CHAPTER 7

Extracting Meaning from Data

How do companies extract meaning from the data they have?

In this chapter we hear from two people with very different approaches to that question—namely, William Cukierski from Kaggle and David Huffaker from Google.

William Cukierski

Will went to Cornell for a BA in physics and to Rutgers to get his PhD in biomedical engineering. He focused on cancer research, studying pathology images. While working on writing his dissertation, he got more and more involved in Kaggle competitions (more about Kaggle in a bit), finishing very near the top in multiple competitions, and now works for Kaggle.

After giving us some background in data science competitions and crowdsourcing, Will will explain how his company works for the participants in the platform as well as for the larger community.

Will will then focus on *feature extraction* and *feature selection*. Quickly, feature extraction refers to taking the raw dump of data you have and curating it more carefully, to avoid the "garbage in, garbage out" scenario you get if you just feed raw data into an algorithm without enough forethought. Feature selection is the process of constructing a subset of the data or functions of the data to be the predictors or variables for your models and algorithms.

Background: Data Science Competitions

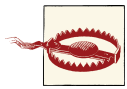
There is a history in the machine learning community of data science competitions—where individuals or teams compete over a period of several weeks or months to design a prediction algorithm. What it predicts depends on the particular dataset, but some examples include whether or not a given person will get in a car crash, or like a particular film. A training set is provided, an evaluation metric determined up front, and some set of rules is provided about, for example, how often competitors can submit their predictions, whether or not teams can merge into larger teams, and so on.

Examples of machine learning competitions include the annual Knowledge Discovery and Data Mining (KDD) competition, the one-time million-dollar Netflix prize (a competition that lasted two years), and, as we'll learn a little later, Kaggle itself.

Some remarks about data science competitions are warranted. First, data science competitions are part of the data science ecosystem—one of the cultural forces at play in the current data science landscape, and so aspiring data scientists ought to be aware of them.

Second, creating these competitions puts one in a position to codify data science, or define its scope. By thinking about the challenges that they've issued, it provides a set of examples for us to explore the central question of this book: what is data science? This is not to say that we will unquestionably accept such a definition, but we can at least use it as a starting point: what attributes of the existing competitions capture data science, and what aspects of data science are missing?

Finally, competitors in the the various competitions get ranked, and so one metric of a "top" data scientist could be their standings in these competitions. But notice that many top data scientists, especially women, and including the authors of this book, don't compete. In fact, there are few women at the top, and we think this phenomenon needs to be explicitly thought through when we expect top ranking to act as a proxy for data science talent.



Data Science Competitions Cut Out All the Messy Stuff

Competitions might be seen as formulaic, dry, and synthetic compared to what you've encountered in normal life. Competitions cut out the messy stuff before you start building models—asking good questions, collecting and cleaning the data, etc.—as well as what happens once you have your model, including visualization and communication. The team of Kaggle data scientists actually spends a lot of time creating the dataset and evaluation metrics, and figuring out what questions to ask, so the question is: while *they're* doing data science, are the contestants?

Background: Crowdsourcing

There are two kinds of crowdsourcing models. First, we have the *distributive* crowdsourcing model, like Wikipedia, which is for relatively simplistic but large-scale contributions. On Wikipedia, the online encyclopedia, anyone in the world can contribute to the content, and there is a system of regulation and quality control set up by volunteers. The net effect is a fairly high-quality compendium of all of human knowledge (more or less).

Then, there's the singular, focused, difficult problems that Kaggle, DARPA, InnoCentive, and other companies specialize in. These companies issue a challenge to the public, but generally only a set of people with highly specialized skills compete. There is usually a cash prize, and glory or the respect of your community, associated with winning.

Crowdsourcing projects have historically had a number of issues that can impact their usefulness. A couple aspects impact the likelihood that people will participate. First off, many lack an evaluation metric. How do you decide who wins? In some cases, the evaluation method isn't always objective. There might be a subjective measure, where the judges decide your design is bad or they just have different taste. This leads to a high barrier to entry, because people don't trust the evaluation criterion. Additionally, one doesn't get recognition until after they've won or at least ranked highly. This leads to high sunk costs for the participants, which people know about in advance—this can become yet another barrier to entry.

Organizational factors can also hinder success. A competition that is not run well conflates participants with mechanical turks: in other words, they assume the competitors to be somewhat mindless and give bad questions and poor prizes. This precedent is just bad for everyone in that it demoralizes data scientists and doesn't help businesses answer more essential questions that get the most from their data. Another common problem is when the competitions don't chunk the work into bite-sized pieces. Either the question is too big to tackle or too small to be interesting.

Learning from these mistakes, we expect a good competition to have a feasible, interesting question, with an evaluation metric that is transparent and objective. The problem is given, the dataset is given, and the metric of success is given. Moreover, the prizes are established up front.

Let's get a bit of historical context for crowdsourcing, since it is not a new idea. Here are a few examples:

- In 1714, the British Royal Navy couldn't measure longitude, and put out a prize worth $6 million in today's dollars to get help. John Harrison, an unknown cabinetmaker, figured out how to make a clock to solve the problem.
- In 2002, the TV network Fox issued a prize for the next pop solo artist, which resulted in the television show American Idol, where contestants compete in an elimination-round style singing competition.
- There's also the X-prize company, which offers "incentivized prize competitions…to bring about radical breakthroughs for the benefits of humanity, thereby inspiring the formation of new industries and the revitalization of markets." A total of $10 million was offered for the Ansari X-prize, a space competition, and $100 million was invested by contestants trying to solve it. Note this shows that it's not always such an efficient process overall—but on the other hand, it could very well be efficient for the people offering the prize if it gets solved.

Terminology: Crowdsourcing and Mechanical Turks

These are a couple of terms that have started creeping into the vernacular over the past few years.

Although crowdsourcing—the concept of using many people to solve a problem independently—is not new, the term was only fairly recently coined in 2006. The basic idea is a that a challenge is issued and contestants compete to find the best solution. *The Wisdom of Crowds* was a book written by James Suriowiecki (Anchor, 2004) with the central thesis that, on average, crowds of people will make better decisions than experts, a related phenomenon. It is only under certain conditions (independence of the individuals rather than group-think where a group of people talking to each other can influence each other into wildly incorrect solutions), where groups of people can arrive at the correct solution. And only certain problems are well-suited to this approach.

Amazon Mechanical Turk is an online crowdsourcing service where humans are given tasks. For example, there might be a set of images that need to be labeled as "happy" or "sad." These labels could then be used as the basis of a training set for a supervised learning problem. An algorithm could then be trained on these human-labeled images to automatically label new images. So the central idea of Mechanical Turk is to have humans do fairly routine tasks to help machines, with the goal of the machines then automating tasks to help the humans! Any researcher with a task they need automated can use Amazon Mechanical Turk as long as they provide compensation for the humans. And any human can sign up and be part of the crowdsourcing service, although there are some quality control issues—if the researcher realizes the human is just labeling every other image as "happy" and not actually looking at the images, then the human won't be used anymore for labeling.

Mechanical Turk is an example of artificial artificial intelligence (yes, double up on the "artificial"), in that the humans are helping the machines helping the humans.

The Kaggle Model

> Being a data scientist is when you learn more and more about more and more, until you know nothing about everything.
>
> — Will Cukierski

Kaggle is a company whose tagline is, "We're making data science a sport." Kaggle forms relationships with companies and with data scientists. For a fee, Kaggle hosts competitions for businesses that essentially want to crowdsource (or leverage the wider data science community) to solve their data problems. Kaggle provides the infrastructure and attracts the data science talent.

They also have in house a bunch of top-notch data scientists, including Will himself. The companies are their paying customers, and they provide datasets and data problems that they want solved. Kaggle crowdsources these problems with data scientists around the world. Anyone can enter. Let's first describe the Kaggle experience for a data scientist and then discuss the customers.

A Single Contestant

In Kaggle competitions, you are given a training set, and also a test set where the *y*s are hidden, but the *x*s are given, so you just use your model to get your predicted *x*s for the test set and upload them into the Kaggle system to see your evaluation score. This way you don't share your actual code with Kaggle unless you win the prize (and Kaggle doesn't have to worry about which version of Python you're running). Note that even giving out just the *x*s is real information—in particular it tells you, for example, what sizes of *x*s your algorithm should optimize for. Also for the purposes of the competition, there is a third hold-out set that contestants never have access to. You don't see the *x*s or the *y*s—that is used to determine the competition winner when the competition closes.

On Kaggle, the participants are encouraged to submit their models up to five times a day during the competitions, which last a few weeks. As contestants submit their predictions, the Kaggle leaderboard updates immediately to display the contestant's current evaluation metric on the hold-out test set. With a sufficient number of competitors doing this, we see a "leapfrogging" between them as shown in Figure 7-1, where one ekes out a 5% advantage, giving others incentive to work

harder. It also establishes a band of accuracy around a problem that you generally don't have—in other words, given no other information, with nobody else working on the problem you're working on, you don't know if your 75% accurate model is the best possible.

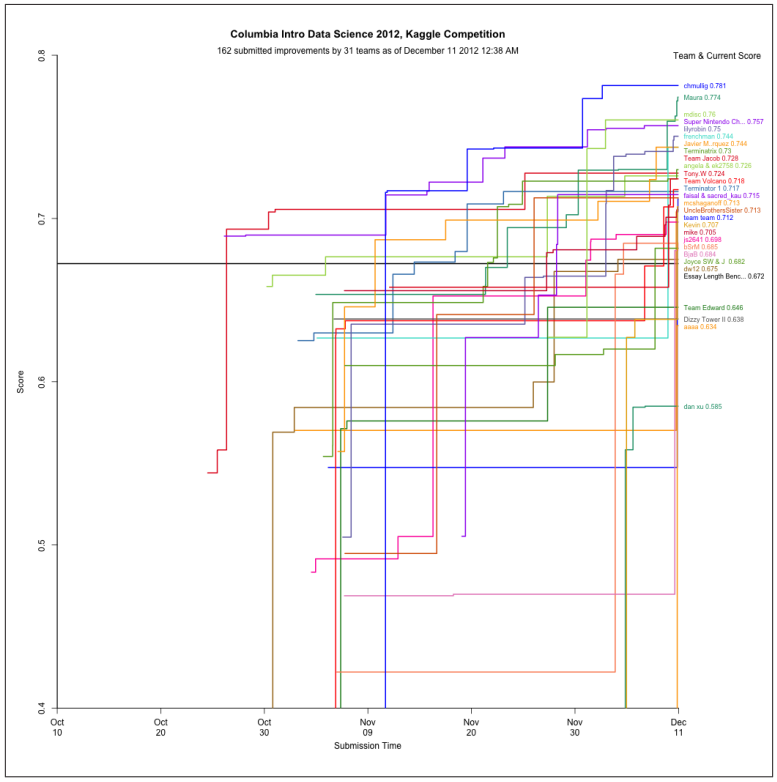


Figure 7-1. Chris Mulligan, a student in Rachel's class, created this leapfrogging visualization to capture the competition in real time as it progressed throughout the semester

This leapfrogging effect is good and bad. It encourages people to squeeze out better performing models, possibly at the risk of overfitting, but it also tends to make models much more complicated as they get better. One reason you don't want competitions lasting too long is that, after a while, the only way to inch up performance is to make things ridiculously complicated. For example, the original Netflix Prize lasted two years and the final winning model was too complicated for them to actually put into production.

Their Customers

So why would companies pay to work with Kaggle? The hole that Kaggle is filling is the following: there's a mismatch between those who need analysis and those with skills. Even though companies desperately need analysis, they tend to hoard data; this is the biggest obstacle for success for those companies that even host a Kaggle competition. Many companies don't host competitions at all, for similar reasons. Kaggle's innovation is that it convinces businesses to share proprietary data with the benefit that their large data problems will be solved for them by crowdsourcing Kaggle's tens of thousands of data scientists around the world.

Kaggle's contests have produced some good results so far. Allstate, the auto insurance company—which has a good actuarial team already—challenged their data science competitors to improve their actuarial model that, given attributes of drivers, approximates the probability of a car crash. The 202 competitors improved Allstate's model by 271% under normalized Gini coefficient (see *http://www.kaggle.com/solutions/casestudies/allstate* for more). Another example includes a company where the prize for competitors was $1,000, and it benefited the company on the order of $100,000.

Is This Fair?

Is it fair to the data scientists already working at the companies that engage with Kaggle? Some of them might lose their job, for example, if the result of the competition is better than the internal model. Is it fair to get people to basically work for free and ultimately benefit a for-profit company? Does it result in data scientists losing their fair market price? Kaggle charges a fee for hosting competitions, and it offers well-defined prizes, so a given data scientist can always choose to not compete. Is that enough?

This seems like it could be a great opportunity for companies, but only while the data scientists of the world haven't realized their value and have extra time on their hands. As soon as they price their skills better they might think twice about working for (almost) free, unless it's for a cause they actually believe in.

When Facebook was recently hiring data scientists, they hosted a Kaggle competition, where the prize was an *interview*. There were 422

competitors. We think it's convenient for Facebook to have interviewees for data science positions in such a posture of gratitude for the mere interview. Cathy thinks this distracts data scientists from asking hard questions about what the data policies are and the underlying ethics of the company.

Kaggle's Essay Scoring Competition

Part of the final exam for the Columbia class was an essay grading contest. The students had to build it, train it, and test it, just like any other Kaggle competition, and group work was encouraged. The details of the essay contest are discussed below, and you access the data at *https://inclass.kaggle.com*.

You are provided access to hand-scored essays so that you can build, train, and test an automatic essay scoring engine. Your success depends upon how closely you can deliver scores to those of human expert graders.

For this competition, there are five essay sets. Each of the sets of essays was generated from a single prompt. Selected essays range from an average length of 150 to 550 words per response. Some of the essays are dependent upon source information and others are not. All responses were written by students ranging in grade levels 7 to 10. All essays were hand graded and were double-scored. Each of the datasets has its own unique characteristics. The variability is intended to test the limits of your scoring engine's capabilities. The data has these columns:

id
: A unique identifier for each individual student essay set

1-5
: An id for each set of essays

essay
: The ascii text of a student's response

rater1
: Rater 1's grade

rater2
: Rater 2's grade

grade
: Resolved score between the raters

Thought Experiment: What Are the Ethical Implications of a Robo-Grader?

Will asked students to consider whether they would want their essays automatically graded by an underlying computer algorithm, and what the ethical implications of automated grading would be. Here are some of their thoughts.

Human graders aren't always fair.

> In the case of doctors, there have been studies where a given doctor is shown the same slide two months apart and gives different diagnoses. We aren't consistent ourselves, even if we think we are. Let's keep that in mind when we talk about the "fairness" of using machine learning algorithms in tricky situations. Machine learning has been used to research cancer, where the stakes are much higher, although there's probably less effort in gaming them.

Are machines making things more structured, and is this inhibiting creativity?

> Some might argue that people *want* things to be standardized. (It also depends on how much you really care about your grade.) It gives us a consistency that we like. People don't want artistic cars, for example; they want safe cars. Even so, is it wise to move from the human to the machine version of same thing for any given thing? Is there a universal answer or is it a case-by-case kind of question?

Is the goal of a test to write a good essay or to do well in a standardized test?

> If the latter, you may as well consider a test like a screening: you follow the instructions, and you get a grade depending on how well you follow instructions. Also, the real profit center for standardized testing is, arguably, to sell books to tell you how to take the tests. How does that translate here? One possible way it could translate would be to have algorithms that game the grader algorithms, by building essays that are graded well but are not written by hand. Then we could see education as turning into a war of machines, between the algorithms the students have and the algorithms the teachers have. We'd probably bet on the students in this war.

Domain Expertise Versus Machine Learning Algorithms

This is a false dichotomy. It isn't either/or. You need both to solve data science problems. However, Kaggle's president Jeremy Howard pissed some domain experts off in a December 2012 *New Scientist* magazine interview with Peter Aldhous, "Specialist Knowledge is Useless and Unhelpful." Here's an excerpt:

PA: What separates the winners from the also-rans?

JH: The difference between the good participants and the bad is the information they feed to the algorithms. You have to decide what to abstract from the data. Winners of Kaggle competitions tend to be curious and creative people. They come up with a dozen totally new ways to think about the problem. The nice thing about algorithms like the random forest is that you can chuck as many crazy ideas at them as you like, and the algorithms figure out which ones work.

PA: That sounds very different from the traditional approach to building predictive models. How have experts reacted?

JH: The messages are uncomfortable for a lot of people. It's controversial because we're telling them: "Your decades of specialist knowledge are not only useless, they're actually unhelpful; your sophisticated techniques are worse than generic methods." It's difficult for people who are used to that old type of science. They spend so much time discussing whether an idea makes sense. They check the visualizations and noodle over it. That is all actively unhelpful.

PA: Is there any role for expert knowledge?

JH: Some kinds of experts are required early on, for when you're trying to work out what problem you're trying to solve. The expertise you need is strategy expertise in answering these questions.

PA: Can you see any downsides to the data-driven, black-box approach that dominates on Kaggle?

JH: Some people take the view that you don't end up with a richer understanding of the problem. But that's just not true: The algorithms tell you what's important and what's not. You might ask why those things are important, but I think that's less interesting. You end up with a predictive model that works. There's not too much to argue about there.

Feature Selection

The idea of feature selection is identifying the subset of data or transformed data that you want to put into your model.

Prior to working at Kaggle, Will placed highly in competitions (which is how he got the job), so he knows firsthand what it takes to build effective predictive models. Feature selection is not only useful for winning competitions—it's an important part of building statistical models and algorithms in general. Just because you have data doesn't mean it *all* has to go into the model.

For example, it's possible you have many redundancies or correlated variables in your raw data, and so you don't want to include all those variables in your model. Similarly you might want to construct new variables by transforming the variables with a logarithm, say, or turning a continuous variable into a binary variable, before feeding them into the model.

Terminology: Features, Explanatory Variables, Predictors

Different branches of academia use different terms to describe the same thing. Statisticians say "explanatory variables" or "dependent variables" or "predictors" when they're describing the subset of data that is the input to a model. Computer scientists say "features."

> Feature extraction and selection are the most important but underrated steps of machine learning. Better features are better than better algorithms.
>
> — Will Cukierski

> We don't have better algorithms, we just have more data.
>
> —Peter Norvig
> *Director of Research for Google*

Is it possible, Will muses, that Norvig really wanted to say we have *better features*? You see, more data is sometimes just more data (example: I can record dice rolls until the cows come home, but after a while I'm not getting any value add because my features will converge), but for the more interesting problems that Google faces, the feature landscape is complex/rich/nonlinear enough to benefit from collecting the data that supports those features.

Why? We are getting bigger and bigger datasets, but that's not always helpful. If the number of features is larger than the number of observations, or if we have a sparsity problem, then large isn't necessarily good. And if the huge data just makes it hard to manipulate because of computational reasons (e.g., it can't all fit on one computer, so the data needs to be sharded across multiple machines) without improving our signal, then that's a net negative.

To improve the performance of your predictive models, you want to improve your feature selection process.

Example: User Retention

Let's give an example for you to keep in mind before we dig into some possible methods. Suppose you have an app that you designed, let's call it Chasing Dragons (shown in Figure 7-2), and users pay a monthly subscription fee to use it. The more users you have, the more money you make. Suppose you realize that only 10% of new users ever come back after the first month. So you have two options to increase your revenue: find a way to increase the retention rate of existing users, or acquire new users. Generally it costs less to keep an existing customer around than to market and advertise to new users. But setting aside that particular cost-benefit analysis of acquistion or retention, let's choose to focus on your user retention situation by building a model that *predicts* whether or not a new user will come back next month based on their behavior this month. You could build such a model in order to *understand* your retention situation, but let's focus instead on building an algorithm that is highly accurate at *predicting*. You might want to use this model to give a free month to users who you predict need the extra incentive to stick around, for example.

A good, crude, simple model you could start out with would be logistic regression, which you first saw back in Chapter 4. This would give you the probability the user returns their second month conditional on their activities in the first month. (There is a rich set of statistical literature called Survival Analysis that could also work well, but that's not necessary in this case—the modeling part isn't want we want to focus on here, it's the data.) You record each user's behavior for the first 30 days after sign-up. You could log every action the user took with timestamps: user clicked the button that said "level 6" at 5:22 a.m., user slew a dragon at 5:23 a.m., user got 22 points at 5:24 a.m., user was shown an ad for deodorant at 5:25 a.m. This would be the data collection phase. Any action the user could take gets recorded.

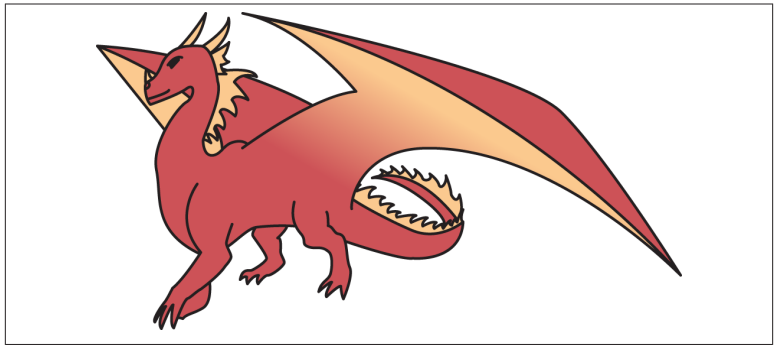


Figure 7-2. Chasing Dragons, the app designed by you

Notice that some users might have thousands of such actions, and other users might have only a few. These would all be stored in time-stamped event logs. You'd then need to process these logs down to a dataset with rows and columns, where each row was a user and each column was a feature. At this point, you shouldn't be selective; you're in the feature generation phase. So your data science team (game designers, software engineers, statisticians, and marketing folks) might sit down and brainstorm features. Here are some examples:

- Number of days the user visited in the first month
- Amount of time until second visit
- Number of points on day j for $j = 1, \ldots, 30$ (this would be 30 separate features)
- Total number of points in first month (sum of the other features)
- Did user fill out Chasing Dragons profile (binary 1 or 0)
- Age and gender of user
- Screen size of device

Use your imagination and come up with as many features as possible. Notice there are redundancies and correlations between these features; that's OK.

Feature Generation or Feature Extraction

This process we just went through of brainstorming a list of features for Chasing Dragons is the process of *feature generation* or *feature extraction*. This process is as much of an art as a science. It's good to have a domain expert around for this process, but it's also good to use your imagination.

In today's technology environment, we're in a position where we can generate tons of features through logging. Contrast this with other contexts like surveys, for example—you're lucky if you can get a survey respondent to answer 20 questions, let alone hundreds.

But how many of these features are just noise? In this environment, when you can capture a lot of data, not all of it might be actually useful information.

Keep in mind that ultimately you're limited in the features you have access to in two ways: whether or not it's possible to even capture the information, and whether or not it even occurs to you at all to try to capture it. You can think of information as falling into the following buckets:

Relevant and useful, but it's impossible to capture it.
: You should keep in mind that there's a lot of information that you're *not* capturing about users—how much free time do they actually have? What other apps have they downloaded? Are they unemployed? Do they suffer from insomnia? Do they have an addictive personality? Do they have nightmares about dragons? Some of this information might be more predictive of whether or not they return next month. There's not much you can do about this, except that it's possible that some of the data you *are* able to capture serves as a proxy by being highly correlated with these unobserved pieces of information: e.g., if they play the game every night at 3 a.m., they might suffer from insomnia, or they might work the night shift.

Relevant and useful, possible to log it, and you did.
: Thankfully it occurred to you to log it during your brainstorming session. It's great that you chose to log it, but just because you chose to log it doesn't mean you know that it's relevant or useful, so that's what you'd like your feature selection process to discover.

Relevant and useful, possible to log it, but you didn't.
: It could be that you didn't think to record whether users uploaded a photo of themselves to their profile, and this action is highly predictive of their likelihood to return. You're human, so sometimes you'll end up leaving out really important stuff, but this shows that your own imagination is a constraint in feature selection. One of the key ways to avoid missing useful features is by doing usability studies (which will be discussed by David Huffaker later on this chapter), to help you think through the user experience and what aspects of it you'd like to capture.

Not relevant or useful, but you don't know that and log it.
: This is what feature selection is all about—you've logged it, but you don't actually need it and you'd like to be able to know that.

Not relevant or useful, and you either can't capture it or it didn't occur to you.
: That's OK! It's not taking up space, and you don't need it.

So let's get back to the logistic regression for your game retention prediction. Let $c_i = 1$ if user i returns to use Chasing Dragons any time in the subsequent month. Again this is crude—you could choose the subsequent week or subsequent two months. It doesn't matter. You just first want to get a working model, and then you can refine it.

Ultimately you want your logistic regression to be of the form:

$$\text{logit}\left(P\left(c_i = 1 \middle| x_i\right)\right) = \alpha + \beta^T \cdot x_i$$

So what should you do? Throw all the hundreds of features you created into one big logistic regression? You could. It's not a terrible thing to do, but if you want to scale up or put this model into production, and get the highest predictive power you can from the data, then let's talk about how you might refine this list of features.

Will found this famous paper by Isabelle Guyon published in 2003 entitled "An Introduction to Variable and Feature Selection" to be a useful resource. The paper focuses mainly on constructing and selecting subsets of features that are *useful* to build a good predictor. This contrasts with the problem of finding or ranking all potentially relevant variables. In it she studies three categories of feature selection methods: filters, wrappers, and embedded methods. Keep the Chasing Dragons prediction example in mind as you read on.

Filters

Filters order possible features with respect to a ranking based on a metric or statistic, such as correlation with the outcome variable. This is sometimes good on a first pass over the space of features, because they then take account of the predictive power of individual features. However, the problem with filters is that you get correlated features. In other words, the filter doesn't care about redundancy. And by treating the features as independent, you're not taking into account possible interactions.

This isn't always bad and it isn't always good, as Isabelle Guyon explains. On the one hand, two redundant features can be more powerful when they are both used; and on the other hand, something that appears useless alone could actually help when combined with another possibly useless-looking feature that an interaction would capture.

Here's an example of a filter: for each feature, run a linear regression with only that feature as a predictor. Each time, note either the p-value or R-squared, and rank order according to the lowest p-value or highest R-squared (more on these two in "Selection criterion" on page 182).

Wrappers

Wrapper feature selection tries to find subsets of features, of some fixed size, that will do the trick. However, as anyone who has studied combinations and permutations knows, the number of possible size k subsets of n things, called $\binom{n}{k}$, grows exponentially. So there's a nasty opportunity for overfitting by doing this.

There are two aspects to wrappers that you need to consider: 1) selecting an algorithm to use to select features and 2) deciding on a selection criterion or filter to decide that your set of features is "good."

Selecting an algorithm

Let's first talk about a set of algorithms that fall under the category of *stepwise regression*, a method for feature selection that involves selecting features according to some selection criterion by either adding or subtracting features to a regression model in a systematic way. There are three primary methods of stepwise regression: forward selection, backward elimination, and a combined approach (forward and backward).

Forward selection

In forward selection you start with a regression model with no features, and gradually add one feature at a time according to which feature improves the model the most based on a selection criterion. This looks like this: build all possible regression models with a single predictor. Pick the best. Now try all possible models that include that best predictor and a second predictor. Pick the best of those. You keep adding one feature at a time, and you stop when your selection criterion no longer improves, but instead gets worse.

Backward elimination

In backward elimination you start with a regression model that includes *all* the features, and you gradually remove one feature at a time according to the feature whose removal makes the biggest improvement in the selection criterion. You stop removing features when removing the feature makes the selection criterion get worse.

Combined approach

Most subset methods are capturing some flavor of minimum-redundancy-maximum-relevance. So, for example, you could have a greedy algorithm that starts with the best feature, takes a few more highly ranked, removes the worst, and so on. This a hybrid approach with a filter method.

Selection criterion

There are a number of selection criteria you could choose from. As a data scientist you have to select which selection criterion to use. Yes! You need a selection criterion to select the selection criterion.

Part of what we wish to impart to you is that in practice, despite the theoretical properties of these various criteria, the choice you make is somewhat arbitrary. One way to deal with this is to try different selection criteria and see how robust your choice of model is. Different selection criterion might produce wildly different models, and it's part of your job to decide what to optimize for and why:

R-squared

Given by the formula $R^2 = 1 - \frac{\sum_i \left(y_i - \hat{y_i}\right)^2}{\sum_i \left(y_i - \bar{y}\right)^2}$, it can be interpreted as the proportion of variance explained by your model.

p-values
: In the context of regression where you're trying to estimate coefficients (the βs), to think in terms of p-values, you make an assumption of there being a *null hypothesis* that the βs are zero. For any given β, the p-value captures the probability of observing the data that you observed, and obtaining the test-statistic (in this case the estimated $\hat{\beta}$) that you got *under the null hypothesis*. Specifically, if you have a low p-value, it is highly unlikely that you would observe such a test-statistic if the null hypothesis actually held. This translates to meaning that (with some confidence) the coefficient is highly likely to be non-zero.

AIC (Akaike Infomation Criterion)
: Given by the formula $2k - 2ln(L)$, where k is the number of parameters in the model and $ln(L)$ is the "maximized value of the log likelihood." The goal is to minimize AIC.

BIC (Bayesian Information Criterion)
: Given by the formula $k^* ln(n) - 2ln(L)$, where k is the number of parameters in the model, n is the number of observations (data points, or users), and $ln(L)$ is the maximized value of the log likelihood. The goal is to minimize BIC.

Entropy
: This will be discussed more in "Embedded Methods: Decision Trees" on page 184.

In practice

As mentioned, stepwise regression is exploring a large space of all possible models, and so there is the danger of overfitting—it will often fit much better in-sample than it does on new out-of-sample data.

You don't have to retrain models at each step of these approaches, because there are fancy ways to see how your objective function (aka selection criterion) changes as you change the subset of features you are trying out. These are called "finite differences" and rely essentially on Taylor Series expansions of the objective function.

One last word: if you have a domain expert on hand, don't go into the machine learning rabbit hole of feature selection unless you've tapped into your expert completely!

Embedded Methods: Decision Trees

Decision trees have an intuitive appeal because outside the context of data science in our every day lives, we can think of breaking big decisions down into a series of questions. See the decision tree in Figure 7-3 about a college student facing the very important decision of how to spend their time.

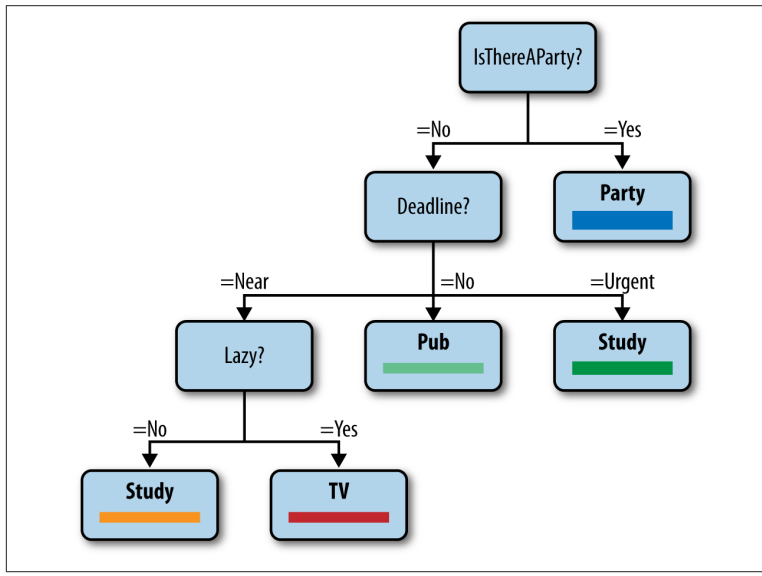


Figure 7-3. Decision tree for college student, aka the party tree (taken with permission from Stephen Marsland's book, Machine Learning: An Algorithmic Perspective [Chapman and Hall/CRC])

This decision is actually dependent on a bunch of factors: whether or not there are any parties or deadlines, how lazy the student is feeling, and what they care about most (parties). The interpretability of decision trees is one of the best features about them.

In the context of a data problem, a decision tree is a classification algorithm. For the Chasing Dragons example, you want to classify users as "Yes, going to come back next month" or "No, not going to come back next month." This isn't really a decision in the colloquial sense, so don't let that throw you. You know that the class of any given user is dependent on many factors (number of dragons the user slew, their age, how many hours they already played the game). And you

want to break it down based on the data you've collected. But how do you construct decision trees from data and what mathematical properties can you expect them to have?

Ultimately you want a tree that is something like Figure 7-4.

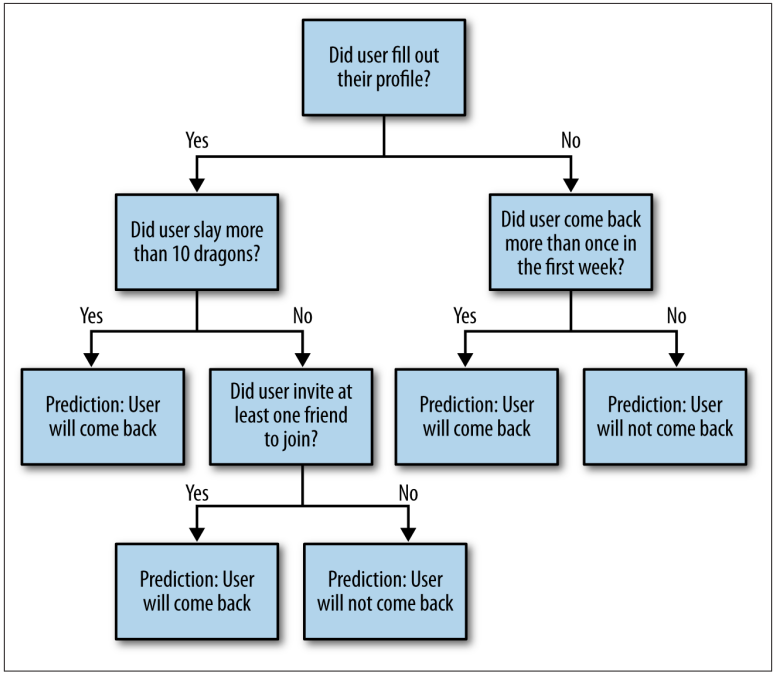


Figure 7-4. Decision tree for Chasing Dragons

But you want this tree to be based on data and not just what you feel like. Choosing a feature to pick at each step is like playing the game 20 Questions really well. You take whatever *the most informative thing* is first. Let's formalize that—we need a notion of "informative."

For the sake of this discussion, assume we break compound questions into multiple yes-or-no questions, and we denote the answers by "0" or "1." Given a random variable X, we denote by $p(X=1)$ and $p(X=0)$ the probability that X is true or false, respectively.

Entropy

To quantify what is the most "informative" feature, we define entropy–effectively a measure for how mixed up something is—for X as follows:

$$H(X) = -p(X=1)\ log_2(p(X=1)) - p(X=0)\ log_2(p(X=0))$$

Note when $p(X=1)=0$ or $p(X=0)=0$, the entropy vanishes, consistent with the fact that:

$$\lim_{t\to 0}\ t \cdot log(t) = 0$$

In particular, if either option has probability zero, the entropy is 0. Moreover, because $p(X=1)=1-p(X=0)$, the entropy is symmetric about 0.5 and maximized at 0.5, which we can easily confirm using a bit of calculus. Figure 7-5 shows a picture of that.

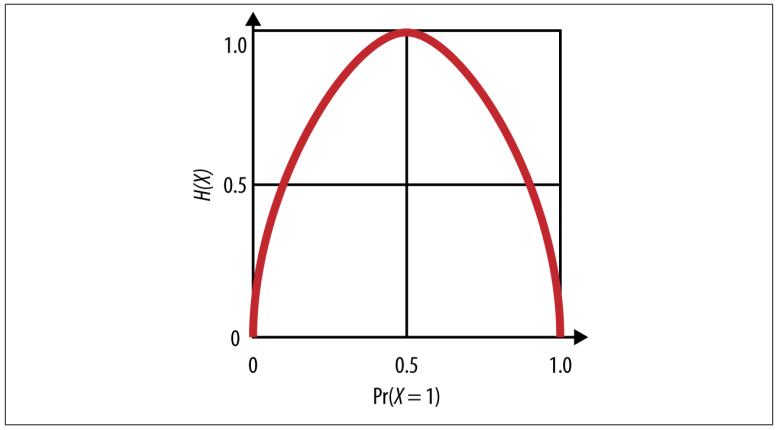


Figure 7-5. Entropy

Mathematically, we kind of get this. But what does it mean in words, and why are we calling it entropy? Earlier, we said that entropy is a measurement of how mixed up something is.

So, for example, if X denotes the event of a baby being born a boy, we'd expect it to be true or false with probability close to 1/2, which corresponds to high entropy, i.e., the bag of babies from which we are selecting a baby is highly mixed.

But if X denotes the event of a rainfall in a desert, then it's low entropy. In other words, the bag of day-long weather events is not highly mixed in deserts.

Using this concept of entropy, we will be thinking of X as the *target* of our model. So, X could be the event that someone buys something on our site. We'd like to know which attribute of the user will tell us the most information about this event X. We will define the *information gain*, denoted $IG(X,a)$, for a given attribute a, as the entropy we *lose* if we know the value of that attribute:

$$IG(X,a) = H(X) - H(X|a)$$

To compute this we need to define $H(X|a)$. We can do this in two steps. For any actual value a_0 of the attribute a we can compute the *specific conditional entropy* $H(X|a=a_0)$ as you might expect:

$$H(X|a=a_0) = -p(X=1|a=a_0)\ log_2\left(p(X=1|a=a_0)\right) - p(X=0|a=a_0)\ log_2\left(p(X=0|a=a_0)\right)$$

and then we can put it all together, for all possible values of a, to get the *conditional entropy* $H(X|a)$:

$$H(X|a) = \Sigma_{ai}\ \ p(a=a_i) \cdot H(X|a=a_i)$$

In words, the conditional entropy asks: how mixed is our bag really if we *know* the value of attribute a? And then information gain can be described as: how much information do we learn about X (or how much entropy do we lose) once we know a?

Going back to how we use the concept of entropy to build decision trees: it helps us decide what feature to split our tree on, or in other words, what's the most informative question to ask?

The Decision Tree Algorithm

You build your decision tree iteratively, starting at the root. You need an algorithm to decide which attribute to split on; e.g., which node should be the next one to identify. You choose that attribute in order to *maximize information gain*, because you're getting the most bang for your buck that way. You keep going until all the points at the end

are in the same class or you end up with no features left. In this case, you take the majority vote.

Often people "prune the tree" afterwards to avoid overfitting. This just means cutting it off below a certain depth. After all, by design, the algorithm gets weaker and weaker as you build the tree, and it's well known that if you build the entire tree, it's often less accurate (with new data) than if you prune it.

This is an example of an embedded feature selection algorithm. (Why embedded?) You don't need to use a filter here because the information gain method is doing your feature selection for you.

Suppose you have your Chasing Dragons dataset. Your outcome variable is *Return*: a binary variable that captures whether or not the user returns next month, and you have tons of predictors. You can use the R library rpart and the function `rpart`, and the code would look like this:

```
# Classification Tree with rpart
library(rpart)

# grow tree
model1 <- rpart(Return ~ profile + num_dragons +
num_friends_invited + gender + age +
num_days, method="class", data=chasingdragons)

printcp(model1) # display the results
plotcp(model1) # visualize cross-validation results
summary(model1) # detailed summary of thresholds picked to
transform to binary

# plot tree
plot(model1, uniform=TRUE,
       main="Classification Tree for Chasing Dragons")
text(model1, use.n=TRUE, all=TRUE, cex=.8)
```

Handling Continuous Variables in Decision Trees

Packages that already implement decision trees can handle continuous variables for you. So you can provide continuous features, and it will determine an optimal threshold for turning the continuous variable into a binary predictor. But if *you* are building a decision tree algorithm yourself, then in the case of continuous variables, you need to determine the correct threshold of a value so that it can be thought of as a binary variable. So you could partition a user's number of dragon slays into "less than 10" and "at least 10," and you'd be getting back to the

binary variable case. In this case, it takes some extra work to decide on the information gain because it depends on the threshold as well as the feature.

In fact, you could think of the decision of where the threshold should live as a separate submodel. It's possible to optimize to this choice by maximizing the entropy on individual attributes, but that's not clearly the best way to deal with continuous variables. Indeed, this kind of question can be as complicated as feature selection itself—instead of a single threshold, you might want to create bins of the value of your attribute, for example. What to do? It will always depend on the situation.

Surviving the Titanic

For fun, Will pointed us to this decision tree for surviving on the Titanic on the BigML website. The original data is from the Encyclopedia Titanica–source code and data are available there. Figure 7-6 provides just a snapshot of it, but if you go to the site, it is interactive.

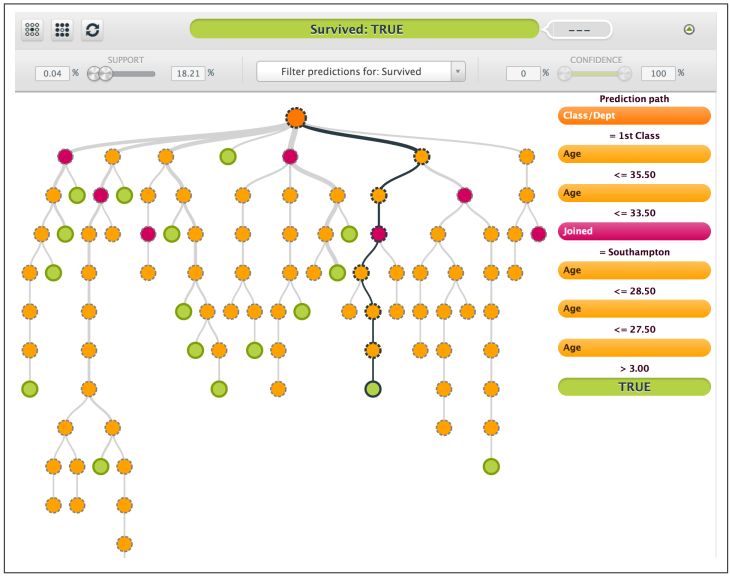


Figure 7-6. Surviving the Titanic

Random Forests

Let's turn to another algorithm for feature selection. Random forests generalize decision trees with *bagging*, otherwise known as *bootstrap aggregating*. We will explain bagging in more detail later, but the effect of using it is to make your models more accurate and more robust, but at the cost of interpretability—random forests are notoriously difficult to understand. They're conversely easy to specify, with two hyperparameters: you just need to specify the number of trees you want in your forest, say N, as well as the number of features to randomly select for each tree, say F.

Before we get into the weeds of the random forest algorithm, let's review bootstrapping. A *bootstrap sample* is a sample with replacement, which means we might sample the same data point more than once. We usually take to the sample size to be 80% of the size of the entire (training) dataset, but of course this parameter can be adjusted depending on circumstances. This is technically a third hyperparameter of our random forest algorithm.

Now to the algorithm. To construct a random forest, you construct N decision trees as follows:

1. For each tree, take a bootstrap sample of your data, and for each node you randomly select F features, say 5 out of the 100 total features.
2. Then you use your entropy-information-gain engine as described in the previous section to decide which among those features you will split your tree on at each stage.

Note that you could decide beforehand how deep the tree should get, or you could prune your trees after the fact, but you typically *don't* prune the trees in random forests, because a great feature of random forests is that they can incorporate idiosyncratic noise.

The code for this would look like:

```
# Author: Jared Lander
#
# we will be using the diamonds data from ggplot
require(ggplot2)

# load and view the diamonds data
data(diamonds)
head(diamonds)
```

```
# plot a histogram with a line marking $12,000
ggplot(diamonds) + geom_histogram(aes(x=price)) +
geom_vline(xintercept=12000)

# build a TRUE/FALSE variable indicating if the price is above
our threshold
diamonds$Expensive <- ifelse(diamonds$price >= 12000, 1, 0)
head(diamonds)

# get rid of the price column
diamonds$price <- NULL

## glmnet
require(glmnet)
# build the predictor matrix, we are leaving out the last
column which is our response
x <- model.matrix(~., diamonds[, -ncol(diamonds)])
# build the response vector
y <- as.matrix(diamonds$Expensive)
# run the glmnet
system.time(modGlmnet <- glmnet(x=x, y=y, family="binomial"))
# plot the coefficient path
plot(modGlmnet, label=TRUE)

# this illustrates that setting a seed allows you to recreate
random results, run them both a few times
set.seed(48872)
sample(1:10)

## decision tree
require(rpart)
# fir a simple decision tree
modTree <- rpart(Expensive ~ ., data=diamonds)
# plot the splits
plot(modTree)
text(modTree)


## bagging (or bootstrap aggregating)
require(boot)
mean(diamonds$carat)
sd(diamonds$carat)
# function for bootstrapping the mean
boot.mean <- function(x, i)
{
    mean(x[i])
}
# allows us to find the variability of the mean
boot(data=diamonds$carat, statistic=boot.mean, R=120)
require(adabag)
```

```
modBag <- bagging(formula=Species ~ ., iris, mfinal=10)

## boosting
require(mboost)
system.time(modglmBoost <- glmboost(as.factor(Expensive) ~ .,
            data=diamonds, family=Binomial(link="logit")))
summary(modglmBoost)
?blackboost

## random forests
require(randomForest)
system.time(modForest <- randomForest(Species ~ ., data=iris,
            importance=TRUE, proximity=TRUE))
```

Criticisms of Feature Selection

Let's address a common criticism of feature selection. Namely, it's no better than data dredging. If we just take whatever answer we get that correlates with our target, however far afield it is, then we could end up thinking that Bangladeshi butter production predicts the S&P. Generally we'd like to first curate the candidate features at least to some extent. Of course, the more observations we have, the less we need to be concerned with spurious signals.

There's a well-known bias-variance tradeoff: a model is "high bias" if it's is too simple (the features aren't encoding enough information). In this case, lots more data doesn't improve our model. On the other hand, if our model is too complicated, then "high variance" leads to overfitting. In this case we want to reduce the number of features we are using.

User Retention: Interpretability Versus Predictive Power

So let's say you built the decision tree, and that it predicts quite well. But should you interpret it? Can you try to find meaning in it?

It could be that it basically tells you "the more the user plays in the first month, the more likely the user is to come back next month," which is kind of useless, and this kind of thing happens when you're doing analysis. It feels circular–*of course* the more they like the app now, the more likely they are to come back. But it could also be that it tells you that showing them ads in the first five minutes *decreases* their chances of coming back, but it's OK to show ads after the first hour, and this

would give you some insight: don't show ads in the first five minutes! Now to study this more, you really would want to do some A/B testing (see Chapter 11), but this initial model and feature selection would help you prioritize the types of tests you might want to run.

It's also worth noting that features that have to do with the user's behavior (user played 10 times this month) are qualitatively different than features that have to do with your behavior (you showed 10 ads, and you changed the dragon to be red instead of green). There's a causation/correlation issue here. If there's a correlation of getting a high number of points in the first month with returning to play next month, does that mean if you just *give* users a high number of points this month without them playing at all, they'll come back? No! It's not the number of points that caused them to come back, it's that they're really into playing the game (a confounding factor), which correlates with both their coming back *and* their getting a high number of points. You therefore would want to do feature selection with *all* variables, but then focus on the ones you can do something about (e.g., show fewer ads) conditional on user attributes.

David Huffaker: Google's Hybrid Approach to Social Research

David's focus is on the effective marriages of both qualitative and quantitative research, and of big and little data. Large amounts of big quantitative data can be more effectively extracted if you take the time to think on a small scale carefully first, and then leverage what you learned on the small scale to the larger scale. And vice versa, you might find patterns in the large dataset that you want to investigate by digging in deeper by doing intensive usability studies with a handful of people, to add more color to the phenomenon you are seeing, or verify interpretations by connecting your exploratory data analysis on the large dataset with relevant academic literature.

David was one of Rachel's colleagues at Google. They had a successful collaboration—starting with complementary skill sets, an explosion of goodness ensued when they were put together to work on Google+ (Google's social layer) with a bunch of other people, especially software engineers and computer scientists. David brings a social scientist perspective to the analysis of social networks. He's strong in quantitative methods for understanding and analyzing online social behavior. He got a PhD in media, technology, and society from Northwestern

University. David spoke about Google's approach to social research to encourage the class to think in ways that connect the qualitative to the quantitative, and the small-scale to the large-scale.

Google does a good job of putting people together. They blur the lines between research and development. They even wrote about it in this July 2012 position paper: Google's Hybrid Approach to Research. Their researchers are embedded on product teams. The work is iterative, and the engineers on the team strive to have near-production code from day 1 of a project. They leverage engineering infrastructure to deploy experiments to their mass user base and to rapidly deploy a prototype at scale. Considering the scale of Google's user base, redesign as they scale up is not a viable option. They instead do experiments with smaller groups of users.

Moving from Descriptive to Predictive

David suggested that as data scientists, we consider how to move into an experimental design so as to move to a causal claim between variables rather than a descriptive relationship. In other words, our goal is to move from the descriptive to the predictive.

As an example, he talked about the genesis of the "circle of friends" feature of Google+. Google knows people want to selectively share; users might send pictures to their family, whereas they'd probably be more likely to send inside jokes to their friends. Google came up with the idea of circles, but it wasn't clear if people would use them. How can Google answer the question of whether people will use circles to organize their social network? It's important to know what motivates users when they decide to share.

Google took a *mixed-method approach*, which means they used multiple methods to triangulate on findings and insights. Some of their methods were small and qualitative, some of them larger and quantitative.

Given a random sample of 100,000 users, they set out to determine the popular names and categories of names given to circles. They identified 168 active users who filled out surveys and they had longer interviews with 12. The depth of these interviews was weighed against selection bias inherent in finding people that are willing to be interviewed.

They found that the majority were engaging in selective sharing, that most people used circles, and that the circle names were most often work-related or school-related, and that they had elements of a strong-link ("epic bros") or a weak-link ("acquaintances from PTA").

They asked the survey participants why they share content. The answers primarily came in three categories. First, the desire to share about oneself: personal experiences, opinions, etc. Second, discourse: people want to participate in a conversation. Third, evangelism: people like to spread information.

Next they asked participants why they choose their audiences. Again, three categories: first, privacy—many people were public or private by default. Second, relevance: they wanted to share only with those who may be interested, and they don't want to pollute other people's data stream. Third, distribution: some people just want to maximize their potential audience.

The takeaway from this study was that people do enjoy selectively sharing content, depending on context and the audience. So Google has to think about designing features for the product around content, context, and audience. They want to not just keep these findings in mind for design, but also when they do data science at scale.

Thought Experiment: Large-Scale Network Analysis

We'll dig more into network analysis in Chapter 10 with John Kelly. But for now, think about how you might take the findings from the Google+ usability studies and explore selectively sharing content on a massive scale using data. You can use large data and look at connections between actors like a graph. For Google+, the users are the nodes and the edges (directed) are "in the same circle." Think about what data you would want to log, and then how you might test some of the hypotheses generated from speaking with the small group of engaged users.

As data scientists, it can be helpful to think of different structures and representations of data, and once you start thinking in terms of networks, you can see them everywhere.

Other examples of networks:

- The nodes are users in Second Life, and the edges correspond to interactions between users. Note there is more than one possible way for players to interact in this game, leading to potentially different kinds of edges.
- The nodes are websites, the (directed) edges are links.
- Nodes are theorems, directed edges are dependencies.

Social at Google

As you may have noticed, "social" is a layer across all of Google. Search now incorporates this layer: if you search for something you might see that your friend "+1"ed it. This is called a *social annotation*. It turns out that people care more about annotation when it comes from someone with domain expertise rather than someone you're very close to. So you might care more about the opinion of a wine expert at work than the opinion of your mom when it comes to purchasing wine.

Note that sounds obvious but if you started the other way around, asking who you'd trust, you might start with your mom. In other words, "close ties"–even if you can determine those—are not the best feature to rank annotations. But that begs the question, what is? Typically in a situation like this, data scientists might use click-through rate, or how long it takes to click.

In general you need to always keep in mind a quantitative metric of success. This defines success for you, so you have to be careful.

Privacy

Human-facing technology has thorny issues of privacy, which makes stuff hard. Google conducted a survey of how people felt uneasy about content. They asked, how does it affect your engagement? What is the nature of your privacy concerns?

Turns out there's a strong correlation between privacy concern and low engagement, which isn't surprising. It's also related to how well you understand what information is being shared, and the question of when you post something, where it goes, and how much control you have over it. When you are confronted with a huge pile of complicated settings, you tend to start feeling passive.

The survey results found broad categories of concern as follows:

Identity theft

- Financial loss

Digital world

- Access to personal data
- Really private stuff I searched on
- Unwanted spam
- Provocative photo (oh *&!$ my boss saw that)
- Unwanted solicitation
- Unwanted ad targeting

Physical world

- Offline threats/harassment
- Harm to my family
- Stalkers
- Employment risks

Thought Experiment: What Is the Best Way to Decrease Concern and Increase Understanding and Control?

So given users' understandable concerns about privacy, students in Rachel's class brainstormed some potential solutions that Google could implement (or that anyone dealing with user-level data could consider).

Possibilities:

- You could write and post a manifesto of your data policy. Google tried that, but it turns out nobody likes to read manifestos.
- You could educate users on your policies a la the Netflix feature "because you liked this, we think you might like this." But it's not always so easy to explain things in complicated models.
- You could simply get rid of all stored data after a year. But you'd still need to explain that you do that.

Maybe we could rephrase the question: how do you design privacy settings to make it easier for people? In particular, how do you make it transparent? Here are some ideas along those lines:

- Make a picture or graph of where data is going.
- Give people a privacy switchboard.
- Provide access to quick settings.
- Make the settings you show people categorized by "things you don't have a choice about" versus "things you do" for the sake of clarity.
- Best of all, you could make reasonable default setting so people don't have to worry about it.

David left us with these words of wisdom: as you move forward and have access to Big Data, you really should complement them with qualitative approaches. Use mixed methods to come to a better understanding of what's going on. Qualitative surveys can really help.

CHAPTER 8

Recommendation Engines: Building a User-Facing Data Product at Scale

Recommendation engines, also called recommendation systems, are the quintessential data product and are a good starting point when you're explaining to non–data scientists what you do or what data science really is. This is because many people have interacted with recommendation systems when they've been suggested books on Amazon.com or gotten recommended movies on Netflix. Beyond that, however, they likely have not thought much about the engineering and algorithms underlying those recommendations, nor the fact that their behavior when they buy a book or rate a movie is generating data that then feeds back into the recommendation engine and leads to (hopefully) improved recommendations for themselves and other people.

Aside from being a clear example of a product that literally uses data as its fuel, another reason we call recommendation systems "quintessential" is that building a solid recommendation system end-to-end requires an understanding of linear algebra *and* an ability to code; it also illustrates the challenges that Big Data poses when dealing with a problem that makes intuitive sense, but that can get complicated when implementing its solution at scale.

In this chapter, Matt Gattis walks us through what it took for him to build a recommendation system for Hunch.com—including why he made certain decisions, and how he thought about trade-offs between

various algorithms when building a large-scale engineering system and infrastructure that powers a user-facing product.

Matt graduated from MIT in CS, worked at SiteAdvisor, and co-founded Hunch as its CTO. Hunch is a website that gives you recommendations of any kind. When they started out, it worked like this: they'd ask people a bunch of questions (people seem to love answering questions), and then someone could ask the engine questions like, "What cell phone should I buy?" or, "Where should I go on a trip?" and it would give them advice. They use machine learning to give better and better advice. Matt's role there was doing the R&D for the underlying recommendation engine.

At first, they focused on trying to make the questions as fun as possible. Then, of course, they saw things needing to be asked that would be extremely informative as well, so they added those. Then they found that they could ask merely 20 questions and then predict the rest of them with 80% accuracy. They were questions that you might imagine and some that are surprising, like whether people were competitive versus uncompetitive, introverted versus extroverted, thinking versus perceiving, etc.—not unlike MBTI.

Eventually Hunch expanded into more of an API model where they crawl the Web for data rather than asking people direct questions. The service can also be used by third parties to personalize content for a given site—a nice business proposition that led to eBay acquiring Hunch.com.

Matt has been building code since he was a kid, so he considers software engineering to be his strong suit. Hunch requires cross-domain experience so he doesn't consider himself a domain expert in any focused way, except for recommendation systems themselves.

The best quote Matt gave us was this: "Forming a data team is kind of like planning a heist." He means that you need people with all sorts of skills, and that one person probably can't do everything by herself. (Think Ocean's Eleven, but sexier.)

A Real-World Recommendation Engine

Recommendation engines are used all the time—what movie would you like, knowing other movies you liked? What book would you like, keeping in mind past purchases? What kind of vacation are you likely to embark on, considering past trips?

There are plenty of different ways to go about building such a model, but they have very similar feels if not implementation. We're going to show you how to do one relatively simple but compete version in this chapter.

To set up a recommendation engine, suppose you have *users*, which form a set U; and you have *items* to recommend, which form a set V. As Kyle Teague told us in Chapter 6, you can denote this as a bipartite graph (shown again in Figure 8-1) if each user and each item has a node to represent it—there are lines from a user to an item if that user has expressed an opinion about that item. Note they might not always *love* that item, so the edges could have weights: they could be positive, negative, or on a continuous scale (or discontinuous, but many-valued like a star system). The implications of this choice can be heavy but we won't delve too deep here—for us they are numeric ratings.

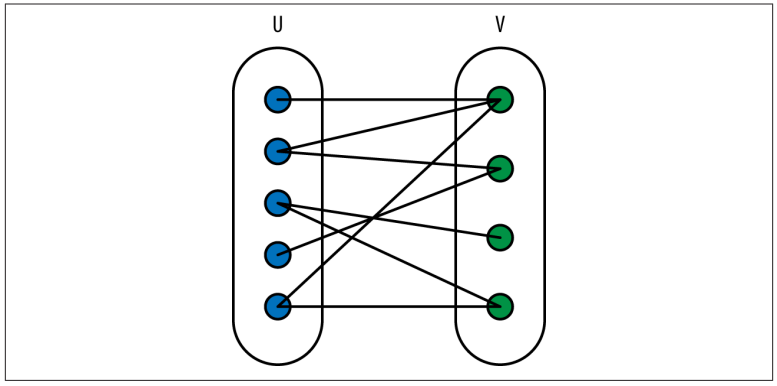


Figure 8-1. Bipartite graph with users and items (television shows) as nodes

Next up, you have training data in the form of some preferences—you know some of the opinions of some of the users on some of the items. From those training data, you want to predict other preferences for your users. That's essentially the output for a recommendation engine.

You may also have metadata on users (i.e., they are male or female, etc.) or on items (the color of the product). For example, users come to your website and set up accounts, so you may know each user's gender, age, and preferences for up to three items.

You represent a given user as a vector of features, sometimes including only metadata—sometimes including only preferences (which would lead to a sparse vector because you don't know all the user's opinions)—and sometimes including both, depending on what you're doing with the vector. Also, you can sometimes bundle all the user vectors together to get a big user matrix, which we call U, through abuse of notation.

Nearest Neighbor Algorithm Review

Let's review the nearest neighbor algorithm (discussed in Chapter 3): if you want to predict whether user A likes something, you look at a user B *closest* to user A who has an opinion, then you assume A's opinion is the same as B's. In other words, once you've identified a similar user, you'd then find something that user A hadn't rated (which you'd assume meant he hadn't ever seen that movie or bought that item), but that user B *had* rated and liked and use that as your recommendation for user A.

As discussed in Chapter 3, to implement this you need a metric so you can measure distance. One example when the opinions are binary: Jaccard distance, i.e., 1–(the number of things they both like divided by the number of things either of them likes). Other examples include cosine similarity or Euclidean distance.

Which Metric Is Best?
You might get a different answer depending on which metric you choose. But that's a good thing. Try out lots of different distance functions and see how your results change and think about why.

Some Problems with Nearest Neighbors

So you *could* use nearest neighbors; it makes some intuitive sense that you'd want to recommend items to people by finding similar people and using those people's opinions to generate ideas and recommendations. But there are a number of problems nearest neighbors poses. Let's go through them:

Curse of dimensionality
: There are too many dimensions, so the closest neighbors are too far away from each other to realistically be considered "close."

Overfitting
: Overfitting is also a problem. So one guy is closest, but that could be pure noise. How do you adjust for that? One idea is to use k-NN, with, say, k=5 rather than k=1, which increases the noise.

Correlated features
: There are tons of features, moreover, that are highly correlated with each other. For example, you might imagine that as you get older you become more conservative. But then counting both age and politics would mean you're double counting a single feature in some sense. This would lead to bad performance, because you're using redundant information and essentially placing double the weight on some variables. It's preferable to build in an understanding of the correlation and project onto smaller dimensional space.

Relative importance of features
: Some features are more informative than others. Weighting features may therefore be helpful: maybe your age has nothing to do with your preference for item 1. You'd probably use something like covariances to choose your weights.

Sparseness
: If your vector (or matrix, if you put together the vectors) is too sparse, or you have lots of missing data, then most things are unknown, and the Jaccard distance means nothing because there's no overlap.

Measurement errors
: There's measurement error (also called reporting error): people may lie.

Computational complexity
: There's a calculation cost—computational complexity.

Sensitivity of distance metrics
: Euclidean distance also has a scaling problem: distances in age outweigh distances for other features if they're reported as 0 (for don't like) or 1 (for like). Essentially this means that raw euclidean distance doesn't make much sense. Also, old and young people might think one thing but middle-aged people something else. We seem to be assuming a linear relationship, but it may not exist. Should you be binning by age group instead, for example?

Preferences change over time
: User preferences may also change over time, which falls outside the model. For example, at eBay, they might be buying a printer, which makes them only want ink for a short time.

Cost to update
: It's also expensive to update the model as you add more data.

The biggest issues are the first two on the list, namely overfitting and the curse of dimensionality problem. How should you deal with them? Let's think back to a method you're already familiar with—linear regression—and build up from there.

Beyond Nearest Neighbor: Machine Learning Classification

We'll first walk through a simplification of the actual machine learning algorithm for this—namely we'll build a separate linear regression model for each item. With each model, we could then predict for a given user, knowing their attributes, whether they would like the item corresponding to that model. So one model might be for predicting whether you like *Mad Men* and another model might be for predicting whether you would like Bob Dylan.

Denote by $f_{i,j}$ user i's stated preference for item j if you have it (or user i's attribute, if item j is a metadata item like age or is_logged_in). This is a subtle point but can get a bit confusing if you don't internalize this: you are treating metadata here *also* as if it's an "item." We mentioned this before, but it's OK if you didn't get it—hopefully it will click more now. When we said we could predict what you might like, we're also saying we could use this to predict your *attribute*; i.e., if we didn't *know* if you were a male/female because that was missing data or we had never asked you, we might be able to predict that.

To let this idea settle even more, assume we have three numeric attributes for each user, so we have $f_{i,1}, f_{i,2}$, and $f_{i,3}$. Then to guess user i's preference on a new item (we temporarily denote this estimate by p_i) we can look for the best choice of β_k so that:

$$p_i = \beta_1 f_{1,i} + \beta_2 f_{2,i} + \beta_3 f_{3,i} + \epsilon$$

The good news: You know how to estimate the coefficients by linear algebra, optimization, and statistical inference: specifically, linear regression.

The bad news: This model only works for one item, and to be complete, you'd need to build as many models as you have items. Moreover, you're not using other items' information at all to create the model for a given item, so you're not leveraging other pieces of information.

But wait, there's more good news: This solves the "weighting of the features" problem we discussed earlier, because linear regression coefficients *are* weights.

Crap, more bad news: overfitting is *still* a problem, and it comes in the form of having huge coefficients when you don't have enough data (i.e., not enough opinions on given items).

Let's make more rigorous the preceding argument that huge coefficients imply overfitting, or maybe even just a bad model. For example, if two of your variables are exactly the same, or are nearly the same, then the coefficients on one can be 100,000 and the other can be –100,000 and really they add nothing to the model. In general you should always have some *prior* on what a reasonable size would be for your coefficients, which you do by normalizing all of your variables and imagining what an "important" effect would translate to in terms of size of coefficients—anything much larger than that (in an absolute value) would be suspect.

To solve the overfitting problem, you impose a Bayesian prior that these weights shouldn't be too far out of whack—this is done by adding a penalty term for large coefficients. In fact, this ends up being equivalent to adding a prior matrix to the covariance matrix. That solution depends on a single parameter, which is traditionally called λ.

But that begs the question: how do you choose λ? You could do it experimentally: use some data as your training set, evaluate how well you did using particular values of λ, and adjust. That's kind of what happens in real life, although note that it's not exactly consistent with the idea of estimating what a reasonable size would be for your coefficient.

You can't use this penalty term for large coefficients and assume the "weighting of the features" problem is still solved, because in fact you'd be penalizing some coefficients way more than others if they start out on different scales. The easiest way to get around this is to normalize your variables before entering them into the model, similar to how we did it in Chapter 6. If you have some reason to think certain variables should have larger coefficients, then you can normalize different variables with different means and variances. At the end of the day, the way you normalize is again equivalent to imposing a prior.

A final problem with this prior stuff: although the problem will have a unique solution (as in the penalty will have a unique minimum) if you make λ large enough, by that time you may not be solving the problem you care about. Think about it: if you make λ absolutely huge, then the coefficients will all go to zero and you'll have no model at all.

The Dimensionality Problem

OK, so we've tackled the overfitting problem, so now let's think about overdimensionality—i.e., the idea that you might have tens of thousands of items. We typically use both Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) to tackle this, and we'll show you how shortly.

To understand how this works before we dive into the math, let's think about how we reduce dimensions and create "latent features" internally every day. For example, people invent concepts like "coolness," but we can't *directly* measure how cool someone is. Other people exhibit different patterns of behavior, which we internally map or reduce to our one dimension of "coolness." So coolness is an example of a latent feature in that it's unobserved and not measurable directly, and we could think of it as reducing dimensions because perhaps it's a combination of many "features" we've observed about the person and implictly weighted in our mind.

Two things are happening here: the dimensionality is reduced into a single feature and the latent aspect of that feature.

But in this algorithm, we don't decide which latent factors to care about. Instead we let the machines do the work of figuring out what the important latent features are. "Important" in this context means

they explain the variance in the answers to the various questions—in other words, they model the answers efficiently.

Our goal is to build a model that has a representation in a low dimensional subspace that gathers "taste information" to generate recommendations. So we're saying here that taste is *latent* but can be approximated by putting together all the observed information we *do* have about the user.

Also consider that most of the time, the rating questions are binary (yes/no). To deal with this, Hunch created a separate variable for every question. They also found that comparison questions may be better at revealing preferences.

Time to Brush Up on Your Linear Algebra if You Haven't Already

A lot of the rest of this chapter likely won't make sense (and we want it to make sense to you!) if you don't know linear algebra and understand the terminology and geometric interpretation of words like *rank* (hint: the linear algebra definition of that word has nothing to do with ranking algorithms), *orthogonal*, *transpose*, *base*, *span*, and *matrix decomposition*. Thinking about data in matrices as points in space, and what it would mean to transform that space or take subspaces can give you insights into your models, why they're breaking, or how to make your code more efficient. This isn't just a mathematical exercise for the sake of it—although there is elegance and beauty in it—it can be the difference between a star-up that fails and a start-up that gets acquired by eBay. We recommend Khan Academy's excellent free online introduction to linear algebra if you need to brush up your linear algebra skills.

Singular Value Decomposition (SVD)

Hopefully we've given you some intuition about what we're going to do. So let's get into the math now starting with singular value decomposition. Given an $m \times n$ matrix X of rank k, it is a theorem from linear algebra that we can always compose it into the product of three matrices as follows:

$$X = USV^T$$

where U is $m \times k$, S is $k \times k$, and V is $k \times n$, the columns of U and V are pairwise orthogonal, and S is diagonal. Note the standard statement of SVD is slightly more involved and has U and V both square unitary matrices, and has the middle "diagonal" matrix a rectangular. We'll be using this form, because we're going to be taking approximations to X of increasingly smaller rank. You can find the proof of the existence of this form as a step in the proof of existence of the general form here.

Let's apply the preceding matrix decomposition to our situation. X is our original dataset, which has users' ratings of items. We have m users, n items, and k would be the rank of X, and consequently would also be an upper bound on the number d of latent variables we decide to care about—note we choose d whereas m, n, and k are defined through our training dataset. So just like in k-NN, where k is a tuning parameter (different k entirely—not trying to confuse you!), in this case, d is the tuning parameter.

Each row of U corresponds to a *user*, whereas V has a row for each *item*. The values along the diagonal of the square matrix S are called the "singular values." They measure the importance of each latent variable—the most important latent variable has the biggest singular value.

Important Properties of SVD

Because the columns of U and V are orthogonal to each other, you can order the columns by singular values via a base change operation. That way, if you put the columns in decreasing order of their corresponding singular values (which you do), then the dimensions are ordered by importance from highest to lowest. You can take lower rank approximation of X by throwing away part of S. In other words, replace S by a submatrix taken from the upper-left corner of S.

Of course, if you cut off part of S you'd have to simultaneously cut off part of U and part of V, but this is OK because you're cutting off the *least important* vectors. This is essentially how you choose the number of latent variables d—you no longer have the original matrix X anymore, only an approximation of it, because d is typically much smaller than k, but it's still pretty close to X. This is what people mean when they talk about "compression," if you've ever heard that term thrown around. There is often an important interpretation to the values in the matrices U and V. For example, you can see, by using SVD, that the

"most important" latent feature is often something like whether someone is a man or a woman.

How would you actually use this for recommendation? You'd take X, fill in all of the empty cells with the average rating for that item (you don't want to fill it in with 0 because that might mean something in the rating system, and SVD can't handle missing values), and then compute the SVD. Now that you've decomposed it this way, it means that you've captured latent features that you can use to compare users if you want to. But that's not what you want—you want a prediction. If you multiply out the U, S, and V^T together, you get an approximation to X—or a prediction, $\hat{X}$—so you can predict a rating by simply looking up the entry for the appropriate user/item pair in the matrix $\hat{X}$.

Going back to our original list of issues with nearest neighbors in "Some Problems with Nearest Neighbors" on page 202, we want to avoid the problem of missing data, but that is not fixed by the preceding SVD approach, nor is the computational complexity problem. In fact, SVD is extremely computationally expensive. So let's see how we can improve on that.

Principal Component Analysis (PCA)

Let's look at another approach for predicting preferences. With this approach, you're still looking for U and V as before, but you don't need S anymore, so you're just searching for U and V such that:

$$X \equiv U \cdot V^T$$

Your optimization problem is that you want to minimize the discrepency between the actual X and your approximation to X via U and V measured via the squared error:

$$argmin \Sigma_{i,j} \left(x_{i,j} - u_i \cdot v_j\right)^2$$

Here you denote by u_i the row of U corresponding to user i, and similarly you denote by v_j the row of V corresponding to item j. As usual, items can include metadata information (so the age vectors of all the users will be a row in V).

Then the dot product $u_i \cdot v_j$ is the *predicted preference* of user i for item j, and you want that to be as close as possible to the *actual preference* $x_{i,j}$.

So, you want to find the best choices of U and V that minimize the squared differences between prediction and observation on everything you actually know, and the idea is that if it's really good on stuff you know, it will also be good on stuff you're guessing. This should sound familiar to you—it's mean squared error, like we used for linear regression.

Now you get to choose a parameter, namely the number d defined as *how may latent features you want to use*. The matrix U will have a row for each user and a column for each latent feature, and the matrix V will have a row for each item and a column for each latent feature.

How do you choose d? It's typically about 100, because it's more than 20 (as we told you, through the course of developing the product, we found that we had a pretty good grasp on someone if we ask them 20 questions) and it's as much as you care to add before it's computationally too much work.

The resulting latent features are the basis of a well-defined subspace of the total n-dimensional space of potential latent variables. There's no reason to think this solution is unique if there are a bunch of missing values in your "answer" matrix. But that doesn't necessarily matter, because you're just looking for *a solution*.

Theorem: The resulting latent features will be uncorrelated

We already discussed that correlation was an issue with k-NN, and who wants to have redundant information going into their model? So a nice aspect of these latent features is that they're uncorrelated. Here's a sketch of the proof:

Say we've found matrices U and V with a fixed product $U \cdot V = X$ such that the squared error term is minimized. The next step is to find the best U and V such that their entries are small—actually we're minimizing the sum of the squares of the entries of U and V. But we can modify U with any invertible $d \times d$ matrix G as long as we modify V by its inverse: $U \cdot V = (U \cdot G) \cdot (G^{-1} \cdot V) = X$.

Assume for now we only modify with determinant 1 matrices G; i.e., we restrict ourselves to volume-preserving transformations. If we ignore for now the size of the entries of V and concentrate only on the size of the entries of U, we are minimizing the surface area of a d-dimensional parallelepiped in n space (specifically, the one generated by the columns of U) where the volume is fixed. This is achieved by making the sides of the parallelepiped mutually orthogonal, which is the same as saying the latent features will be uncorrelated.

But don't forget, we've ignored V! However, it turns out that V's rows will also be mutually orthogonal when we force U's columns to be. This is not hard to see if you keep in mind X has its SVD as discussed previously. In fact, the SVD and this form $U \cdot V$ have a lot in common, and some people just call this an SVD algorithm, even though it's not quite.

Now we allow modifications with nontrivial determinant—so, for example, let G be some scaled version of the identity matrix. Then if we do a bit of calculus, it turns out that the best choice of scalar (i.e., to minimize the sum of the squares of the entries of U and of V) is in fact the *geometric mean* of those two quantities, which is cool. In other words, we're minimizing the arithmetic mean of them with a single parameter (the scalar) and the answer is the geometric mean.

So that's the proof. Believe us?

Alternating Least Squares

But how do you do this? How do you actually find U and V? In reality, as you will see next, you're not first minimizing the squared error and then minimizing the size of the entries of the matrices U and V. You're actually doing both at the same time.

So your goal is to find U and V by solving the optimization problem described earlier. This optimization doesn't have a nice closed formula like ordinary least squares with one set of coefficients. Instead, you need an iterative algorithm like gradient descent. As long as your problem is convex you'll converge OK (i.e., you won't find yourself at a local, but not global, maximum), and you can force your problem to be convex using regularization.

Here's the algorithm:

- Pick a random V.

- Optimize U while V is fixed.
- Optimize V while U is fixed.
- Keep doing the preceding two steps until you're not changing very much at all. To be precise, you choose an ϵ and if your coefficients are each changing by less than ϵ, then you declare your algorithm "converged."

Theorem with no proof: The preceding algorithm will converge if your prior is large enough

If you enlarge your prior, you make the optimization easier because you're artificially creating a more convex function—on the other hand, if your prior is huge, then all your coefficients will be zero anyway, so that doesn't really get you anywhere. So actually you might not want to enlarge your prior. Optimizing your prior is philosophically screwed because how is it a *prior* if you're back-fitting it to do what you want it to do? Plus you're mixing metaphors here to some extent by searching for a close approximation of X at the same time you are minimizing coefficients. The more you care about coefficients, the less you care about X. But in actuality, you only want to care about X.

Fix V and Update U

The way you do this optimization is user by user. So for user i, you want to find:

$$argmin_{u_i} \Sigma_{j \in P_i} (p_{i,j} - u_i * v_j)^2$$

where v_j is fixed. In other words, you just care about *this user* for now.

But wait a minute, this is the same as linear least squares, and has a closed form solution! In other words, set:

$$u_i = \left(V_{*,i}^{\tau} V_{*,i}\right)^{-1} V_{*,i}^{\tau} P_{*i},$$

where $V_{*,i}$ is the subset of V for which you have preferences coming from user i. Taking the inverse is easy because it's $d \times d$, which is small. And there aren't that many preferences per user, so solving this many times is really not that hard. Overall you've got a doable update for U.

When you fix U and optimize V, it's analogous—you only ever have to consider the users that rated that movie, which may be pretty large for popular movies but on average isn't; but even so, you're only ever inverting a $d \times d$ matrix.

Another cool thing: because each user is only dependent on their item's preferences, you can parallelize this update of U or V. You can run it on as many different machines as you want to make it fast.

Last Thoughts on These Algorithms

There are lots of different versions of this approach, but we hope we gave you a flavor for the trade-offs between these methods and how they can be used to make predictions. Sometimes you need to extend a method to make it work in your particular case.

For example, you can add new users, or new data, and keep optimizing U and V. You can choose which users you think need more updating to save computing time. Or if they have enough ratings, you can decide not to update the rest of them.

As with any machine learning model, you should perform cross-validation for this model—leave out a bit and see how you did, which we've discussed throughout the book. This is a way of testing overfitting problems.

Thought Experiment: Filter Bubbles

What are the implications of using error minimization to predict preferences? How does presentation of recommendations affect the feedback collected?

For example, can you end up in local maxima with rich-get-richer effects? In other words, does showing certain items at the beginning give them an unfair advantage over other things? And so do certain things just get popular or not based on luck?

How do you correct for this?

Exercise: Build Your Own Recommendation System

In Chapter 6, we did some exploratory data analysis on the GetGlue dataset. Now's your opportunity to build a recommendation system with that dataset. The following code isn't for GetGlue, but it is Matt's code to illustrate implementing a recommendation system on a relatively small dataset. Your challenge is to adjust it to work with the GetGlue data.

Sample Code in Python

```
import math,numpy

pu    =    [[(0,0,1),(0,1,22),(0,2,1),(0,3,1),(0,5,0)],[(1,0,1),
(1,1,32),(1,2,0),(1,3,0),(1,4,1),(1,5,0)],[(2,0,0),(2,1,18),
(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),(3,1,40),(3,2,1),
(3,3,0),(3,4,0),(3,5,1)],[(4,0,0),(4,1,40),(4,2,0),(4,4,1),
(4,5,0)],[(5,0,0),(5,1,25),(5,2,1),(5,3,1),(5,4,1)]]

pv     =     [[(0,0,1),(0,1,1),(0,2,0),(0,3,1),(0,4,0),(0,5,0)],
[(1,0,22),(1,1,32),(1,2,18),(1,3,40),(1,4,40),(1,5,25)],
[(2,0,1),(2,1,0),(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),
(3,1,0),(3,2,1),(3,3,0),(3,5,1)],[(4,1,1),(4,2,0),(4,3,0),
(4,4,1),(4,5,1)],[(5,0,0),(5,1,0),(5,2,1),(5,3,1),(5,4,0)]]

V = numpy.mat([[ 0.15968384,  0.9441198 ,  0.83651085],
               [ 0.73573009,  0.24906915,  0.85338239],
               [ 0.25605814,  0.6990532 ,  0.50900407],
               [ 0.2405843 ,  0.31848888,  0.60233653],
               [ 0.24237479,  0.15293281,  0.22240255],
               [ 0.03943766,  0.19287528,  0.95094265]])

print V

U = numpy.mat(numpy.zeros([6,3]))
L = 0.03

for iter in xrange(5):

    print "\n----- ITER %s -----"%(iter+1)

    print "U"
    urs = []
    for uset in pu:
        vo = []
        pvo = []
```

```
            for i,j,p in uset:
                vor = []
                for k in xrange(3):
                    vor.append(V[j,k])
                vo.append(vor)
                pvo.append(p)
            vo = numpy.mat(vo)
            ur = numpy.linalg.inv(vo.T*vo +
                L*numpy.mat(numpy.eye(3))) *
                vo.T * numpy.mat(pvo).T
            urs.append(ur.T)
        U = numpy.vstack(urs)
        print U

        print "V"
        vrs = []
        for vset in pv:
            uo = []
            puo = []
            for j,i,p in vset:
                uor = []
                for k in xrange(3):
                    uor.append(U[i,k])
                uo.append(uor)
                puo.append(p)
            uo = numpy.mat(uo)
                 vr  =  numpy.linalg.inv(uo.T*uo  +  L*numpy.mat(num
py.eye(3))) * uo.T * numpy.mat(puo).T
            vrs.append(vr.T)
        V = numpy.vstack(vrs)
        print V

        err = 0.
        n = 0.
        for uset in pu:
            for i,j,p in uset:
                err += (p - (U[i]*V[j].T)[0,0])**2
                n += 1
        print math.sqrt(err/n)

    print
    print U*V.T
```