mortgage, owns a home outright, or rents.

Key Ideas for Scaling Data

- Variables measured on different scales need to be transformed to similar scales, so that their impact on algorithms is not determined mainly by their scale.
- A common scaling method is normalization (standardization)—subtracting the mean and dividing by the standard deviation.
- Another method is Gower's distance, which scales all variables to the 0–1 range (it is often used with mixed numeric and categorical data).

Summary

For dimension reduction of numeric data, the main tools are either principal components analysis or *K*-means clustering. Both require attention to proper scaling of the data to ensure meaningful data reduction.

For clustering with highly structured data in which the clusters are well separated, all methods will likely produce a similar result. Each method offers its own advantage. *K*-means scales to very large data and is easily understood. Hierarchical clustering can be applied to mixed data types—numeric and categorical—and lends itself to an intuitive display (the dendrogram). Model-based clustering is founded on statistical theory and provides a more rigorous approach, as opposed to the heuristic methods. For very large data, however, *K*-means is the main method used.

With noisy data, such as the loan and stock data (and much of the data that a data scientist will face), the choice is more stark. *K*-means, hierarchical clustering, and especially model-based clustering all produce very different solutions. How should a data scientist proceed? Unfortunately, there is no simple rule of thumb to guide the choice. Ultimately, the method used will depend on the data size and the goal of the application.

Bibliography

[bokeh] Bokeh Development Team. "Bokeh: Python library for interactive visualization" (2014). *http://www.bokeh.pydata.org*.

[Deng-Wickham-2011] Deng, H. and Wickham, H. "Density estimation in R" (2011). *http://vita.had.co.nz/papers/density-estimation.pdf*.

[Wikipedia-2016] "Diving." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 10 Mar 2016. Web. 19 Mar 2016.

[Donoho-2015] Donoho, David. "50 Years of Data Science" (2015). *http://courses.csail.mit.edu/18.337/2015/docs/50YearsDataScience.pdf*.

[Duong-2001] Duang, Tarn. "An introduction to kernel density estimation" (2001). *http://www.mvstat.net/tduong/research/seminars/seminar-2001-05.pdf*.

[Few-2007] Few, Stephen. "Save the Pies for Dessert." Visual Intelligence Newsletter, Perceptual Edge (2007). *https://www.perceptualedge.com/articles/visual_business_intelligence/save_the_pies_for_dessert.pdf*.

[Hintze-Nelson-1998] Hintze, J. and Nelson, R. "Violin Plots: A Box Plot-Density Trace Synergism." *The American Statistician* 52.2 (May 1998): 181–184.

[Galton-1886] Galton, Francis. "Regression towards mediocrity in Hereditary stature." *The Journal of the Anthropological Institute of Great Britain and Ireland*, 15:246-273. JSTOR 2841583.

[ggplot2] Wickham, Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York (2009). ISBN: 978-0-387-98140-6. *http://had.co.nz/ggplot2/book*.

[Hyndman-Fan-1996] Hyndman, R. J. and Fan, Y. "Sample quantiles in statistical packages," *American Statistician* 50, (1996) 361–365.

[lattice] Sarkar, Deepayan. *Lattice: Multivariate Data Visualization with R*. Springer (2008). ISBN 978-0-387-75968-5. *http://lmdvr.r-forge.r-project.org*.

[Legendre] Legendre, Adrien-Marie. *Nouvelle methodes pour la determination des orbites des cometes*. F. Didot, Paris (1805).

[NIST-Handbook-2012] *NIST/SEMATECH e-Handbook of Statistical Methods*, *http://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm* (2012).

[R-base-2015] R Core Team. "R: A Language and Environment for Statistical Computing," R Foundation for Statistical Computing (2015). *http://www.R-project.org/*.

[seaborne] Wasdom, Michael. "Seaborn: statistical data visualization" (2015). *http://stanford.edu/~mwaskom/software/seaborn/#*.

[Stigler-Gauss] Stigler, Stephen M. "Gauss and the Invention of Least Squares." *Ann. Stat.* 9(3), 465–474 (1981).

[Trellis-Graphics] Becker, R., Cleveland, W, Shyu, M. and Kaluzny, S. "A Tour of Trellis Graphics" (1996). *http://polisci.msu.edu/jacoby/icpsr/graphics/manuscripts/Trellis_tour.pdf*.

[Tukey-1962] Tukey, John W. "The Future of Data Analysis." *Ann. Math. Statist.* 33 (1962), no. 1, 1–67. *https://projecteuclid.org/download/pdf_1/euclid.aoms/1177704711*

[Tukey-1977] Tukey, John W. *Exploratory Data Analysis*. Pearson (1977). ISBN: 978-0-201-07616-5.

[Tukey-1987] Tukey, John W. Edited by Jones, L. V. *The collected works of John W. Tukey: Philosophy and Principles of Data Analysis 1965–1986*, Volume IV. Chapman and Hall/CRC (1987). ISBN: 978-0-534-05101-3.

[UCLA] "R Library: Contrast Coding Systems for Categorical Variables." UCLA: Statistical Consulting Group. *http://www.ats.ucla.edu/stat/r/library/contrast_coding.htm*. Accessed June 2016.

[Zhang-Wang-2007] Zhang, Qi and Wang, Wei. 19th International Conference on Scientific and Statistical Database Management, IEEE Computer Society (2007).

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

About the Authors

Peter Bruce founded and grew the Institute for Statistics Education at Statistics.com, which now offers about 100 courses in statistics, roughly a third of which are aimed at the data scientist. In recruiting top authors as instructors and forging a marketing strategy to reach professional data scientists, Peter has developed both a broad view of the target market and his own expertise to reach it.

Andrew Bruce has over 30 years of experience in statistics and data science in academia, government, and business. He has a PhD in statistics from the University of Washington and has published numerous papers in refereed journals. He has developed statistical-based solutions to a wide range of problems faced by a variety of industries, from established financial firms to internet startups, and offers a deep understanding of the practice of data science.

Colophon

The animal on the cover of *Practical Statistics for Data Scientists* is a lined shore crab (*Pachygrapsus crassipes*), also known as a striped shore crab. It is found along the coasts and beaches of the Pacific Ocean in North America, Central America, Korea, and Japan. These crustaceans live under rocks, in tidepools, and within crevices. They spend about half their time on land, and periodically return to the water to wet their gills.

The lined shore crab is named for the green stripes on its brown-black carapace. It has red claws and purple legs, which also have a striped or mottled pattern. The crab generally grows to be 3–5 centimeters in size; females are slightly smaller. Their eyes are on flexible stalks that can rotate to give them a full field of vision as they walk.

Crabs are omnivores, feeding primarily on algae, but also mollusks, worms, fungi, dead animals, and other crustaceans (depending on what is available). They moult many times as they grow to adulthood, taking in water to expand and crack open their old shell. Once this is achieved, they spend several difficult hours getting free, and then must hide until the new shell hardens.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Pictorial Museum of Animated Nature*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.