CHAPTER 3 Algorithms

In the previous chapter we discussed in general how models are used in data science. In this chapter, we're going to be diving into algorithms.

An algorithm is a procedure or set of steps or rules to accomplish a task. Algorithms are one of the fundamental concepts in, or building blocks of, computer science: the basis of the design of elegant and efficient code, data preparation and processing, and software engineering.

Some of the basic types of tasks that algorithms can solve are sorting, searching, and graph-based computational problems. Although a given task such as sorting a list of objects could be handled by multiple possible algorithms, there is some notion of "best" as measured by efficiency and computational time, which matters especially when you're dealing with massive amounts of data and building consumer-facing products.

Efficient algorithms that work sequentially or in parallel are the basis of pipelines to process and prepare data. With respect to data science, there are at least three classes of algorithms one should be aware of:

1. Data munging, preparation, and processing algorithms, such as sorting, MapReduce, or Pregel.

   We would characterize these types of algorithms as data engineering, and while we devote a chapter to this, it's not the emphasis of this book. This is not to say that you won't be doing data wrangling and munging—just that we don't emphasize the algorithmic aspect of it.

2. Optimization algorithms for parameter estimation, including Stochastic Gradient Descent, Newton's Method, and Least Squares. We mention these types of algorithms throughout the book, and they underlie many R functions.
3. Machine learning algorithms are a large part of this book, and we discuss these more next.

Machine Learning Algorithms

Machine learning algorithms are largely used to predict, classify, or cluster.

Wait! Back in the previous chapter, didn't we already say modeling could be used to predict or classify? Yes. Here's where some lines have been drawn that can make things a bit confusing, and it's worth understanding who drew those lines.

Statistical *modeling* came out of statistics departments, and machine learning *algorithms* came out of computer science departments. Certain methods and techniques are considered to be part of both, and you'll see that we often use the words somewhat interchangeably.

You'll find some of the methods in this book, such as linear regression, in machine learning books as well as intro to statistics books. It's not necessarily useful to argue over who the rightful owner is of these methods, but it's worth pointing out here that it can get a little vague or ambiguous about what the actual difference is.

In general, machine learning algorithms that are the basis of artificial intelligence (AI) such as image recognition, speech recognition, recommendation systems, ranking and personalization of content—often the basis of data products—are not usually part of a core statistics curriculum or department. They aren't generally designed to infer the underlying *generative process* (e.g., to model something), but rather to predict or classify with the most accuracy.

These differences in methods reflect in cultural differences in the approaches of machine learners and statisticians that Rachel observed at Google, and at industry conferences. Of course, data scientists can and should use both approaches.

There are some broad generalizations to consider:

Interpreting parameters

Statisticians think of the parameters in their linear regression models as having real-world interpretations, and typically want to be able to find meaning in behavior or describe the real-world phenomenon corresponding to those parameters. Whereas a software engineer or computer scientist might be wanting to build their linear regression algorithm into production-level code, and the predictive model is what is known as a *black box algorithm*, they don't generally focus on the interpretation of the parameters. If they do, it is with the goal of handtuning them in order to optimize *predictive power*.

Confidence intervals

Statisticians provide confidence intervals and posterior distributions for parameters and estimators, and are interested in capturing the variability or uncertainty of the parameters. Many machine learning algorithms, such as k-means or k-nearest neighbors (which we cover a bit later in this chapter), don't have a notion of confidence intervals or uncertainty.

The role of explicit assumptions

Statistical models make explicit assumptions about data-generating processes and distributions, and you use the data to estimate parameters. Nonparametric solutions, like we'll see later in this chapter, don't make any assumptions about probability distributions, or they are implicit.

We say the following lovingly and with respect: statisticians have chosen to spend their lives investigating uncertainty, and they're never 100% confident about anything. Software engineers like to build things. They want to build models that predict the best they can, but there are no concerns about uncertainty—just build it! At companies like Facebook or Google, the philosophy is to build and iterate often. If something breaks, it can be fixed. A data scientist who somehow manages to find a balance between the statistical and computer science approaches, and to find value in both these ways of being, can thrive. Data scientists are the multicultural statistician-computer scientist hybrid, so we're not tied to any one way of thinking over another; they both have value. We'll sum up our take on this with guest speaker Josh Wills' (Chapter 13) well-tweeted quote:

Data scientist (noun): Person who is better at statistics than any software engineer and better at software engineering than any statistician.

— Josh Wills

Three Basic Algorithms

Many business or real-world problems that can be solved with data can be thought of as *classification* and *prediction* problems when we express them mathematically. Happily, a whole host of models and algorithms can be used to classify and predict.

Your real challenge as a data scientist, once you've become familiar with how to implement them, is understanding which ones to use depending on the context of the problem and the underlying assumptions. This partially comes with experience—you start seeing enough problems that you start thinking, "Ah, this is a classification problem with a binary outcome" or, "This is a classification problem, but oddly I don't even have any labels" and you know what to do. (In the first case, you could use logistic regression or Naive Bayes, and in the second you could start with k-means—more on all these shortly!)

Initially, though, when you hear about these methods in isolation, it takes some effort on your part as a student or learner to think, "In the real world, how do I know that this algorithm is the solution to the problem I'm trying to solve?"

It's a real mistake to be the type of person walking around with a hammer looking for a nail to bang: "I know linear regression, so I'm going to try to solve every problem I encounter with it." Don't do that. Instead, try to understand the context of the problem, and the attributes it has *as a problem*. Think of those in mathematical terms, and then think about the algorithms you know and how they map to this type of problem.

If you're not sure, it's good to talk it through with someone who does. So ask a coworker, head to a meetup group, or start one in your area! Also, maintain the attitude that it's *not obvious* what to do and that's what makes it a problem, and so you're going to approach it circumspectly and methodically. You don't have to be the know-it-all in the room who says, "Well, *obviously* we should use linear regression with a penalty function for regularization," even if that seems to you the right approach.

We're saying all this because one of the unfortunate aspects of textbooks is they often give you a bunch of techniques and then problems that tell you *which* method to use that solves the problem (e.g., use linear regression to predict height from weight). Yes, implementing and understanding linear regression the first few times is not obvious, so you need practice with that, but it needs to be addressed that the real challenge once you have mastery over technique is *knowing when to use linear regression in the first place*.

We're not going to give a comprehensive overview of *all* possible machine learning algorithms, because that would make this a machine learning book, and there are already plenty of those.

Having said that, in this chapter we'll introduce three basic algorithms now and introduce others throughout the book in context. By the end of the book, you should feel more confident about your ability to learn new algorithms so that you can pick them up along the way as problems require them.

We'll also do our best to demonstrate the thought processes of data scientists who had to figure out which algorithm to use in context and why, but it's also upon you as a student and learner to *force yourself* to think about what the attributes of the problem were that made a given algorithm the right algorithm to use.

With that said, we still need to give you some basic tools to use, so we'll start with linear regression, k-nearest neighbors (k-NN), and k-means. In addition to what was just said about trying to understand the attributes of problems that could use these as solutions, look at these three algorithms from the perspective of: what patterns can we as humans see in the data with our eyes that we'd like to be able to automate with a machine, especially taking into account that as the data gets more complex, we can't see these patterns?

Linear Regression

One of the most common statistical methods is linear regression. At its most basic, it's used when you want to express the mathematical relationship between two variables or attributes. When you use it, you are making the assumption that there is a *linear* relationship between an outcome variable (sometimes also called the response variable, dependent variable, or label) and a predictor (sometimes also called an independent variable, explanatory variable, or feature); or between one variable and several other variables, in which case you're *modeling* the relationship as having a linear structure.

WTF. So Is It an Algorithm or a Model?

While we tried to make a distinction between the two earlier, we admit the colloquial use of the words "model" and "algorithm" gets confusing because the two words seem to be used interchangeably when their actual definitions are not the same thing at all. In the purest sense, an algorithm is a set of rules or steps to follow to accomplish some task, and a model is an attempt to describe or capture the world. These two seem obviously different, so it seems the distinction should should be obvious. Unfortunately, it isn't. For example, regression can be described as a statistical model as well as a machine learning algorithm. You'll waste your time trying to get people to discuss this with any precision.

In some ways this is a historical artifact of statistics and computer science communities developing methods and techniques in parallel and using different words for the same methods. The consequence of this is that the distinction between machine learning and statistical modeling is muddy. Some methods (for example, k-means, discussed in the next section) we might call an *algorithm* because it's a series of computational steps used to cluster or classify objects—on the other hand, k-means can be reinterpreted as a special case of a Gaussian mixture *model*. The net result is that colloquially, people use the terms algorithm and model interchangeably when it comes to a lot of these methods, so try not to let it worry you. (Though it bothers us, too.)

Assuming that there is a *linear* relationship between an outcome variable and a predictor is a big assumption, but it's also the simplest one you *can* make—linear functions are more basic than nonlinear ones in a mathematical sense—so in that sense it's a good starting point.

In some cases, it makes sense that changes in one variable correlate linearly with changes in another variable. For example, it makes sense that the more umbrellas you sell, the more money you make. In those cases you can feel good about the linearity assumption. Other times, it's harder to justify the assumption of linearity except locally: in the spirit of calculus, everything can be approximated by line segments as long as functions are continuous.

Let's back up. Why would you even want to build a linear model in the first place? You might want to use this relationship to *predict* future outcomes, or you might want to understand or *describe* the relationship to get a grasp on the situation. Let's say you're studying the relationship between a company's sales and how much that company spends on advertising, or the number of friends someone has on a social networking site and the time that person spends on that site daily. These are all numerical outcomes, which mean linear regression would be a wise choice, at least for a first pass at your problem.

One entry point for thinking about linear regression is to think about deterministic lines first. We learned back in grade school that we could describe a line with a slope and an intercept, $y = f(x) = \beta_0 + \beta_1 * x$. But the setting there was always deterministic.

Even for the most mathematically sophisticated among us, if you haven't done it before, it's a new mindset to start thinking about stochastic functions. We still have the same components: points listed out explicitly in a table (or as tuples), and functions represented in equation form or plotted on a graph. So let's build up to linear regression starting from a deterministic function.

Example 1. Overly simplistic example to start. Suppose you run a social networking site that charges a monthly subscription fee of $25, and that this is your only source of revenue. Each month you collect data and count your number of users and total revenue. You've done this daily over the course of two years, recording it all in a spreadsheet. You could express this data as a series of points. Here are the first four:

$$S = \{(x, y) = (1, 25), (10, 250), (100, 2500), (200, 5000)\}$$

If you showed this to someone else who didn't even know how much you charged or anything about your business model (what kind of friend wasn't paying attention to your business model?!), they might notice that there's a clear relationship enjoyed by all of these points, namely $y = 25x$. They likely could do this in their head, in which case they figured out that:

- There's a linear pattern.
- The coefficient relating x and y is 25.
- It seems deterministic.

You can even plot it as in Figure 3-1 to verify they were right (even though you knew they were because you made the business model in the first place). It's a line!

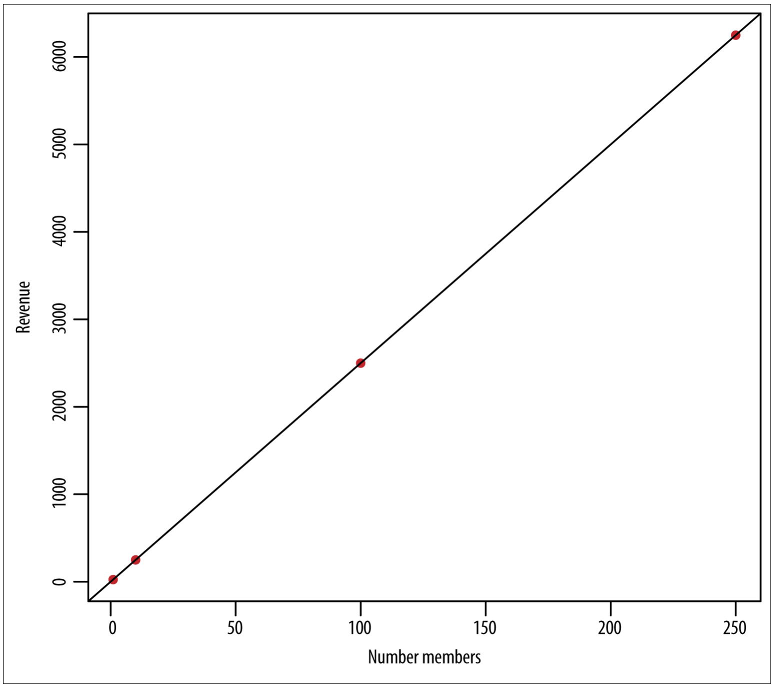

Figure 3-1. An obvious linear pattern

Example 2. Looking at data at the user level. Say you have a dataset *keyed* by user (meaning each row contains data for a single user), and the columns represent user behavior on a social networking site over a period of a week. Let's say you feel comfortable that the data is clean at this stage and that you have on the order of hundreds of thousands of users. The names of the columns are total_num_friends, total_new_friends_this_week, num_visits, time_spent, number_apps_downloaded, number_ads_shown, gender, age, and so on. During the course of your exploratory data analysis, you've randomly sampled 100 users to keep it simple, and you plot pairs of these variables, for example, x = total_new_friends and y = time_spent (in seconds). The business context might be that eventually you want to be able to promise advertisers who bid for space on your website in advance a certain number of users, so you want to be able to forecast number of users several days or weeks in advance. But for now, you are simply trying to build intuition and understand your dataset.

You eyeball the first few rows and see:

7	276
3	43
4	82
6	136
10	417
9	269

Now, your brain can't figure out what's going on by just looking at them (and your friend's brain probably can't, either). They're in no obvious particular order, and there are a lot of them. So you try to plot it as in Figure 3-2.

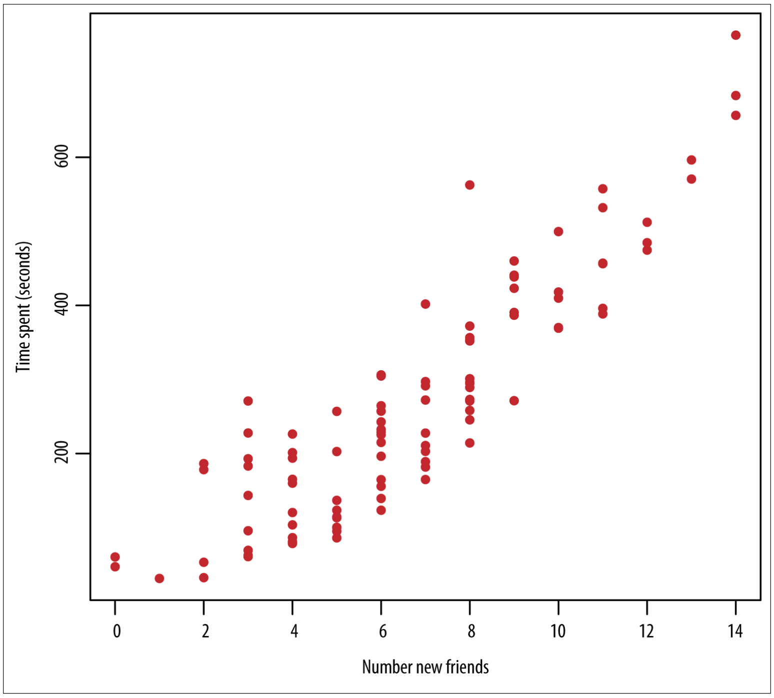

Figure 3-2. Looking kind of linear

It looks like there's *kind of* a linear relationship here, and it makes sense; the more new friends you have, the more time you might spend on the site. But how can you figure out how to describe that relationship? Let's also point out that there is no perfectly *deterministic* relationship between number of new friends and time spent on the site, but it makes sense that there is an *association* between these two variables.

Start by writing something down

There are two things you want to capture in the model. The first is the *trend* and the second is the *variation*. We'll start first with the trend.

First, let's start by assuming there actually *is* a relationship and that it's linear. It's the best you can do at this point.

There are many lines that look more or less like they might work, as shown in Figure 3-3.

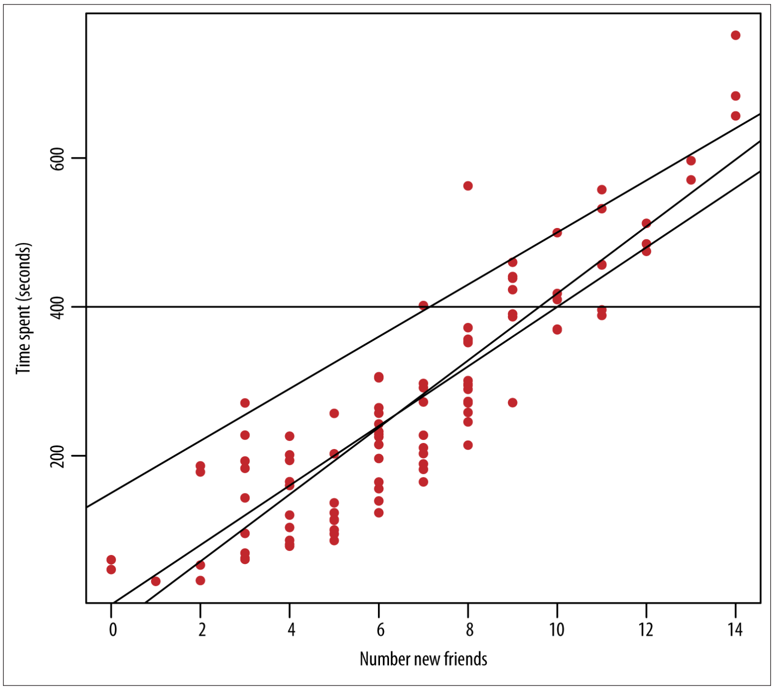

Figure 3-3. Which line is the best fit?

So how do you pick which one?

Because you're assuming a linear relationship, start your model by assuming the functional form to be:

$$y = \beta_0 + \beta_1 x$$

Now your job is to find the best choices for β_0 and β_1 using the observed data to estimate them: $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$.

Writing this with matrix notation results in this:

$$y = x \cdot \beta$$

There you go: you've written down your model. Now the rest is *fitting* the model.

Fitting the model

So, how do you calculate β? The intuition behind linear regression is that you want to find the line that minimizes the distance between all the points and the line.

Many lines look approximately correct, but your goal is to find the optimal one. Optimal could mean different things, but let's start with optimal to mean the line that, on average, is closest to all the points. But what does *closest* mean here?

Look at Figure 3-4. Linear regression seeks to find the line that minimizes the sum of the squares of the vertical distances between the approximated or predicted $\hat{y}_i$s and the observed y_is. You do this because you want to minimize your prediction errors. This method is called *least squares* estimation.

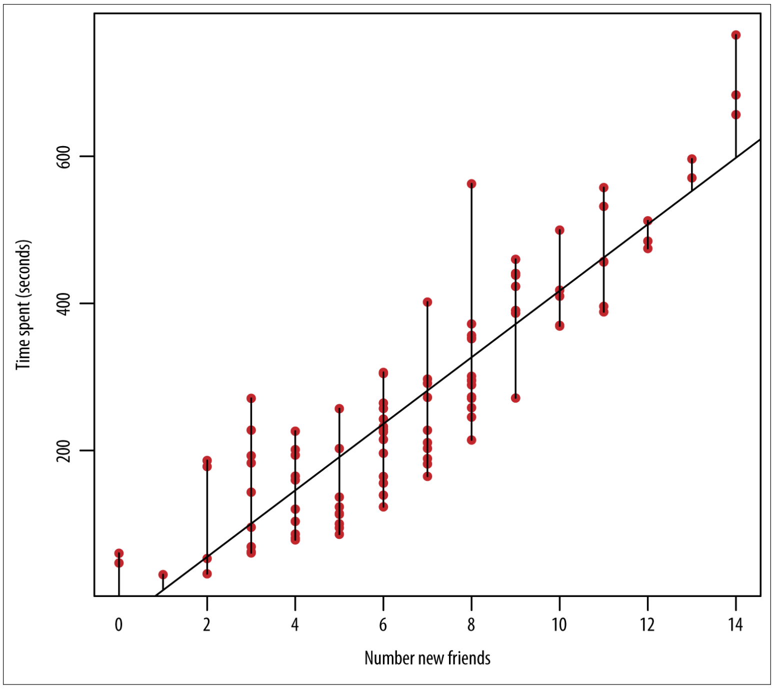

Figure 3-4. The line closest to all the points

To find this line, you'll define the "residual sum of squares" (RSS), denoted $RSS(\beta)$, to be:

$$RSS(\beta) = \Sigma_i \left(y_i - \beta x_i\right)^2$$

where i ranges over the various data points. It is the sum of all the squared vertical distances between the observed points and any given line. Note this is a function of β and you want to optimize with respect to β to find the optimal line.

To minimize $RSS(\beta) = (y - \beta x)^t (y - \beta x)$, differentiate it with respect to β and set it equal to zero, then solve for β. This results in:

$$\hat{\beta} = \left(x^t x\right)^{-1} x^t y$$

Here the little "hat" symbol on top of the β is there to indicate that it's the *estimator* for β. You don't know the true value of β; all you have is the observed data, which you plug into the estimator to get an estimate.

To actually fit this, to get the βs, all you need is one line of R code where you've got a column of y's and a (single) column of x's:

```
model <- lm(y ~ x)
```

So for the example where the first few rows of the data were:

x	y
7	276
3	43
4	82
6	136
10	417
9	269

The R code for this would be:

```
> model <- lm (y~x)
> model

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     -32.08        45.92

> coefs <- coef(model)
> plot(x, y, pch=20,col="red", xlab="Number new friends",
  ylab="Time spent (seconds)")
> abline(coefs[1],coefs[2])
```

And the estimated line is $\hat{y} = -32.08 + 45.92x$, which you're welcome to round to $\hat{y} = -32 + 46x$, and the corresponding plot looks like the lefthand side of Figure 3-5.

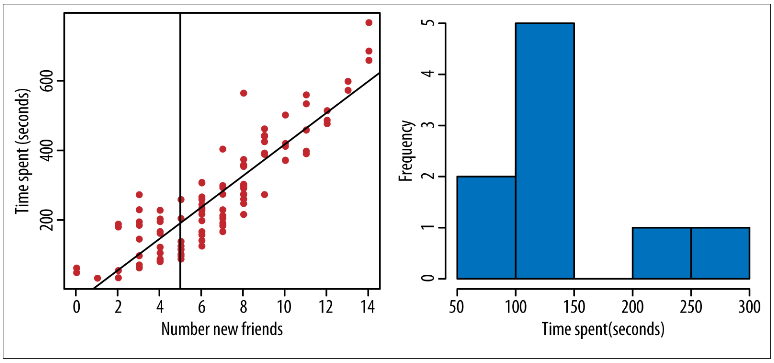

Figure 3-5. On the left is the fitted line. We can see that for any fixed value, say 5, the values for y vary. For people with 5 new friends, we display their time spent in the plot on the right.

But it's up to you, the data scientist, whether you think you'd actually want to use this linear model to describe the relationship or predict new outcomes. If a new x-value of 5 came in, meaning the user had five new friends, how confident are you in the output value of $-32.08 + 45.92*5 = 195.7$ seconds?

In order to get at this question of confidence, you need to extend your model. You know there's variation among time spent on the site by people with five new friends, meaning you certainly wouldn't make the claim that everyone with five new friends is guaranteed to spend 195.7 seconds on the site. So while you've so far modeled the *trend*, you haven't yet modeled the *variation*.

Extending beyond least squares

Now that you have a *simple linear regression model* down (one output, one predictor) using least squares estimation to estimate your βs, you can build upon that model in three primary ways, described in the upcoming sections:

1. Adding in modeling assumptions about the errors
2. Adding in more predictors
3. Transforming the predictors

Adding in modeling assumptions about the errors. If you use your model to predict y for a given value of x, your prediction is deterministic and doesn't capture the variablility in the observed data. See on the righthand side of Figure 3-5 that for a fixed value of $x = 5$, there is variability among the time spent on the site. You want to capture this variability in your model, so you extend your model to:

$$y = \beta_0 + \beta_1 x + \epsilon$$

where the new term ϵ is referred to as *noise*, which is the stuff that you haven't accounted for by the relationships you've figured out so far. It's also called the *error term*—ϵ represents the *actual error*, the difference between the observations and the *true* regression line, which you'll never know and can only estimate with your $\hat{\beta}$s.

One often makes the modeling assumption that the noise is normally distributed, which is denoted:

$$\epsilon \sim N\left(0, \sigma^2\right)$$

Note this is sometimes not a reasonable assumption. If you are dealing with a known fat-tailed distribution, and if your linear model is picking up only a small part of the value of the variable y, then the error terms are likely also fat-tailed. This is the most common situation in financial modeling.

That's not to say we don't use linear regression in finance, though. We just don't attach the "noise is normal" assumption to it.

With the preceding assumption on the distribution of noise, this model is saying that, for any given value of x, the conditional distribution of y given x is $p(y|x) \sim N\left(\beta_0 + \beta_1 x, \sigma^2\right)$.

So, for example, among the set of people who had five new friends this week, the amount of the time they spent on the website had a *normal distribution* with a mean of $\beta_0 + \beta_1 * 5$ and a variance of σ^2, and you're going to estimate your parameters β_0, β_1, σ from the data.

How do you fit this model? How do you get the parameters β_0, β_1, σ from the data?

Turns out that no matter how the ϵs are distributed, the least squares estimates that you already derived are the optimal estimators for βs because they have the property of being unbiased and of being the minimum variance estimators. If you want to know more about these properties and see a proof for this, we refer you to any good book on statistical inference (for example, *Statistical Inference* by Casella and Berger).

So what can you do with your observed data to estimate the variance of the errors? Now that you have the estimated line, you can see how far away the observed data points are from the line itself, and you can treat these differences, also known as *observed errors* or *residuals* ,as observations themselves, or estimates of the actual errors, the ϵs. Define $e_i = y_i - \hat{y}_i = y_i - \left(\hat{\beta}_0 + \hat{\beta}_1 x_i\right)$ for $i = 1, \ldots, n$.

Then you estimate the variance (σ^2) of ϵ, as:

$$\frac{\sum_i e_i^2}{n-2}$$

Why are we dividing by n–2? A natural question. Dividing by n–2, rather than just n, produces an *unbiased estimator*. The 2 corresponds to the number of model parameters. Here again, Casella and Berger's book is an excellent resource for more background information.

This is called the *mean squared error* and captures how much the predicted value varies from the observed. *Mean squared error* is a useful quantity for any prediction problem. In regression in particular, it's *also* an estimator for your variance, but it can't always be used or interpreted that way. It appears in the evaluation metrics in the following section.

Evaluation metrics

We asked earlier how confident you would be in these estimates and in your model. You have a couple values in the output of the R function that help you get at the issue of how confident you can be in the estimates: p-values and R-squared. Going back to our model in R, if we type in **`summary(model)`**, which is the name we gave to this model, the output would be:

```
summary (model)
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-121.17  -52.63   -9.72   41.54  356.27

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -32.083     16.623   -1.93   0.0565 .
x             45.918      2.141   21.45   <2e-16 ***
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 77.47 on 98 degrees of freedom
Multiple R-squared: 0.8244,    Adjusted R-squared: 0.8226
F-statistic:   460 on 1 and 98 DF,  p-value: < 2.2e-16
```

R-squared

$R^2 = 1 - \frac{\sum_i \left(y_i - \hat{y}_i\right)^2}{\sum_i \left(y_i - \bar{y}\right)^2}$. This can be interpreted as the proportion of variance explained by our model. Note that mean squared error is in there getting divided by total error, which is the proportion of variance *unexplained* by our model and we calculate 1 minus that.

p-values

Looking at the output, the estimated βs are in the column marked Estimate. To see the p-values, look at $Pr(>|t|)$. We can interpret the values in this column as follows: We are making a null hypothesis that the βs are zero. For any given β, the p-value captures the probability of observing the data that we observed, and obtaining the test-statistic that we obtained *under the null hypothesis*. This means that if we have a low p-value, it is highly unlikely to observe such a test-statistic under the null hypothesis, and the coefficient is highly likely to be nonzero and therefore significant.

Cross-validation

Another approach to evaluating the model is as follows. Divide our data up into a training set and a test set: 80% in the training and 20% in the test. Fit the model on the training set, then look at the *mean squared error* on the test set and compare it to that on the training set. Make this comparison across sample size as well. If the mean squared errors are approximately the same, then our model generalizes well and we're not in danger of overfitting. See Figure 3-6 to see what this might look like. This approach is highly recommended.

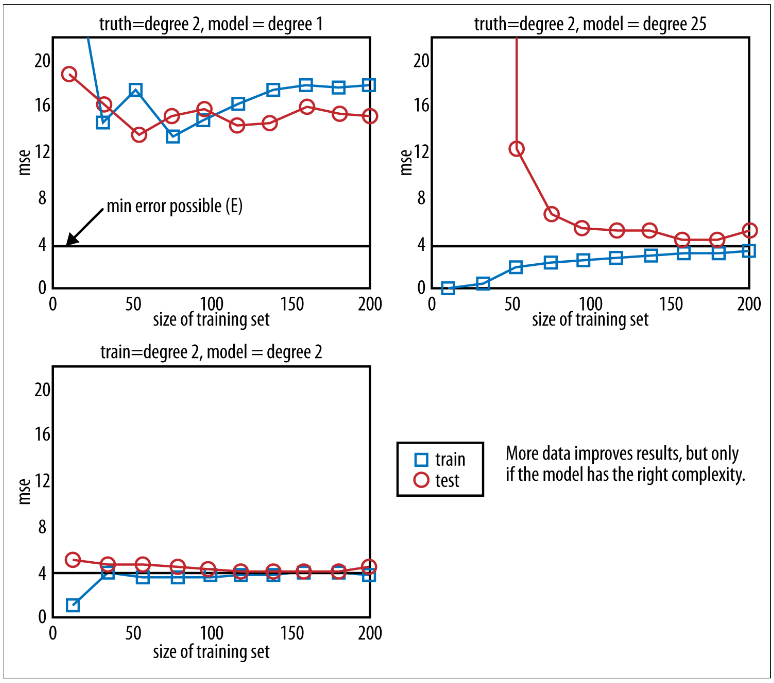

Figure 3-6. Comparing mean squared error in training and test set, taken from a slide of Professor Nando de Freitas; here, the ground truth is known because it came from a dataset with data simulated from a known distribution

Other models for error terms

The mean squared error is an example of what is called a *loss function*. This is the standard one to use in linear regression because it gives us a pretty nice measure of closeness of fit. It has the additional desirable property that by assuming that εs are normally distributed, we can rely on the maximum likelihood principle. There are other loss functions such as one that relies on absolute value rather than squaring. It's also possible to build custom loss functions specific to your particular problem or context, but for now, you're safe with using mean square error.

Adding other predictors. What we just looked at was simple linear regression—one outcome or dependent variable and one predictor. But we can extend this model by building in other predictors, which is called *multiple linear regression*:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon.$$

All the math that we did before holds because we had expressed it in matrix notation, so it was already generalized to give the appropriate estimators for the β. In the example we gave of predicting time spent on the website, the other predictors could be the user's age and gender, for example. We'll explore feature selection more in Chapter 7, which means figuring out which additional predictors you'd want to put in your model. The R code will just be:

```
model <- lm(y ~ x_1 + x_2 + x_3)
```

Or to add in interactions between variables:

```
model <- lm(y ~ x_1 + x_2 + x_3 + x_2 * x_3)
```

One key here is to make scatterplots of y against each of the predictors as well as between the predictors, and histograms of $y|x$ for various values of each of the predictors to help build intuition. As with simple linear regression, you can use the same methods to evaluate your model as described earlier: looking at R^2, p-values, and using training and testing sets.

Transformations. Going back to one x predicting one y, why did we assume a linear relationship? Instead, maybe, a better model would be a polynomial relationship like this:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

Wait, but isn't this *linear* regression? Last time we checked, polynomials weren't linear. To think of it as *linear*, you transform or create new variables—for example, $z = x^2$—and build a regression model based on z. Other common transformations are to take the log or to pick a threshold and turn it into a binary predictor instead.

If you look at the plot of time spent versus number friends, the shape looks a little bit curvy. You could potentially explore this further by building up a model and checking to see whether this yields an improvement.

What you're facing here, though, is one of the biggest challenges for a modeler: you never know the truth. It's possible that the true model is quadratic, but you're assuming linearity or vice versa. You do your best to evaluate the model as discussed earlier, but you'll never *really* know if you're right. More and more data can sometimes help in this regard as well.

Review

Let's review the assumptions we made when we built and fit our model:

- Linearity
- Error terms normally distributed with mean 0
- Error terms independent of each other
- Error terms have constant variance across values of x
- The predictors we're using are the *right* predictors

When and why do we perform linear regression? Mostly for two reasons:

- If we want to predict one variable knowing others
- If we want to explain or understand the relationship between two or more things

Exercise

To help understand and explore new concepts, you can simulate fake datasets in R. The advantage of this is that you "play God" because you actually know the underlying truth, and you get to see how good your model is at recovering the truth.

Once you've better understood what's going on with your fake dataset, you can then transfer your understanding to a real one. We'll show you how to simulate a fake dataset here, then we'll give you some ideas for how to explore it further:

```
# Simulating fake data
x_1 <- rnorm(1000,5,7) # from a normal distribution simulate
                       # 1000 values with a mean of 5 and
                       #  standard deviation of 7
hist(x_1, col="grey") # plot p(x)
true_error <- rnorm(1000,0,2)
true_beta_0 <- 1.1
```

```
true_beta_1 <- -8.2

y <- true_beta_0 + true_beta_1*x_1 + true_error
hist(y) # plot p(y)
plot(x_1,y, pch=20,col="red") # plot p(x,y)
```

1. Build a regression model and see that it recovers the true values of the βs.
2. Simulate another fake variable x_2 that has a Gamma distribution with parameters you pick. Now make the truth be that y is a linear combination of both x_1 and x_2. Fit a model that only depends on x_1. Fit a model that only depends on x_2. Fit a model that uses both. Vary the sample size and make a plot of mean square error of the training set and of the test set versus sample size.
3. Create a new variable, z, that is equal to ${x_1}^2$. Include this as one of the predictors in your model. See what happens when you fit a model that depends on x_1 only and then also on z. Vary the sample size and make a plot of mean square error of the training set and of the test set versus sample size.
4. Play around more by (a) changing parameter values (the true βs), (b) changing the distribution of the true error, and (c) including more predictors in the model with other kinds of probability distributions. (`rnorm()` means randomly generate values from a normal distribution. `rbinom()` does the same for binomial. So look up these functions online and try to find more.)
5. Create scatterplots of all pairs of variables and histograms of single variables.

k-Nearest Neighbors (k-NN)

K-NN is an algorithm that can be used when you have a bunch of objects that have been classified or labeled in some way, and other similar objects that haven't gotten classified or labeled yet, and you want a way to automatically label them.

The objects could be data scientists who have been classified as "sexy" or "not sexy"; or people who have been labeled as "high credit" or "low credit"; or restaurants that have been labeled "five star," "four star," "three star," "two star," "one star," or if they really suck, "zero stars." More seriously, it could be patients who have been classified as "high cancer risk" or "low cancer risk."

Take a second and think whether or not linear regression would work to solve problems of this type.

OK, so the answer is: it depends. When you use linear regression, the output is a continuous variable. Here the output of your algorithm is going to be a categorical label, so linear regression wouldn't solve the problem as it's described.

However, it's not impossible to solve it with linear regression plus the concept of a "threshold." For example, if you're trying to predict people's credit scores from their ages and incomes, and then picked a threshold such as 700 such that if your prediction for a given person whose age and income you observed was above 700, you'd label their predicted credit as "high," or toss them into a bin labeled "high." Otherwise, you'd throw them into the bin labeled "low." With more thresholds, you could also have more fine-grained categories like "very low," "low," "medium," "high," and "very high."

In order to do it this way, with linear regression you'd have to establish the bins as ranges of a continuous outcome. But not everything is on a continuous scale like a credit score. For example, what if your labels are "likely Democrat," "likely Republican," and "likely independent"? What do you do now?

The intution behind k-NN is to consider the *most similar* other items defined in terms of their attributes, look at their labels, and give the unassigned item the majority vote. If there's a tie, you randomly select among the labels that have tied for first.

So, for example, if you had a bunch of movies that were labeled "thumbs up" or "thumbs down," and you had a movie called "Data Gone Wild" that hadn't been rated yet—you could look at its attributes: length of movie, genre, number of sex scenes, number of Oscar-winning actors in it, and budget. You could then find other movies with similar attributes, look at *their* ratings, and then give "Data Gone Wild" a rating without ever having to watch it.

To automate it, two decisions must be made: first, how do you define *similarity* or closeness? Once you define it, for a given unrated item, you can say how similar *all* the labeled items are to it, and you can take the *most similar* items and call them *neighbors*, who each have a "vote."

This brings you to the second decision: how many neighbors should you look at or "let vote"? This value is *k*, which ultimately you'll choose as the data scientist, and we'll tell you how.

Make sense? Let's try it out with a more realistic example.

Example with credit scores

Say you have the age, income, and a credit category of high or low for a bunch of people and you want to use the age and income to predict the credit label of "high" or "low" for a new person.

For example, here are the first few rows of a dataset, with income represented in thousands:

```
age income credit
69       3     low
66      57     low
49      79     low
49      17     low
58      26    high
44      71    high
```

You can plot people as points on the plane and label people with an empty circle if they have low credit ratings, as shown in Figure 3-7.

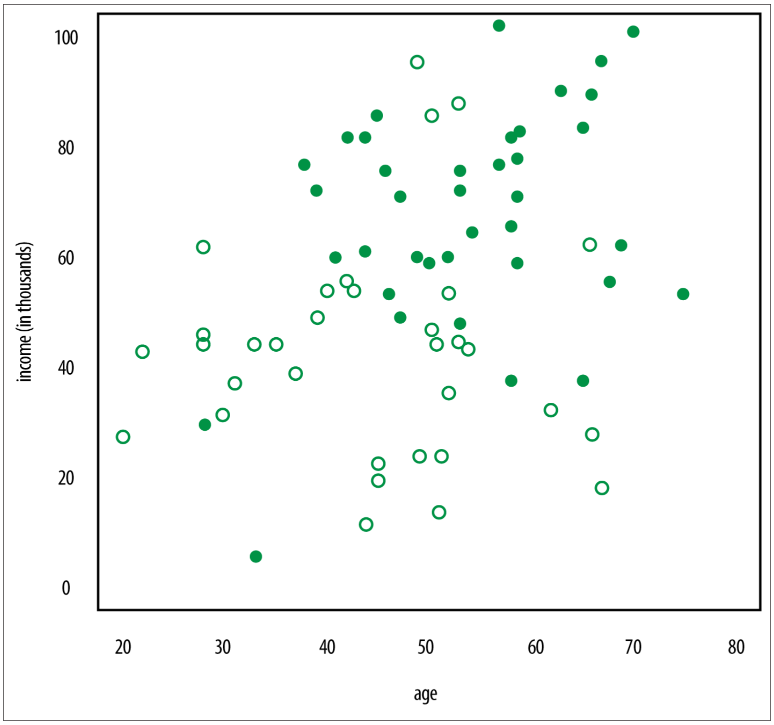

Figure 3-7. Credit rating as a function of age and income

What if a new guy comes in who is 57 years old and who makes $37,000? What's his likely credit rating label? Look at Figure 3-8. Based on the other people near him, what credit score label do you think he should be given? Let's use k-NN to do it automatically.

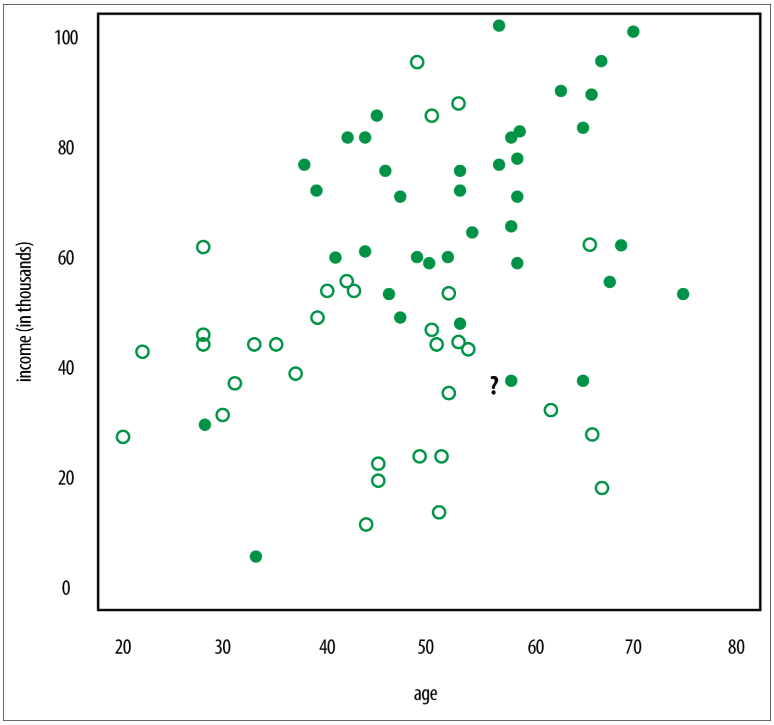

Figure 3-8. What about that guy?

Here's an overview of the process:

1. Decide on your *similarity* or *distance* metric.
2. Split the original labeled dataset into training and test data.
3. Pick an evaluation metric. (Misclassification rate is a good one. We'll explain this more in a bit.)
4. Run k-NN a few times, changing *k* and checking the evaluation measure.
5. Optimize *k* by picking the one with the best evaluation measure.
6. Once you've chosen *k*, use the same training set and now create a new test set with the people's ages and incomes that you have *no*

labels for, and want to predict. In this case, your new test set only has one lonely row, for the 57-year-old.

Similarity or distance metrics

Definitions of "closeness" and similarity vary depending on the context: closeness in social networks could be defined as the number of overlapping friends, for example.

For the sake of our problem of what a neighbor is, we can use Euclidean distance on the plane if the variables are on the same scale. And that can sometimes be a big IF.

Caution: Modeling Danger Ahead!

The scalings question is a really big deal, and if you do it wrong, your model could just suck.

Let's consider an example: Say you measure age in years, income in dollars, and credit rating as credit scores normally are given—something like SAT scores. Then two people would be represented by triplets such as $(25, 54000, 700)$ and $(35, 76000, 730)$. In particular, their "distance" would be completely dominated by the difference in their salaries.

On the other hand, if you instead measured salary in *thousands of dollars*, they'd be represented by the triplets $(25, 54, 700)$ and $(35, 76, 730)$, which would give all three variables similar kinds of influence.

Ultimately the way you scale your variables, or equivalently in this situation the way you define your concept of distance, has a potentially enormous effect on the output. In statistics it is called your "prior."

Euclidean distance is a good go-to distance metric for attributes that are real-valued and can be plotted on a plane or in multidimensional space. Some others are:

Cosine Similarity

Also can be used between two real-valued vectors, $\vec{x}$ and $\vec{y}$, and will yield a value between –1 (exact opposite) and 1 (exactly the same) with 0 in between meaning independent. Recall the definition $\cos\left(\vec{x},\vec{y}\right)=\frac{\vec{x}\cdot\vec{y}}{\|\vec{x}\|\|\vec{y}\|}$.

Jaccard Distance or Similarity

This gives the distance between a set of objects—for example, a list of Cathy's friends $A=\{Kahn, Mark, Laura,\ldots\}$ and a list of Rachel's friends $B=\{Mladen, Kahn, Mark,\ldots\}$—and says how similar those two sets are: $J(A,B)=\frac{|A\cap B|}{|A\cup B|}$.

Mahalanobis Distance

Also can be used between two real-valued vectors and has the advantage over Euclidean distance that it takes into account correlation and is scale-invariant. $d\left(\vec{x},\vec{y}\right)=\sqrt{\left(\vec{x}-\vec{y}\right)^T S^{-1}\left(\vec{x}-\vec{y}\right)}$, where S is the covariance matrix.

Hamming Distance

Can be used to find the distance between two strings or pairs of words or DNA sequences of the same length. The distance between olive and ocean is 4 because aside from the "o" the other 4 letters are different. The distance between shoe and hose is 3 because aside from the "e" the other 3 letters are different. You just go through each position and check whether the letters the same in that position, and if not, increment your count by 1.

Manhattan

This is also a distance between two real-valued k-dimensional vectors. The image to have in mind is that of a taxi having to travel the city streets of Manhattan, which is laid out in a grid-like fashion (you can't cut diagonally across buildings). The distance is therefore defined as $d\left(\vec{x},\vec{y}\right)=\Sigma_i^k\left|x_i-y_i\right|$, where i is the ith element of each of the vectors.

There are many more distance metrics available to you depending on your type of data. We start with a Google search when we're not sure where to start.

What if your attributes are a mixture of kinds of data? This happens in the case of the movie ratings example: some were numerical attributes, such as budget and number of actors, and one was categorical, genre. But you can always define your own custom distance metric.

For example, you can say if movies are the same genre, that will contribute "0" to their distance. But if they're of a different genre, that will contribute "10," where you picked the value 10 based on the fact that this was on the same scale as budget (millions of dollars), which is in the range of 0 and 100. You could do the same with number of actors. You could play around with the 10; maybe 50 is better.

You'll want to justify why you're making these choices. The justification could be that you tried different values and when you tested the algorithm, this gave the best evaluation metric. Essentially this 10 is either a second tuning parameter that you've introduced into the algorithm on top of the *k*, or a prior you've put on the model, depending on your point of view and how it's used.

Training and test sets

For any machine learning algorithm, the general approach is to have a training phase, during which you create a model and "train it"; and then you have a testing phase, where you use new data to test how good the model is.

For k-NN, the training phase is straightforward: it's just reading in your data with the "high" or "low" credit data points marked. In testing, you pretend you don't know the true label and see how good you are at guessing using the k-NN algorithm.

To do this, you'll need to save some clean data from the overall data for the testing phase. Usually you want to save randomly selected data, let's say 20%.

Your R console might look like this:

```
> head(data)
  age income credit
1  69      3    low
2  66     57    low
3  49     79    low
4  49     17    low
5  58     26   high
6  44     71   high

n.points <- 1000 # number of rows in the dataset
sampling.rate <- 0.8

# we need the number of points in the test set to calculate
# the misclassification rate
num.test.set.labels <- n.points * (1 - sampling.rate)
```

```
# randomly sample which rows will go in the training set
training <- sample(1:n.points, sampling.rate * n.points,
                   replace=FALSE)
# define the training set to be those rows
train <- subset(data[training, ], select = c(Age, Income))

# the other rows are going into the test set
testing <- setdiff(1:n.points, training)
# define the test set to be the other rows
test <- subset(data[testing, ], select = c(Age, Income))

# this is the subset of labels for the training set
cl <- data$Credit[training]
# subset of labels for the test set, we're withholding these
true.labels <- data$Credit[testing]
```

Pick an evaluation metric

How do you evaluate whether your model did a good job?

This isn't easy or universal—you may decide you want to penalize certain kinds of misclassification more than others. False negatives may be way worse than false positives. Coming up with the evaluation metric could be something you work on with a domain expert.

For example, if you were using a classification algorithm to predict whether someone had cancer or not, you would want to minimize false negatives (misdiagnosing someone as not having cancer when they actually do), so you could work with a doctor to tune your evaluation metric.

Note you want to be careful because if you really wanted to have *no* false negatives, you could just tell *everyone* they have cancer. So it's a trade-off between *sensitivity* and *specificity*, where sensitivity is here defined as the probability of correctly diagnosing an ill patient as ill; specificity is here defined as the probability of correctly diagnosing a well patient as well.

Other Terms for Sensitivity and Specificity

Sensitivity is also called the *true positive rate* or *recall* and varies based on what academic field you come from, but they all mean the same thing. And *specificity* is also called the *true negative rate*. There is also the *false positive rate* and the *false negative rate*, and these don't get other special names.

Another evaluation metric you could use is *precision*, defined in Chapter 5. The fact that some of the same formulas have different names is due to the fact that different academic disciplines have developed these ideas separately. So *precision* and *recall* are the quantities used in the field of information retrieval. Note that *precision* is not the same thing as *specificity*.

Finally, we have *accuracy*, which is the ratio of the number of correct labels to the total number of labels, and the misclassification rate, which is just 1–*accuracy*. Minimizing the *misclassification rate* then just amounts to maximizing *accuracy*.

Putting it all together

Now that you have a distance measure and an evaluation metric, you're ready to roll.

For each person in your test set, you'll pretend you don't know his label. Look at the labels of his three nearest neighbors, say, and use the label of the majority vote to label him. You'll label all the members of the test set and then use the misclassification rate to see how well you did. All this is done automatically in R, with just this single line of R code:

```
knn (train, test, cl, k=3)
```

Choosing k

How do you choose *k*? This is a parameter you have control over. You might need to understand your data pretty well to get a good guess, and then you can try a few different *k*'s and see how your evaluation changes. So you'll run k-nn a few times, changing *k*, and checking the evaluation metric each time.

Binary Classes

When you have binary classes like "high credit" or "low credit," picking *k* to be an odd number can be a good idea because there will always be a majority vote, no ties. If there is a tie, the algorithm just randomly picks.

```
# we'll loop through and see what the misclassification rate
# is for different values of k
for (k in 1:20) {
   print(k)
```

```
    predicted.labels <- knn(train, test, cl, k)
    # We're using the R function knn()
    num.incorrect.labels <- sum(predicted.labels != true.labels)
    misclassification.rate <- num.incorrect.labels /
                              num.test.set.labels
    print(misclassification.rate)
}
```

Here's the output in the form (k, misclassification rate):

```
k  misclassification.rate
1, 0.28
2, 0.315
3, 0.26
4, 0.255
5, 0.23
6, 0.26
7, 0.25
8, 0.25
9, 0.235
10, 0.24
```

So let's go with $k = 5$ because it has the lowest misclassification rate, and now you can apply it to your guy who is 57 with a $37,000 salary. In the R console, it looks like:

```
> test <- c(57,37)
> knn(train,test,cl, k = 5)
[1] low
```

The output by majority vote is a low credit score when k = 5.

Test Set in k-NN

Notice we used the function knn() twice and used it in different ways. In the first way, the test set was some data we were using to evaluate how good the model was. In the second way, the "test" set was actually a new data point that we wanted a prediction for. We could also have given it many rows of people who we wanted predictions for. But notice that R doesn't know the difference whether what you're putting in for the test set is truly a "test" set where you know the real labels, or a test set where you don't know and want predictions.

What are the modeling assumptions?

In the previous chapter we discussed modeling and modeling assumptions. So what were the modeling assumptions here?

The k-NN algorithm is an example of a nonparametric approach. You had no modeling assumptions about the underlying data-generating distributions, and you weren't attempting to estimate any parameters. But you still made *some* assumptions, which were:

- Data is in some feature space where a notion of "distance" makes sense.
- Training data has been labeled or classified into two or more classes.
- You pick the number of neighbors to use, *k*.
- You're assuming that the *observed features* and the *labels* are somehow associated. They may not be, but ultimately your evaluation metric will help you determine how good the algorithm is at labeling. You might want to add more features and check how that alters the evaluation metric. You'd then be tuning both *which* features you were using and *k*. But as always, you're in danger here of overfitting.

Both linear regression and k-NN are examples of "supervised learning," where you've observed both *x* and *y*, and you want to know the function that brings *x* to *y*. Next up, we'll look at an algorithm you can use when you don't know what the right answer is.

k-means

So far we've only seen supervised learning, where we know beforehand what label (aka the "right answer") is and we're trying to get our model to be as accurate as possible, defined by our chosen evaluation metric.

k-means is the first *unsupervised* learning technique we'll look into, where the goal of the algorithm is to determine the definition of the right answer by finding clusters of data for you.

Let's say you have some kind of data at the user level, e.g., Google+ data, survey data, medical data, or SAT scores.

Start by adding structure to your data. Namely, assume each row of your dataset corresponds to a user as follows:

```
age gender income state household size
```

Your goal is to *segment* the users. This process is known by various names: besides being called segmenting, you could say that you're going to *stratify*, *group*, or *cluster* the data. They all mean finding similar types of users and bunching them together.

Why would you want to do this? Here are a few examples:

- You might want to give different users different experiences. Marketing often does this; for example, to offer toner to people who are known to own printers.
- You might have a model that works better for specific groups. Or you might have different models for different groups.
- Hierarchical modeling in statistics does something like this; for example, to separately model geographical effects from household effects in survey results.

To see why an algorithm like this might be useful, let's first try to construct something by hand. That might mean you'd bucket users using handmade thresholds.

So for an attribute like age, you'd create bins: 20–24, 25–30, etc. The same technique could be used for other attributes like income. States or cities are in some sense their own buckets, but you might want fewer buckets, depending on your model and the number of data points. In that case, you could bucket the buckets and think of "East Coast" and "Midwest" or something like that.

Say you've done that for each attribute. You may have 10 age buckets, 2 gender buckets, and so on, which would result in $10 \times 2 \times 50 \times 10 \times 3 = 30{,}000$ possible bins, which is big.

Imagine this data existing in a five-dimensional space where each axis corresponds to one attribute. So there's a gender axis, an income axis, and so on. You can also label the various possible buckets along the corresponding axes, and if you did so, the resulting grid would consist of every possible bin—a bin for each possible combination of attributes.

Each user would then live in one of those 30,000 five-dimensional cells. But wait, it's highly unlikely you'd want to build a different marketing campaign for each bin. So you'd have to bin the bins…

Now you likely see the utility of having an algorithm to do this for you, especially if you could choose beforehand how many bins you want. That's exactly what k-means is: a *clustering* algorithm where *k* is the number of bins.

2D version

Let's back up to a simpler example than the five-dimensional one we just discussed. Let's say you have users where you know how many ads have been shown to each user (the number of impressions) and how many times each has clicked on an ad (number of clicks).

Figure 3-9 shows a simplistic picture that illustrates what this might look like.

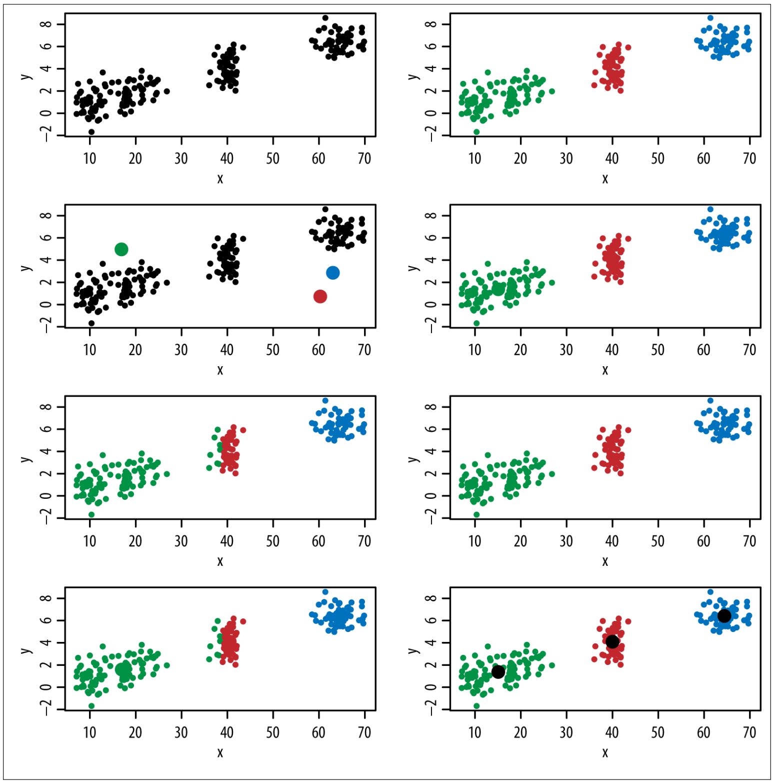

Figure 3-9. Clustering in two dimensions; look at the panels in the left column from top to bottom, and then the right column from top to bottom

Visually you can see in the top-left that the data naturally falls into clusters. This may be easy for you to do with your eyes when it's only in two dimensions and there aren't that many points, but when you get to higher dimensions and more data, you need an algorithm to help with this pattern-finding process. k-means algorithm looks for clusters in *d* dimensions, where *d* is the number of features for each data point.

Here's how the algorithm illustrated in Figure 3-9 works:

1. Initially, you randomly pick *k* centroids (or points that will be the center of your clusters) in *d*-space. Try to make them near the data but different from one another.
2. Then assign each data point to the closest centroid.
3. Move the centroids to the average location of the data points (which correspond to users in this example) assigned to it.
4. Repeat the preceding two steps until the assignments don't change, or change very little.

It's up to you to interpret if there's a natural way to describe these groups once the algorithm's done. Sometimes you'll need to jiggle around *k* a few times before you get natural groupings.

This is an example of *unsupervised* learning because the labels are not known and are instead discovered by the algorithm.

k-means has some known issues:

- Choosing *k* is more an art than a science, although there are bounds: $1 \leq k \leq n$, where *n* is number of data points.
- There are convergence issues—the solution can fail to exist, if the algorithm falls into a loop, for example, and keeps going back and forth between two possible solutions, or in other words, there isn't a single unique solution.
- Interpretability can be a problem—sometimes the answer isn't at all useful. Indeed that's often the biggest problem.

In spite of these issues, it's pretty fast (compared to other clustering algorithms), and there are broad applications in marketing, computer vision (partitioning an image), or as a starting point for other models.

In practice, this is just one line of code in R:

```
kmeans(x, centers, iter.max = 10, nstart = 1,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy",
                     "MacQueen"))
```

Your dataset needs to be a matrix, x, each column of which is one of your features. You specify *k* by selecting centers. It defaults to a certain number of iterations, which is an argument you can change. You can also select the specific algorithm it uses to discover the clusters.

Historical Perspective: k-means

Wait, didn't we just describe the algorithm? It turns out there's more than one way to go after k-means clustering.

The standard k-means algorithm is attributed to separate work by Hugo Steinhaus and Stuart Lloyd in 1957, but it wasn't called "k-means" then. The first person to use that term was James MacQueen in 1967. It wasn't published outside Bell Labs until 1982.

Newer versions of the algorithm are Hartigan-Wong and Lloyd and Forgy, named for their inventors and developed throughout the '60s and '70s. The algorithm we described is the default, Hartigan-Wong. It's fine to use the default.

As history keeps marching on, it's worth checking out the more recent k-means++ developed in 2007 by David Arthur and Sergei Vassilvitskii (now at Google), which helps avoid convergence issues with k-means by optimizing the initial seeds.

Exercise: Basic Machine Learning Algorithms

Continue with the NYC (Manhattan) Housing dataset you worked with in the preceding chapter: *http://abt.cm/1g3A12P*.

- Analyze sales using regression with any predictors you feel are relevant. Justify why regression was appropriate to use.
- Visualize the coefficients and fitted model.
- Predict the neighborhood using a k-NN classifier. Be sure to withhold a subset of the data for testing. Find the variables and the *k* that give you the lowest prediction error.
- Report and visualize your findings.
- Describe any decisions that could be made or actions that could be taken from this analysis.

Solutions

In the preceding chapter, we showed how to explore and clean this dataset, so you'll want to do that first before you build your regression model. Following are two pieces of R code. The first shows how you might go about building your regression models, and the second shows how you might clean and prepare your data and then build a k-NN classifier.

Sample R code: Linear regression on the housing dataset

```
Author: Ben Reddy

model1 <- lm(log(sale.price.n) ~ log(gross.sqft),data=bk.homes)
## what's going on here?

bk.homes[which(bk.homes$gross.sqft==0),]

bk.homes <- bk.homes[which(bk.homes$gross.sqft>0 &
            bk.homes$land.sqft>0),]
model1 <- lm(log(sale.price.n) ~ log(gross.sqft),data=bk.homes)
summary(model1)

plot(log(bk.homes$gross.sqft),log(bk.homes$sale.price.n))
abline(model1,col="red",lwd=2)
plot(resid(model1))

model2 <- lm(log(sale.price.n) ~ log(gross.sqft) +
  log(land.sqft) + factor(neighborhood),data=bk.homes)
summary(model2)
plot(resid(model2))

## leave out intercept for ease of interpretability
model2a <- lm(log(sale.price.n) ~ 0 + log(gross.sqft) +
  log(land.sqft) + factor(neighborhood),data=bk.homes)
summary(model2a)
plot(resid(model2a))

## add building type
model3 <- lm(log(sale.price.n) ~ log(gross.sqft) +
  log(land.sqft) + factor(neighborhood) +
  factor(building.class.category),data=bk.homes)
summary(model3)
plot(resid(model3))

## interact neighborhood and building type
model4 <- lm(log(sale.price.n) ~ log(gross.sqft) +
  log(land.sqft) +  factor(neighborhood)*
  factor(building.class.category),data=bk.homes)
```

```
summary(model4)
plot(resid(model4))
```

Sample R code: K-NN on the housing dataset

```
Author: Ben Reddy
require(gdata)
require(geoPlot)
require(class)

mt <- read.xls("rollingsales_manhattan.xls",
  pattern="BOROUGH",stringsAsFactors=FALSE)
head(mt)
summary(mt)

names(mt) <- tolower(names(mt))

mt$sale.price.n <- as.numeric(gsub("[^[:digit:]]","",
                              mt$sale.price))
sum(is.na(mt$sale.price.n))
sum(mt$sale.price.n==0)

names(mt) <- tolower(names(mt))

## clean/format the data with regular expressions
mt$gross.sqft <- as.numeric(gsub("[^[:digit:]]","",
                            mt$gross.square.feet))
mt$land.sqft <- as.numeric(gsub("[^[:digit:]]","",
                           mt$land.square.feet))

mt$sale.date <- as.Date(mt$sale.date)
mt$year.built <- as.numeric(as.character(mt$year.built))
mt$zip.code <- as.character(mt$zip.code)

## - standardize data (set year built start to 0; land and
gross sq ft; sale price (exclude $0 and possibly others); possi
bly tax block; outside dataset for coords of tax block/lot?)
min_price <- 10000
mt <- mt[which(mt$sale.price.n>=min_price),]

n_obs <- dim(mt)[1]

mt$address.noapt <- gsub("[,][[:print:]]*","",
                    gsub("[ ]+"," ",trim(mt$address)))

mt_add <- unique(data.frame(mt$address.noapt,mt$zip.code,
                 stringsAsFactors=FALSE))
names(mt_add) <- c("address.noapt","zip.code")
mt_add <- mt_add[order(mt_add$address.noapt),]

#find duplicate addresses with different zip codes
```

```
dup <- duplicated(mt_add$address.noapt)
# remove them
dup_add <- mt_add[mt_add$dup,1]
mt_add <- mt_add[(mt_add$address.noapt != dup_add[1] &
          mt_add$address.noapt != dup_add[2]),]

n_add <- dim(mt_add)[1]

# sample 500 addresses so we don't run over our Google Maps
API daily limit (and so we're not waiting forever)
n_sample <- 500
add_sample <- mt_add[sample.int(n_add,size=n_sample),]

# first, try a query with the addresses we have
query_list <- addrListLookup(data.frame(1:n_sample,
  add_sample$address.noapt,rep("NEW YORK",times=n_sample),
  rep("NY",times=n_sample),add_sample$zip.code,
  rep("US",times=n_sample)))[,1:4]

query_list$matched <- (query_list$latitude != 0)

unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

# try changing EAST/WEST to E/W
query_list[unmatched_inds,1:4] <- addrListLookup
  (data.frame(1:unmatched,gsub(" WEST "," W ",
  gsub(" EAST "," E ",add_sample[unmatched_inds,1])),
  rep("NEW YORK",times=unmatched), rep("NY",times=unmatched),
      add_sample[unmatched_inds,2],rep("US",times=unmatched)))[,
1:4]

query_list$matched <- (query_list$latitude != 0)
unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

# try changing STREET/AVENUE to ST/AVE
query_list[unmatched_inds,1:4] <- addrListLookup
  (data.frame(1:unmatched,gsub(" WEST "," W ",
  gsub(" EAST "," E ",gsub(" STREET"," ST",
  gsub(" AVENUE"," AVE",add_sample[unmatched_inds,1])))),
  rep("NEW YORK",times=unmatched), rep("NY",times=unmatched),
      add_sample[unmatched_inds,2],rep("US",times=unmatched)))[,
1:4]

query_list$matched <- (query_list$latitude != 0)
unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

## have to be satisfied for now
add_sample <- cbind(add_sample,query_list$latitude,
  query_list$longitude)
names(add_sample)[3:4] <- c("latitude","longitude")

add_sample <- add_sample[add_sample$latitude!=0,]

add_use <- merge(mt,add_sample)
add_use <- add_use[!is.na(add_use$latitude),]

# map coordinates
map_coords <- add_use[,c(2,4,26,27)]
table(map_coords$neighborhood)
map_coords$neighborhood <- as.factor(map_coords$neighborhood)

geoPlot(map_coords,zoom=12,color=map_coords$neighborhood)


## - knn function
## - there are more efficient ways of doing this,
## but oh well...

map_coords$class <- as.numeric(map_coords$neighborhood)
n_cases <- dim(map_coords)[1]
split <- 0.8

train_inds <- sample.int(n_cases,floor(split*n_cases))
test_inds <- (1:n_cases)[-train_inds]

k_max <- 10
knn_pred <- matrix(NA,ncol=k_max,nrow=length(test_inds))
knn_test_error <- rep(NA,times=k_max)

for (i in 1:k_max) {
    knn_pred[,i] <- knn(map_coords[train_inds,3:4],
  map_coords[test_inds,3:4],cl=map_coords[train_inds,5],k=i)
    knn_test_error[i] <- sum(knn_pred[,i]!=
      map_coords[test_inds,5])/length(test_inds)
}

plot(1:k_max,knn_test_error)
```

Modeling and Algorithms at Scale

The data you've been dealing with so far in this chapter has been pretty small on the Big Data spectrum. What happens to these models and algorithms when you have to scale up to massive datasets?

In some cases, it's entirely appropriate to sample and work with a smaller dataset, or to run the same model across multiple *sharded* datasets. (Sharding is where the data is broken up into pieces and divided among diffrent machines, and then you look at the empirical distribution of the estimators across models.) In other words, there are statistical solutions to these engineering challenges.

However, in some cases we want to fit these models at scale, and the challenge of scaling up models generally translates to the challenge of creating parallelized versions or approximations of the optimization methods. Linear regression at scale, for example, relies on matrix inversions or approximations of matrix inversions.

Optimization with Big Data calls for new approaches and theory—this is the frontier! From a 2013 talk by Peter Richtarik from the University of Edinburgh: "In the Big Data domain classical approaches that rely on optimization methods with multiple iterations are not applicable as the computational cost of even a single iteration is often too excessive; these methods were developed in the past when problems of huge sizes were rare to find. We thus need new methods which would be simple, gentle with data handling and memory requirements, and scalable. Our ability to solve truly huge scale problems goes hand in hand with our ability to utilize modern parallel computing architectures such as multicore processors, graphical processing units, and computer clusters."

Much of this is outside the scope of the book, but a data scientist needs to be aware of these issues, and some of this is discussed in Chapter 14.

Summing It All Up

We've now introduced you to three algorithms that are the basis for the solutions to many real-world problems. If you understand these three, you're already in good shape. If you don't, don't worry, it takes a while to sink in.

Regression is the basis of many forecasting and classification or prediction models in a variety of contexts. We showed you how you can predict a continuous outcome variable with one or more predictors. We'll revisit it again in Chapter 5, where we'll learn *logistic* regression, which can be used for classification of binary outcomes; and in Chapter 6, where we see it in the context of time series modeling. We'll also build up your feature selection skills in Chapter 7.

k-NN and k-means are two examples of clustering algorithms, where we want to group together similar objects. Here the notions of *distance* and *evaluation measures* became important, and we saw there is some subjectivity involved in picking these. We'll explore clustering algorithms including Naive Bayes in the next chapter, and in the context of social networks (Chapter 10). As we'll see, *graph clustering* is an interesting area of research. Other examples of clustering algorithms not explored in this book are *hierarchical clustering* and *model-based clustering*.

For further reading and a more advanced treatment of this material, we recommend the standard classic Hastie and Tibshirani book, *Elements of Statistical Learning* (Springer). For an in-depth exploration of building regression models in a Bayesian context, we highly recommend Andrew Gelman and Jennifer Hill's *Data Analysis using Regression and Multilevel/Hierarchical Models*.

Thought Experiment: Automated Statistician

Rachel attended a workshop in Big Data Mining at Imperial College London in May 2013. One of the speakers, Professor Zoubin Ghahramani from Cambridge University, said that one of his long-term research projects was to build an "automated statistician." What do you think that means? What do you think would go into building one?

Does the idea scare you? Should it?