Machine Learning

> I am always ready to learn although I do not always like being taught.
>
> Winston Churchill

Many people imagine that data science is mostly machine learning and that data scientists mostly build and train and tweak machine-learning models all day long. (Then again, many of those people don't actually know what machine learning *is*.) In fact, data science is mostly turning business problems into data problems and collecting data and understanding data and cleaning data and formatting data, after which machine learning is almost an afterthought. Even so, it's an interesting and essential afterthought that you pretty much have to know about in order to do data science.

Modeling

Before we can talk about machine learning we need to talk about *models*.

What is a model? It's simply a specification of a mathematical (or probabilistic) relationship that exists between different variables.

For instance, if you're trying to raise money for your social networking site, you might build a *business model* (likely in a spreadsheet) that takes inputs like "number of users" and "ad revenue per user" and "number of employees" and outputs your annual profit for the next several years. A cookbook recipe entails a model that relates inputs like "number of eaters" and "hungriness" to quantities of ingredients needed. And if you've ever watched poker on television, you know that they estimate each player's "win probability" in real time based on a model that takes into account the cards that have been revealed so far and the distribution of cards in the deck.

The business model is probably based on simple mathematical relationships: profit is revenue minus expenses, revenue is units sold times average price, and so on. The recipe model is probably based on trial and error — someone went in a kitchen and tried different combinations of ingredients until they found one they liked. And the poker model is based on probability theory, the rules of poker, and some reasonably innocuous assumptions about the random process by which cards are dealt.

What Is Machine Learning?

Everyone has her own exact definition, but we'll use *machine learning* to refer to creating and using models that are *learned from data*. In other contexts this might be called *predictive modeling* or *data mining*, but we will stick with machine learning. Typically, our goal will be to use existing data to develop models that we can use to *predict* various outcomes for new data, such as:

- Predicting whether an email message is spam or not
- Predicting whether a credit card transaction is fraudulent
- Predicting which advertisement a shopper is most likely to click on
- Predicting which football team is going to win the Super Bowl

We'll look at both *supervised* models (in which there is a set of data labeled with the correct answers to learn from), and *unsupervised* models (in which there are no such labels). There are various other types like *semisupervised* (in which only some of the data are labeled) and *online* (in which the model needs to continuously adjust to newly arriving data) that we won't cover in this book.

Now, in even the simplest situation there are entire universes of models that might describe the relationship we're interested in. In most cases we will ourselves choose a *parameterized* family of models and then use data to learn parameters that are in some way optimal.

For instance, we might assume that a person's height is (roughly) a linear function of his weight and then use data to learn what that linear function is. Or we might assume that a decision tree is a good way to diagnose what diseases our patients have and then use data to learn the "optimal" such tree. Throughout the rest of the book we'll be investigating different families of models that we can learn.

But before we can do that, we need to better understand the fundamentals of machine learning. For the rest of the chapter, we'll discuss some of those basic concepts, before we move on to the models themselves.

Overfitting and Underfitting

A common danger in machine learning is *overfitting* — producing a model that performs well on the data you train it on but that generalizes poorly to any new data. This could involve learning *noise* in the data. Or it could involve learning to identify specific inputs rather than whatever factors are actually predictive for the desired output.

The other side of this is *underfitting*, producing a model that doesn't perform well even on the training data, although typically when this happens you decide your model isn't good enough and keep looking for a better one.

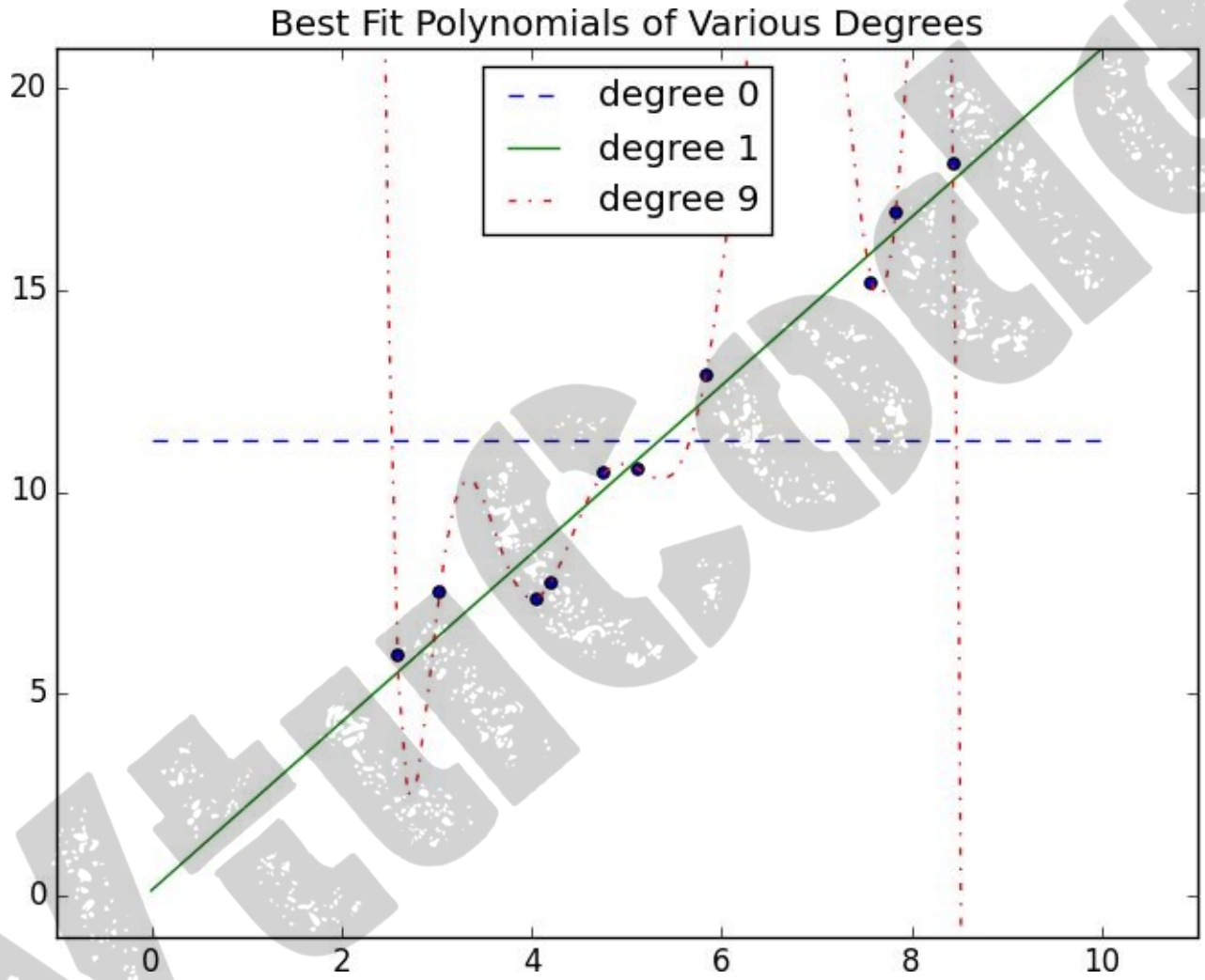

Figure 11-1. Overfitting and underfitting

In Figure 11-1, I've fit three polynomials to a sample of data. (Don't worry about how; we'll get to that in later chapters.)

The horizontal line shows the best fit degree 0 (i.e., constant) polynomial. It severely *underfits* the training data. The best fit degree 9 (i.e., 10-parameter) polynomial goes through every training data point exactly, but it very severely *overfits* — if we were to pick a few more data points it would quite likely miss them by a lot. And the degree 1 line strikes a nice balance — it's pretty close to every point, and (if these data are representative) the line will likely be close to new data points as well.

Clearly models that are too complex lead to overfitting and don't generalize well beyond

the data they were trained on. So how do we make sure our models aren't too complex? The most fundamental approach involves using different data to train the model and to test the model.

The simplest way to do this is to split your data set, so that (for example) two-thirds of it is used to train the model, after which we measure the model's performance on the remaining third:

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

Often, we'll have a matrix `x` of input variables and a vector `y` of output variables. In that case, we need to make sure to put corresponding values together in either the training data or the test data:

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                                # pair corresponding values
    train, test = split_data(data, 1 - test_pct)  # split the data set of pairs
    x_train, y_train = zip(*train)                  # magical un-zip trick
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

so that you might do something like:

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

If the model was overfit to the training data, then it will hopefully perform really poorly on the (completely separate) test data. Said differently, if it performs well on the test data, then you can be more confident that it's *fitting* rather than *overfitting*.

However, there are a couple of ways this can go wrong.

The first is if there are common patterns in the test and train data that wouldn't generalize to a larger data set.

For example, imagine that your data set consists of user activity, one row per user per week. In such a case, most users will appear in both the training data and the test data, and certain models might learn to *identify* users rather than discover relationships involving *attributes*. This isn't a huge worry, although it did happen to me once.

A bigger problem is if you use the test/train split not just to judge a model but also to *choose* from among many models. In that case, although each individual model may not be overfit, the "choose a model that performs best on the test set" is a meta-training that makes the test set function as a second training set. (Of course the model that performed best on the test set is going to perform well on the test set.)

In such a situation, you should split the data into three parts: a *training* set for building models, a *validation* set for choosing among trained models, and a *test* set for judging the final model.

Correctness

When I'm not doing data science, I dabble in medicine. And in my spare time I've come up with a cheap, noninvasive test that can be given to a newborn baby that predicts — with greater than 98% accuracy — whether the newborn will ever develop leukemia. My lawyer has convinced me the test is unpatentable, so I'll share with you the details here: predict leukemia if and only if the baby is named Luke (which sounds sort of like "leukemia").

As we'll see below, this test is indeed more than 98% accurate. Nonetheless, it's an incredibly stupid test, and a good illustration of why we don't typically use "accuracy" to measure how good a model is.

Imagine building a model to make a *binary* judgment. Is this email spam? Should we hire this candidate? Is this air traveler secretly a terrorist?

Given a set of labeled data and such a predictive model, every data point lies in one of four categories:

- True positive: "This message is spam, and we correctly predicted spam."
- False positive (Type 1 Error): "This message is not spam, but we predicted spam."
- False negative (Type 2 Error): "This message is spam, but we predicted not spam."
- True negative: "This message is not spam, and we correctly predicted not spam."

We often represent these as counts in a *confusion matrix*:

	Spam	**not Spam**
predict "Spam"	True Positive	False Positive
predict "Not Spam"	False Negative	True Negative

Let's see how my leukemia test fits into this framework. These days approximately 5 babies out of 1,000 are named Luke. And the lifetime prevalence of leukemia is about 1.4%, or 14 out of every 1,000 people.

If we believe these two factors are independent and apply my "Luke is for leukemia" test to 1 million people, we'd expect to see a confusion matrix like:

	leukemia	**no leukemia**	**total**
"Luke"	70	4,930	5,000
not "Luke"	13,930	981,070	995,000
total	14,000	986,000	1,000,000

We can then use these to compute various statistics about model performance. For example, *accuracy* is defined as the fraction of correct predictions:

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

That seems like a pretty impressive number. But clearly this is not a good test, which means that we probably shouldn't put a lot of credence in raw accuracy.

It's common to look at the combination of *precision* and *recall*. Precision measures how accurate our *positive* predictions were:

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

And recall measures what fraction of the positives our model identified:

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)       # 0.005
```

These are both terrible numbers, reflecting that this is a terrible model.

Sometimes precision and recall are combined into the *F1 score*, which is defined as:

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

This is the *harmonic mean* of precision and recall and necessarily lies between them.

Usually the choice of a model involves a trade-off between precision and recall. A model that predicts "yes" when it's even a little bit confident will probably have a high recall but a low precision; a model that predicts "yes" only when it's extremely confident is likely to have a low recall and a high precision.

Alternatively, you can think of this as a trade-off between false positives and false negatives. Saying "yes" too often will give you lots of false positives; saying "no" too often will give you lots of false negatives.

Imagine that there were 10 risk factors for leukemia, and that the more of them you had the more likely you were to develop leukemia. In that case you can imagine a continuum of tests: "predict leukemia if at least one risk factor," "predict leukemia if at least two risk factors," and so on. As you increase the threshhold, you increase the test's precision (since people with more risk factors are more likely to develop the disease), and you decrease the

test’s recall (since fewer and fewer of the eventual disease-sufferers will meet the threshhold). In cases like this, choosing the right threshhold is a matter of finding the right trade-off.

The Bias-Variance Trade-off

Another way of thinking about the overfitting problem is as a trade-off between bias and variance.

Both are measures of what would happen if you were to retrain your model many times on different sets of training data (from the same larger population).

For example, the degree 0 model in "Overfitting and Underfitting" will make a lot of mistakes for pretty much any training set (drawn from the same population), which means that it has a high *bias*. However, any two randomly chosen training sets should give pretty similar models (since any two randomly chosen training sets should have pretty similar average values). So we say that it has a low *variance*. High bias and low variance typically correspond to underfitting.

On the other hand, the degree 9 model fit the training set perfectly. It has very low bias but very high variance (since any two training sets would likely give rise to very different models). This corresponds to overfitting.

Thinking about model problems this way can help you figure out what do when your model doesn't work so well.

If your model has high bias (which means it performs poorly even on your training data) then one thing to try is adding more features. Going from the degree 0 model in "Overfitting and Underfitting" to the degree 1 model was a big improvement.

If your model has high variance, then you can similarly *remove* features. But another solution is to obtain more data (if you can).

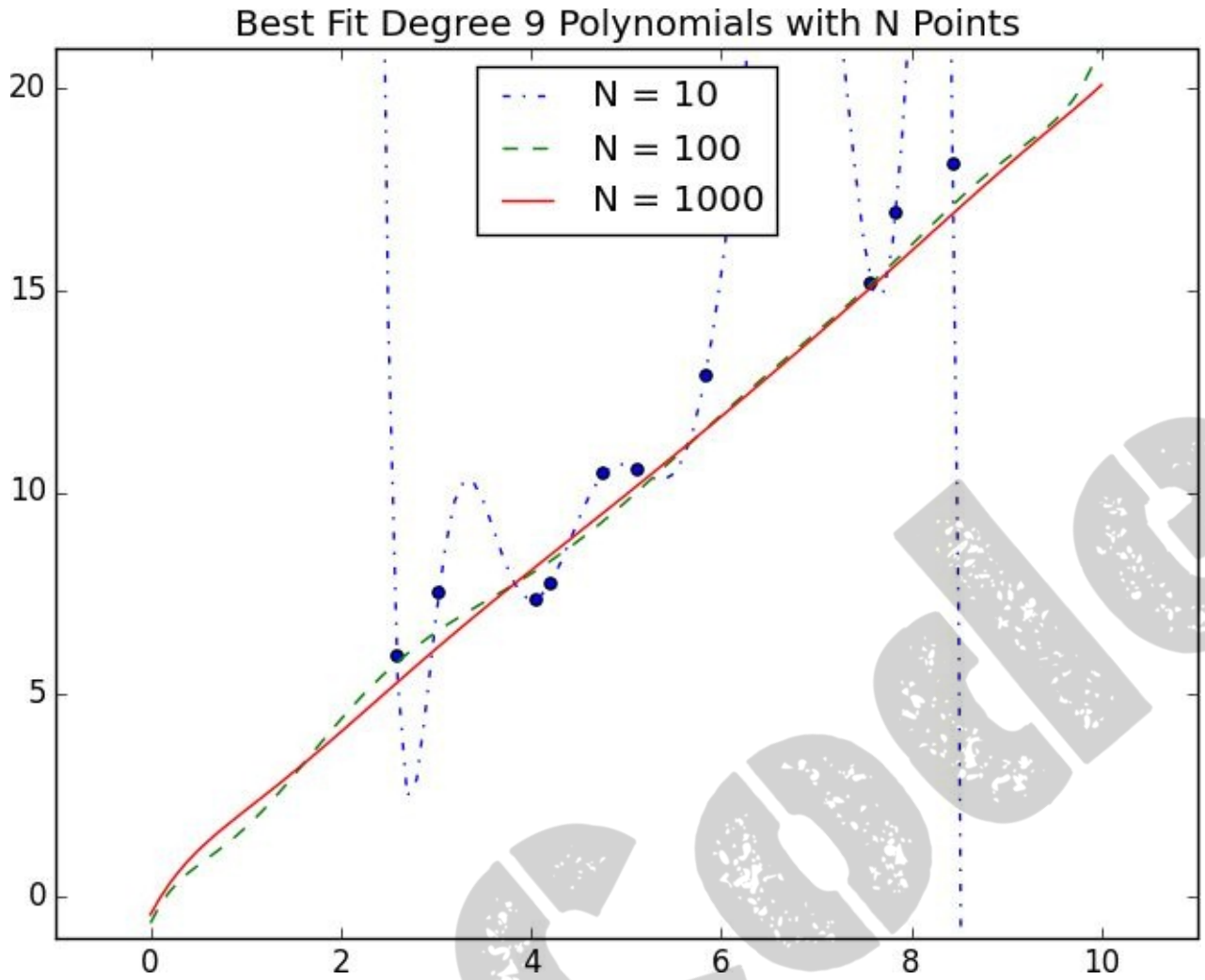

Figure 11-2. Reducing variance with more data

In Figure 11-2, we fit a degree 9 polynomial to different size samples. The model fit based on 10 data points is all over the place, as we saw before. If we instead trained on 100 data points, there's much less overfitting. And the model trained from 1,000 data points looks very similar to the degree 1 model.

Holding model complexity constant, the more data you have, the harder it is to overfit.

On the other hand, more data won't help with bias. If your model doesn't use enough features to capture regularities in the data, throwing more data at it won't help.

Feature Extraction and Selection

As we mentioned, when your data doesn't have enough features, your model is likely to underfit. And when your data has too many features, it's easy to overfit. But what are features and where do they come from?

Features are whatever inputs we provide to our model.

In the simplest case, features are simply given to you. If you want to predict someone's salary based on her years of experience, then years of experience is the only feature you have.

(Although, as we saw in "Overfitting and Underfitting", you might also consider adding years of experience squared, cubed, and so on if that helps you build a better model.)

Things become more interesting as your data becomes more complicated. Imagine trying to build a spam filter to predict whether an email is junk or not. Most models won't know what to do with a raw email, which is just a collection of text. You'll have to extract features. For example:

- Does the email contain the word "Viagra"?
- How many times does the letter d appear?
- What was the domain of the sender?

The first is simply a yes or no, which we typically encode as a 1 or 0. The second is a number. And the third is a choice from a discrete set of options.

Pretty much always, we'll extract features from our data that fall into one of these three categories. What's more, the type of features we have constrains the type of models we can use.

The Naive Bayes classifier we'll build in Chapter 13 is suited to yes-or-no features, like the first one in the preceding list.

Regression models, as we'll study in Chapter 14 and Chapter 16, require numeric features (which could include dummy variables that are 0s and 1s).

And decision trees, which we'll look at in Chapter 17, can deal with numeric or categorical data.

Although in the spam filter example we looked for ways to create features, sometimes we'll instead look for ways to remove features.

For example, your inputs might be vectors of several hundred numbers. Depending on the situation, it might be appropriate to distill these down to handful of important dimensions (as in "Dimensionality Reduction") and use only those small number of features. Or it might be appropriate to use a technique (like regularization, which we'll look at in "Regularization") that penalizes models the more features they use.

How do we choose features? That's where a combination of *experience* and *domain expertise* comes into play. If you've received lots of emails, then you probably have a sense that the presence of certain words might be a good indicator of spamminess. And you might also have a sense that the number of d's is likely not a good indicator of spamminess. But in general you'll have to try different things, which is part of the fun.

k-Nearest Neighbors

> If you want to annoy your neighbors, tell the truth about them.
>
> Pietro Aretino

Imagine that you're trying to predict how I'm going to vote in the next presidential election. If you know nothing else about me (and if you have the data), one sensible approach is to look at how my *neighbors* are planning to vote. Living in downtown Seattle, as I do, my neighbors are invariably planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.

Now imagine you know more about me than just geography — perhaps you know my age, my income, how many kids I have, and so on. To the extent my behavior is influenced (or characterized) by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

The Model

Nearest neighbors is one of the simplest predictive models there is. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:

- Some notion of distance
- An assumption that points that are close to one another are similar

Most of the techniques we'll look at in this book look at the data set as a whole in order to learn patterns in the data. Nearest neighbors, on the other hand, quite consciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

What's more, nearest neighbors is probably not going to help you understand the drivers of whatever phenomenon you're looking at. Predicting my votes based on my neighbors' votes doesn't tell you much about what causes me to vote the way I do, whereas some alternative model that predicted my vote based on (say) my income and marital status very well might.

In the general situation, we have some data points and we have a corresponding set of labels. The labels could be `True` and `False`, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?" Or they could be categories, like movie ratings (G, PG, PG-13, R, NC-17). Or they could be the names of presidential candidates. Or they could be favorite programming languages.

In our case, the data points will be vectors, which means that we can use the `distance` function from Chapter 4.

Let's say we've picked a number *k* like 3 or 5. Then when we want to classify some new data point, we find the `k` nearest labeled points and let them vote on the new output.

To do this, we'll need a function that counts votes. One possibility is:

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

But this doesn't do anything intelligent with ties. For example, imagine we're rating movies and the five nearest movies are rated G, G, PG, PG, and R. Then G has two votes and PG also has two votes. In that case, we have several options:

- Pick one of the winners at random.
- Weight the votes by distance and pick the weighted winner.
- Reduce *k* until we find a unique winner.

We'll implement the third:

```
def majority_vote(labels):
    """assumes that labels are ordered from nearest to farthest"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again without the farthest
```

This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

With this function it's easy to create a classifier:

```
def knn_classify(k, labeled_points, new_point):
    """each labeled point should be a pair (point, label)"""

    # order the labeled points from nearest to farthest
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # find the labels for the k closest
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # and let them vote
    return majority_vote(k_nearest_labels)
```

Let's take a look at how this works.

Example: Favorite Languages

The results of the first DataSciencester user survey are back, and we've found the preferred programming languages of our users in a number of large cities:

```
# each entry is ([longitude, latitude], favorite_language)

cities = [([-122.3 , 47.53], "Python"),  # Seattle
          ([ -96.85, 32.85], "Java"),    # Austin
          ([ -89.33, 43.13], "R"),       # Madison
          # ... and so on
]
```

The VP of Community Engagement wants to know if we can use these results to predict the favorite programming languages for places that weren't part of our survey.

As usual, a good first step is plotting the data (Figure 12-1):

```
# key is language, value is pair (longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# we want each language to have a different marker and color
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# create a scatter series for each language
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                      label=language, zorder=10)

plot_state_borders(plt)      # pretend we have a function that does this

plt.legend(loc=0)            # let matplotlib choose the location
plt.axis([-130,-60,20,55])   # set the axes

plt.title("Favorite Programming Languages")
plt.show()
```

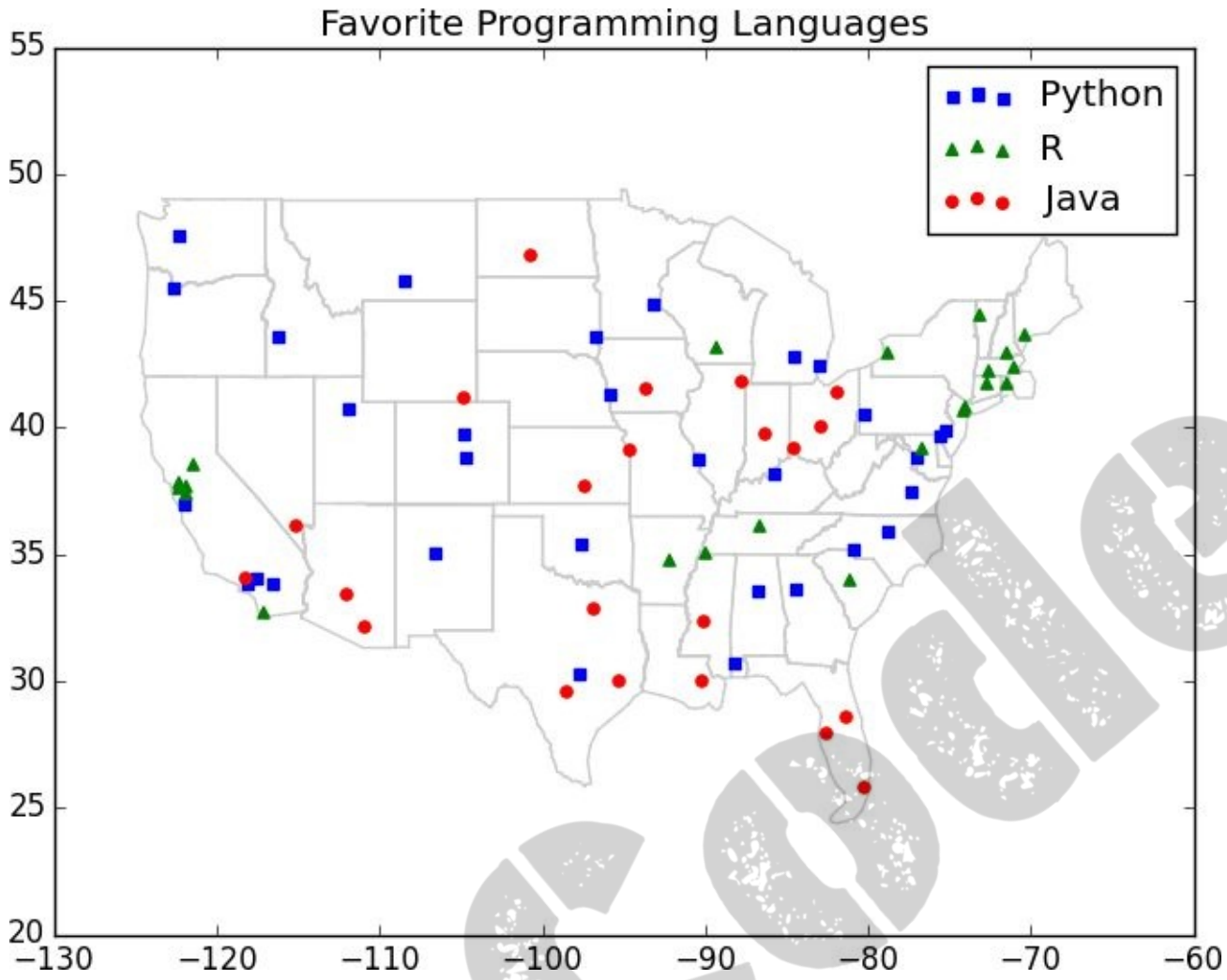

Figure 12-1. Favorite programming languages

> **NOTE**
>
> You may have noticed the call to `plot_state_borders()`, a function that we haven't actually defined. There's an implementation on the book's GitHub page, but it's a good exercise to try to do it yourself:
>
> 1. Search the Web for something like *state boundaries latitude longitude*.
> 2. Convert whatever data you can find into a list of segments [(long1, lat1), (long2, lat2)].
> 3. Use `plt.plot()` to draw the segments.

Since it looks like nearby places tend to like the same language, *k*-nearest neighbors seems like a reasonable choice for a predictive model.

To start with, let's look at what happens if we try to predict each city's preferred language using its neighbors other than itself:

```
# try several different values for k
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)
```

```
        if predicted_language == actual_language:
            num_correct += 1

    print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

It looks like 3-nearest neighbors performs the best, giving the correct result about 59% of the time:

```
1 neighbor[s]: 40 correct out of 75
3 neighbor[s]: 44 correct out of 75
5 neighbor[s]: 41 correct out of 75
7 neighbor[s]: 35 correct out of 75
```

Now we can look at what regions would get classified to which languages under each nearest neighbors scheme. We can do that by classifying an entire grid worth of points, and then plotting them as we did the cities:

```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # or 3, or 5, or…

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

For instance, Figure 12-2 shows what happens when we look at just the nearest neighbor ($k = 1$).

We see lots of abrupt changes from one language to another with sharp boundaries. As we increase the number of neighbors to three, we see smoother regions for each language (Figure 12-3).

And as we increase the neighbors to five, the boundaries get smoother still (Figure 12-4).

Here our dimensions are roughly comparable, but if they weren't you might want to rescale the data as we did in "Rescaling".

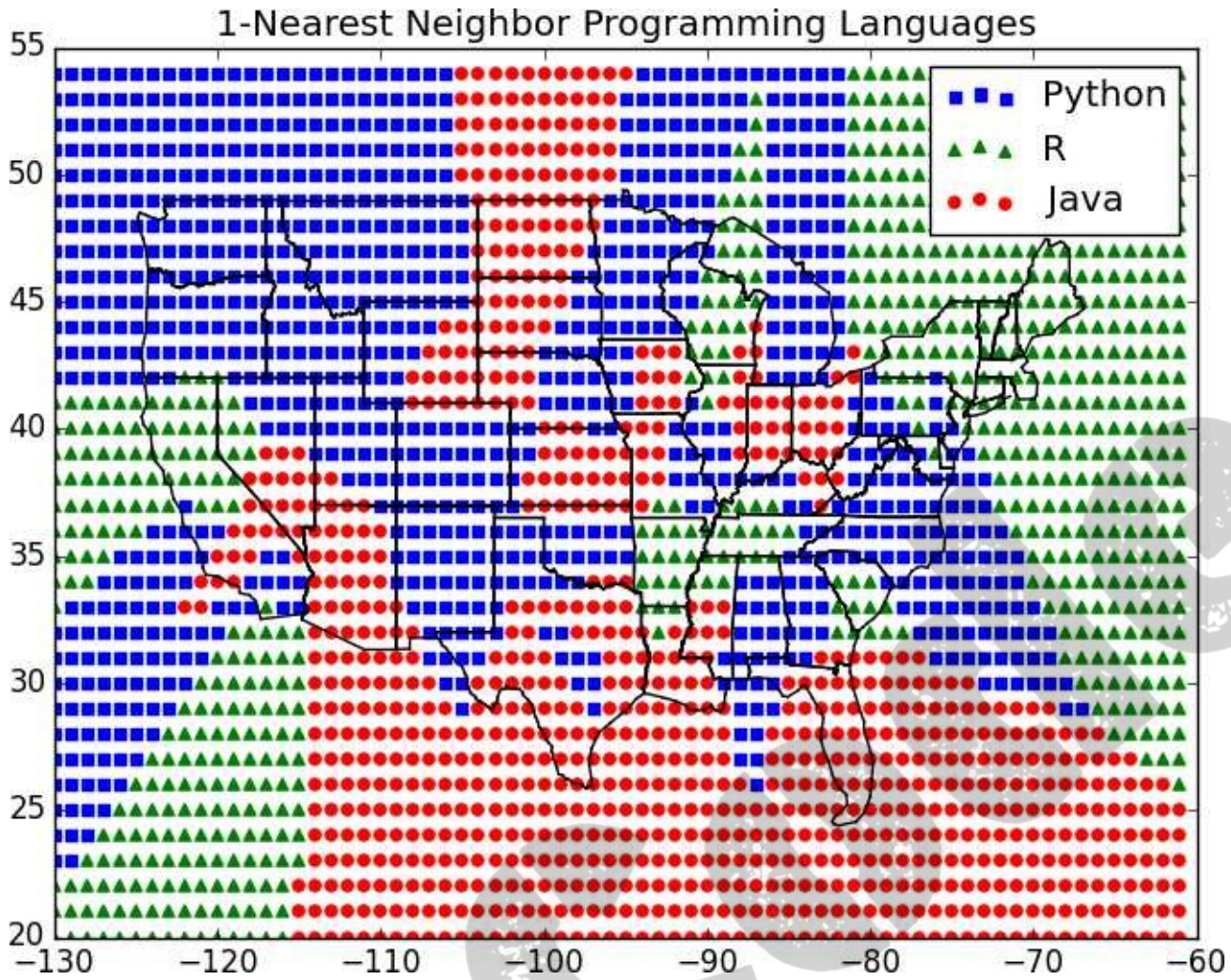

Figure 12-2. 1-Nearest neighbor programming languages

The Curse of Dimensionality

k-nearest neighbors runs into trouble in higher dimensions thanks to the "curse of dimensionality," which boils down to the fact that high-dimensional spaces are *vast*. Points in high-dimensional spaces tend not to be close to one another at all. One way to see this is by randomly generating pairs of points in the d-dimensional "unit cube" in a variety of dimensions, and calculating the distances between them.

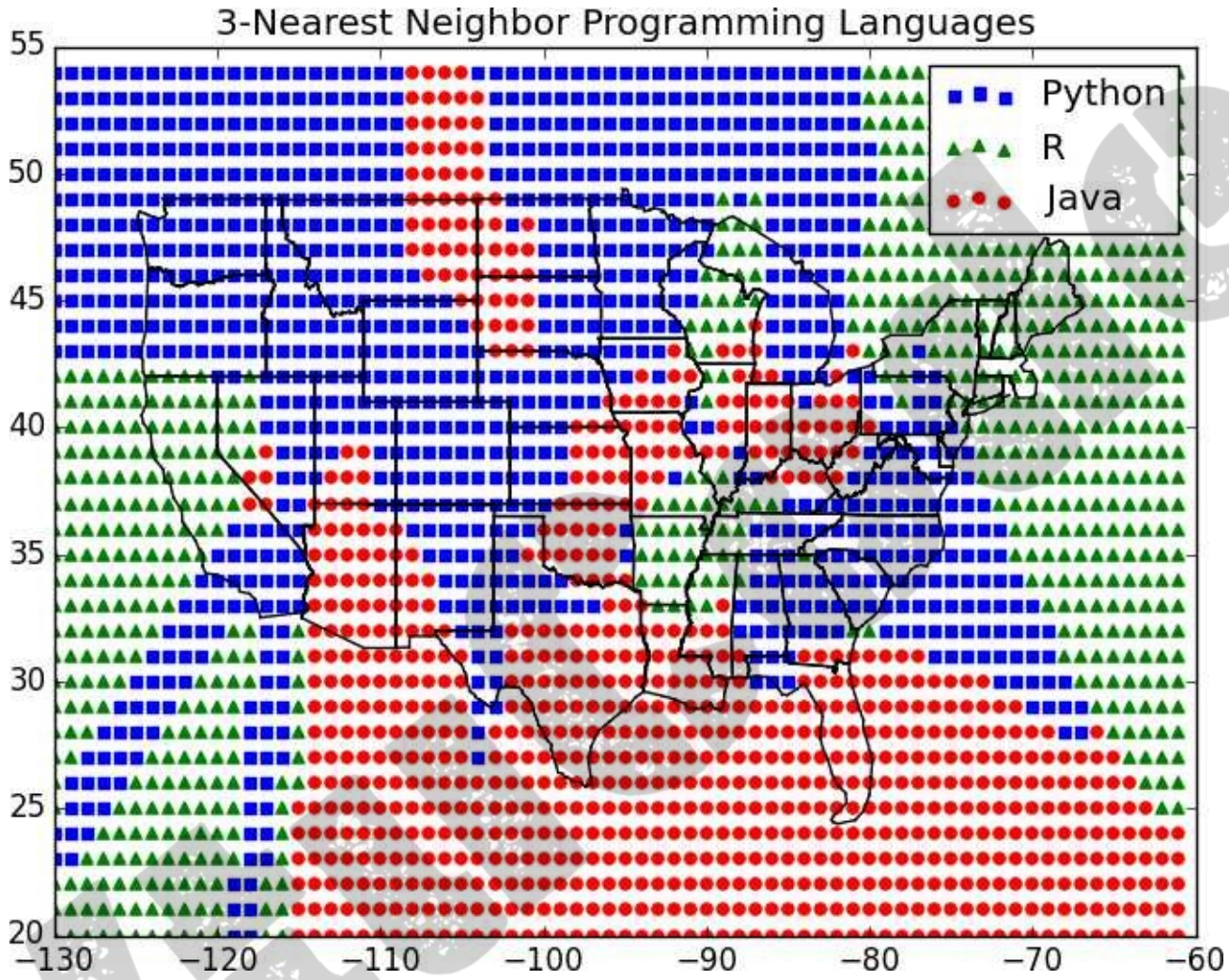

Figure 12-3. 3-Nearest neighbor programming languages

Generating random points should be second nature by now:

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

as is writing a function to generate the distances:

```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

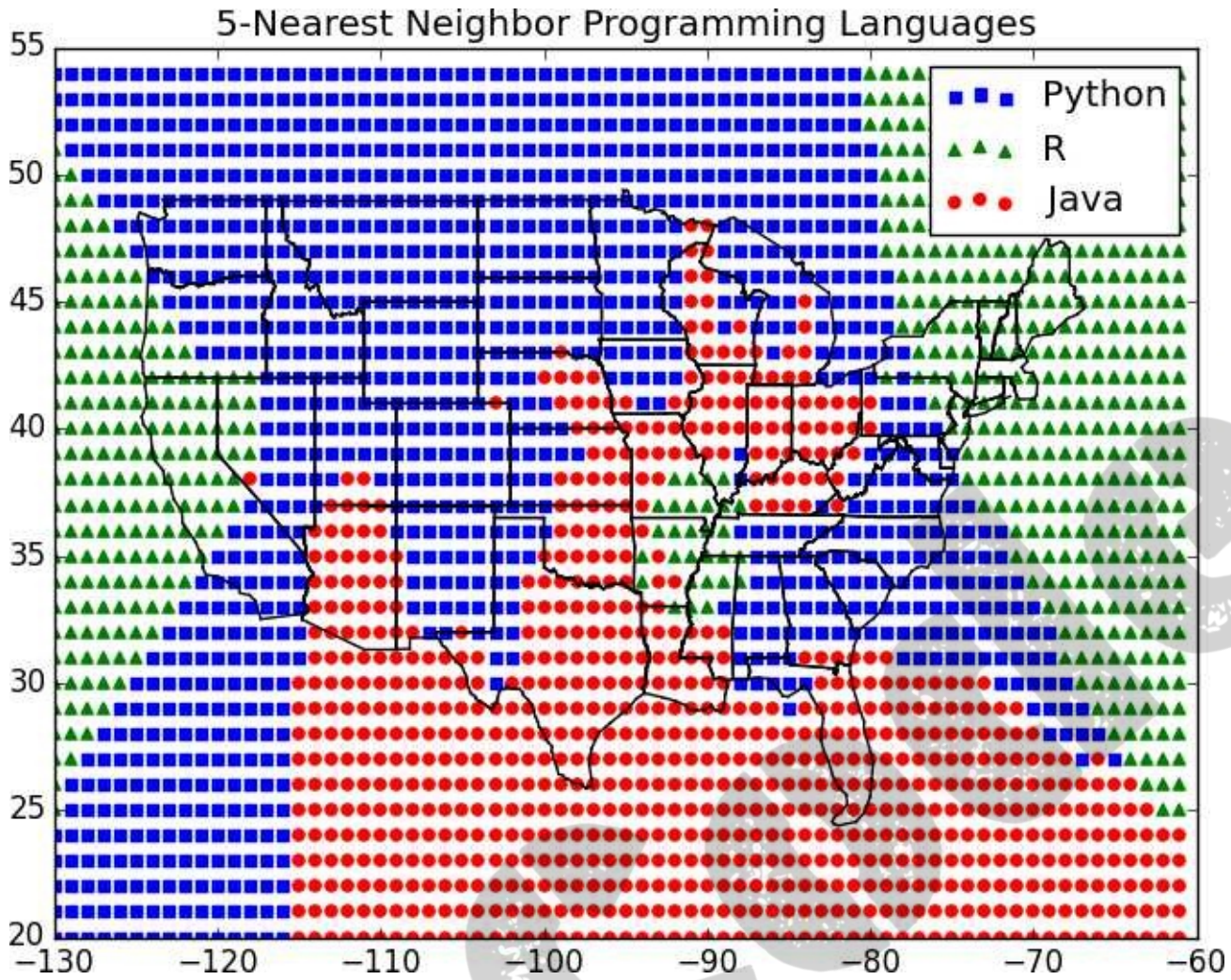

Figure 12-4. 5-Nearest neighbor programming languages

For every dimension from 1 to 100, we'll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension (Figure 12-5):

```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
  distances = random_distances(dim, 10000)  # 10,000 random pairs
  avg_distances.append(mean(distances))     # track the average
  min_distances.append(min(distances))      # track the minimum
```

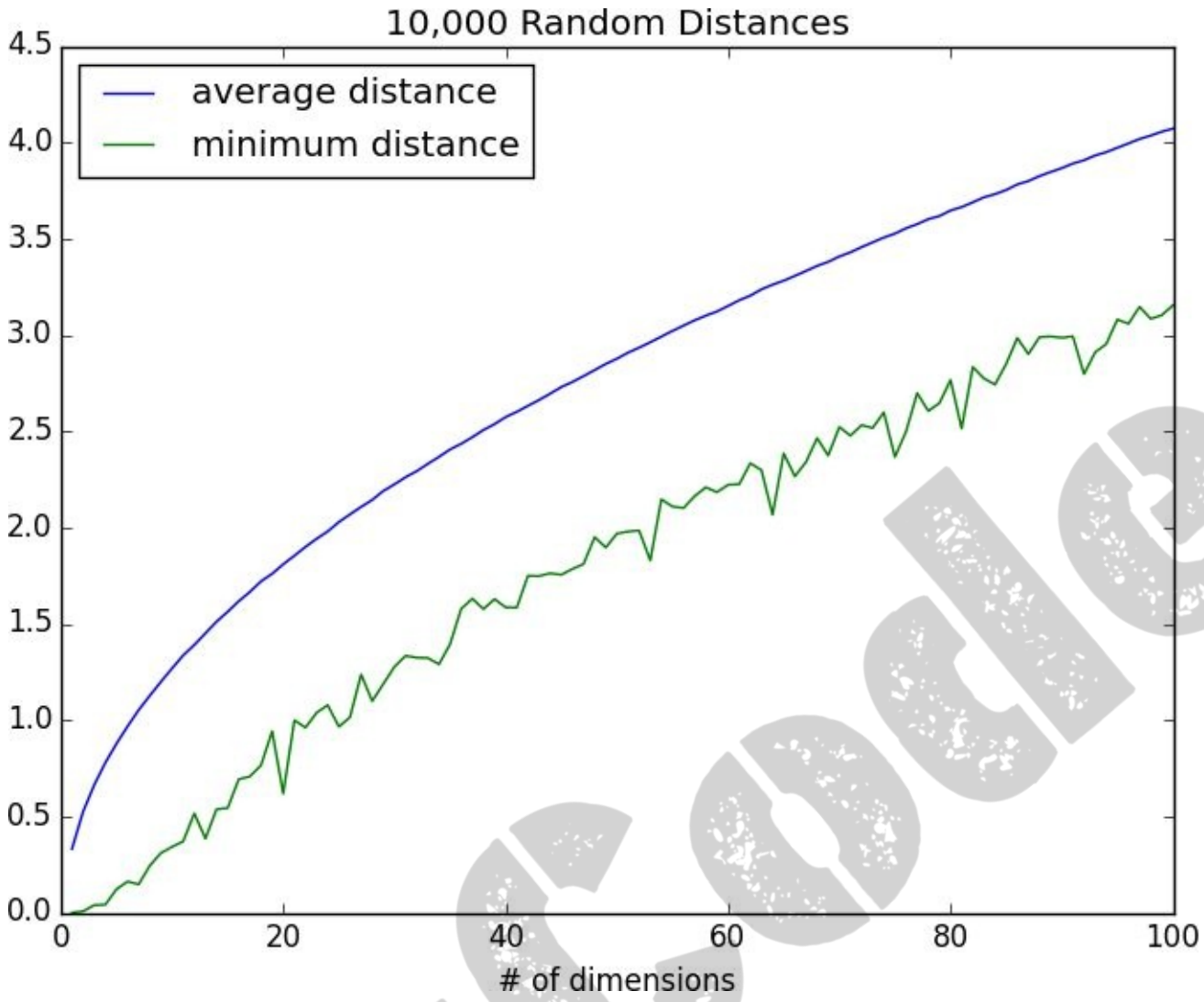

Figure 12-5. The curse of dimensionality

As the number of dimensions increases, the average distance between points increases. But what's more problematic is the ratio between the closest distance and the average distance (Figure 12-6):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

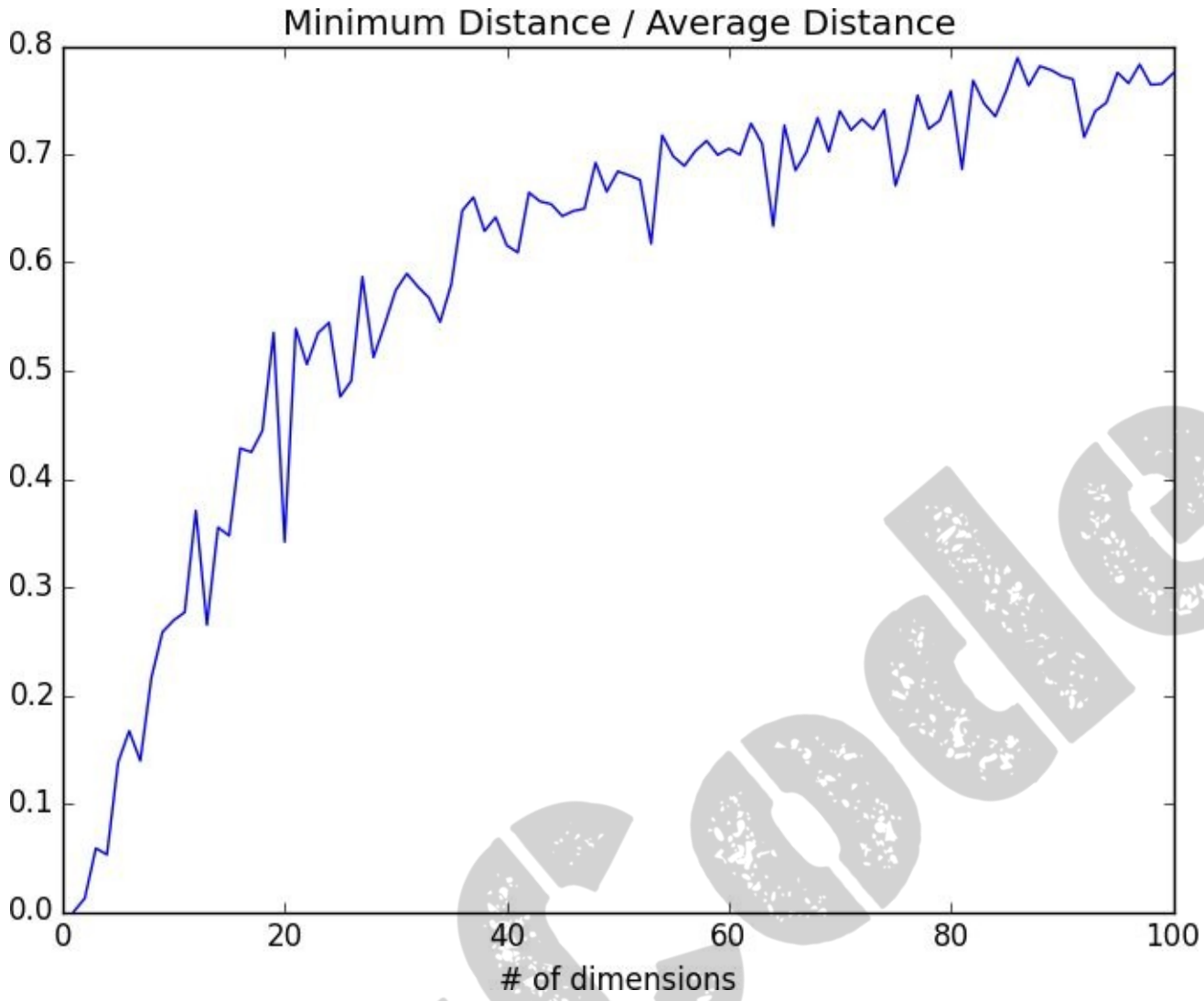

Figure 12-6. The curse of dimensionality again

In low-dimensional data sets, the closest points tend to be much closer than average. But two points are close only if they're close in every dimension, and every extra dimension — even if just noise — is another opportunity for each point to be further away from every other point. When you have a lot of dimensions, it's likely that the closest points aren't much closer than average, which means that two points being close doesn't mean very much (unless there's a lot of structure in your data that makes it behave as if it were much lower-dimensional).

A different way of thinking about the problem involves the sparsity of higher-dimensional spaces.

If you pick 50 random numbers between 0 and 1, you'll probably get a pretty good sample of the unit interval (Figure 12-7).

Figure 12-7. Fifty random points in one dimension

If you pick 50 random points in the unit square, you'll get less coverage (Figure 12-8).

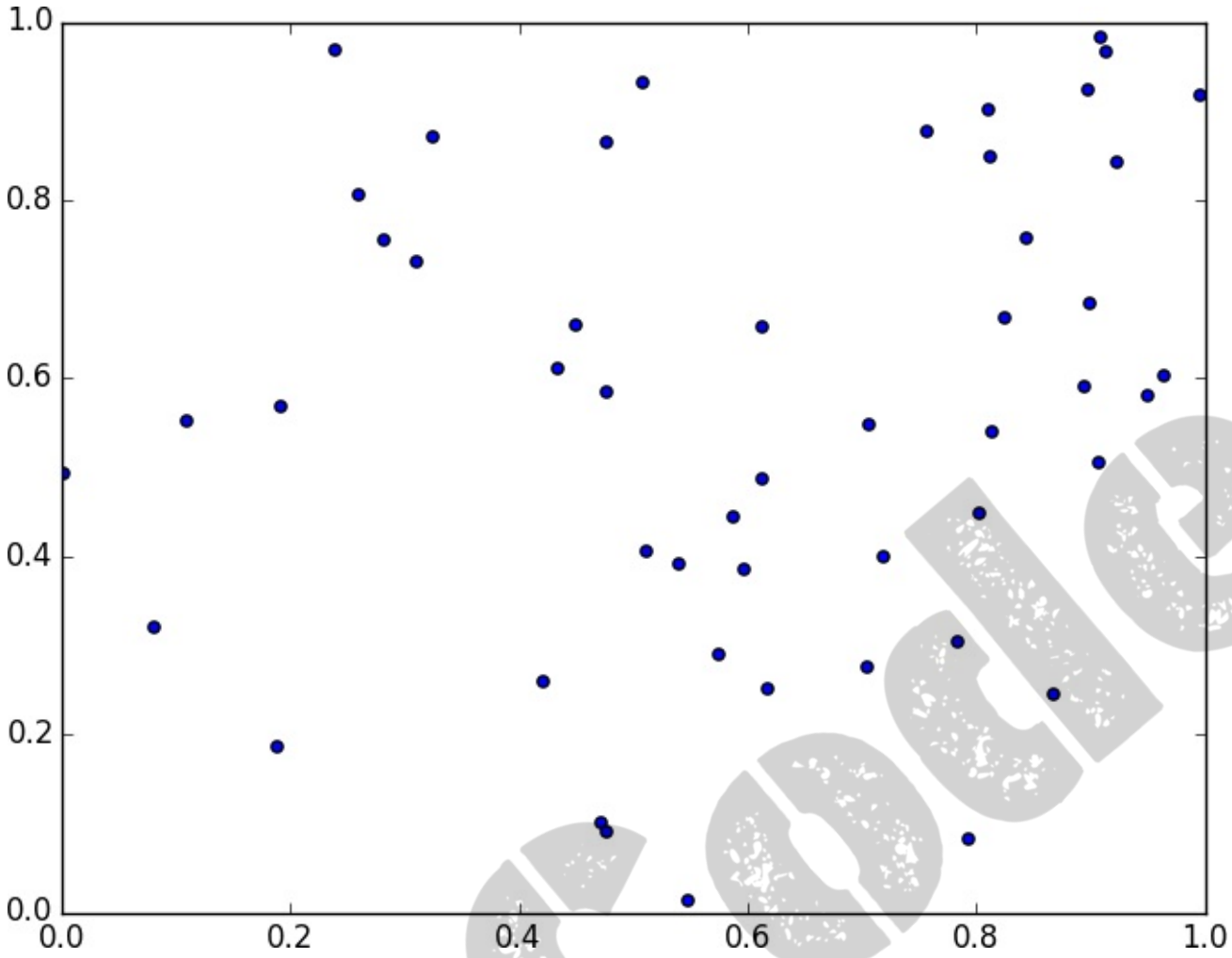

Figure 12-8. Fifty random points in two dimensions

And in three dimensions less still (Figure 12-9).

`matplotlib` doesn't graph four dimensions well, so that's as far as we'll go, but you can see already that there are starting to be large empty spaces with no points near them. In more dimensions — unless you get exponentially more data — those large empty spaces represent regions far from all the points you want to use in your predictions.

So if you're trying to use nearest neighbors in higher dimensions, it's probably a good idea to do some kind of dimensionality reduction first.

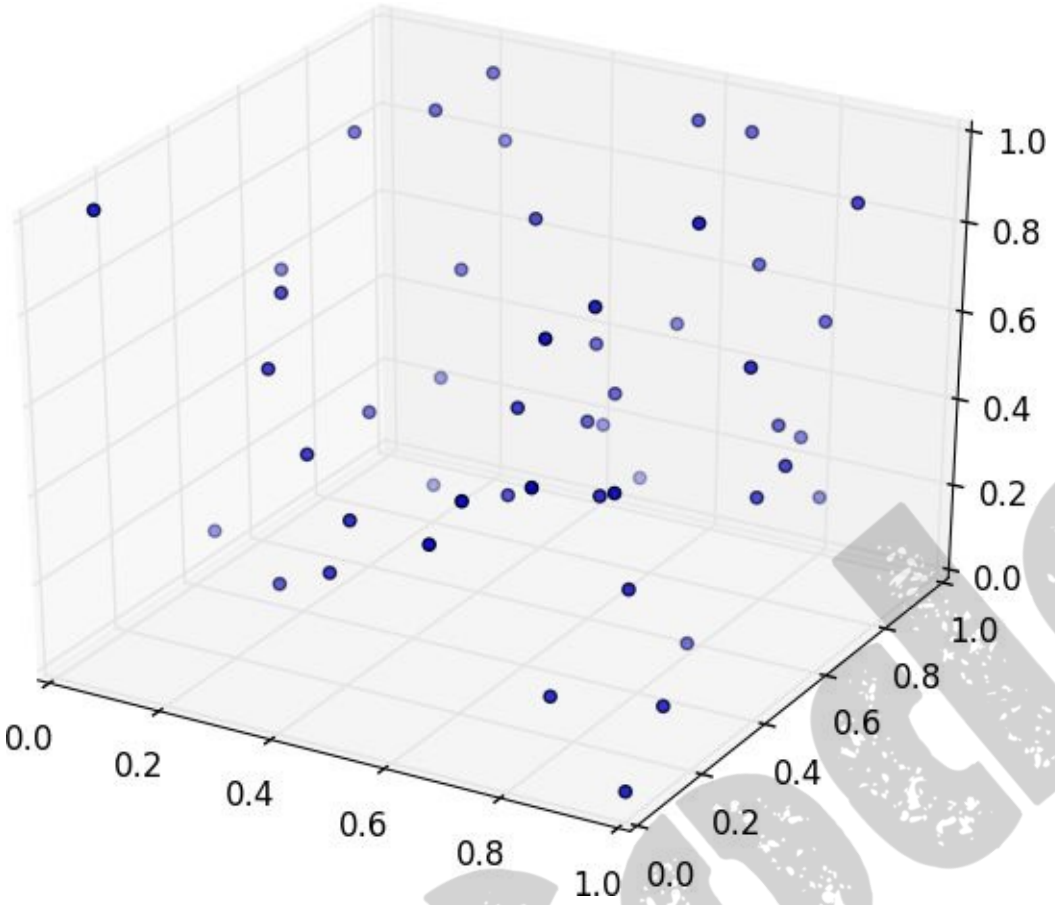

Figure 12-9. Fifty random points in three dimensions

Naive Bayes

> It is well for the heart to be naive and for the mind not to be.
>
> Anatole France

A social network isn't much good if people can't network. Accordingly, DataSciencester has a popular feature that allows members to send messages to other members. And while most of your members are responsible citizens who send only well-received "how's it going?" messages, a few miscreants persistently spam other members about get-rich schemes, no-prescription-required pharmaceuticals, and for-profit data science credentialing programs. Your users have begun to complain, and so the VP of Messaging has asked you to use data science to figure out a way to filter out these spam messages.

A Really Dumb Spam Filter

Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let *S* be the event "the message is spam" and *V* be the event "the message contains the word *viagra*." Then Bayes's Theorem tells us that the probability that the message is spam conditional on containing the word *viagra* is:

$$P(S \mid V) = [P(V \mid S)P(S)] / [P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)]$$

The numerator is the probability that a message is spam *and* contains *viagra*, while the denominator is just the probability that a message contains *viagra*. Hence you can think of this calculation as simply representing the proportion of *viagra* messages that are spam.

If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate $P(V \mid S)$ and $P(V \mid \neg S)$. If we further assume that any message is equally likely to be spam or not-spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S \mid V) = P(V \mid S) / [P(V \mid S) + P(V \mid \neg S)]$$

For example, if 50% of spam messages have the word *viagra*, but only 1% of nonspam messages do, then the probability that any given *viagra*-containing email is spam is:

$$0.5 / (0.5 + 0.01) = 98\ \%$$

A More Sophisticated Spam Filter

Imagine now that we have a vocabulary of many words $w_1, \ldots, w_n$. To move this into the realm of probability theory, we'll write X_i for the event "a message contains the word w_i." Also imagine that (through some unspecified-at-this-point process) we've come up with an estimate $P(X_i \mid S)$ for the probability that a spam message contains the *i*th word, and a similar estimate $P(X_i \mid \neg S)$ for the probability that a nonspam message contains the *i*th word.

The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not. Intuitively, this assumption means that knowing whether a certain spam message contains the word "viagra" gives you no information about whether that same message contains the word "rolex." In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n \mid S) = P(X_1 = x_1 \mid S) \times \cdots \times P(X_n = x_n \mid S)$$

This is an extreme assumption. (There's a reason the technique has "naive" in its name.) Imagine that our vocabulary consists *only* of the words "viagra" and "rolex," and that half of all spam messages are for "cheap viagra" and that the other half are for "authentic rolex." In this case, the Naive Bayes estimate that a spam message contains both "viagra" and "rolex" is:

$$P(X_1 = 1, X_2 = 1 \mid S) = P(X_1 = 1 \mid S)P(X_2 = 1 \mid S) = .5 \times .5 = .25$$

since we've assumed away the knowledge that "viagra" and "rolex" actually never occur together. Despite the unrealisticness of this assumption, this model often performs well and is used in actual spam filters.

The same Bayes's Theorem reasoning we used for our "viagra-only" spam filter tells us that we can calculate the probability a message is spam using the equation:

$$P(S \mid X = x) = P(X = x \mid S) / [P(X = x \mid S) + P(X = x \mid \neg S)]$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to avoid a problem called *underflow*, in which computers don't deal well with floating-point numbers that are too close to zero. Recalling from algebra that $\log(ab) = \log a + \log b$ and that $\exp(\log x) = x$, we usually compute $p_1 * \cdots * p_n$ as the equivalent (but floating-point-friendlier):

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

The only challenge left is coming up with estimates for $P(X_i \mid S)$ and $P(X_i \mid \neg S)$, the probabilities that a spam message (or nonspam message) contains the word w_i. If we have a fair number of "training" messages labeled as spam and not-spam, an obvious first try is to estimate $P(X_i \mid S)$ simply as the fraction of spam messages containing word w_i.

This causes a big problem, though. Imagine that in our training set the vocabulary word "data" only occurs in nonspam messages. Then we'd estimate $P(\text{"data"} \mid S) = 0$. The result is that our Naive Bayes classifier would always assign spam probability 0 to *any* message containing the word "data," even a message like "data on cheap viagra and authentic rolex watches." To avoid this problem, we usually use some kind of smoothing.

In particular, we'll choose a *pseudocount* — *k* — and estimate the probability of seeing the *i*th word in a spam as:

$$P(X_i \mid S) = (k + \text{number of spams containing } w_i) / (2k + \text{number of spams})$$

Similarly for $P(X_i \mid \neg S)$. That is, when computing the spam probabilities for the *i*th word, we assume we also saw *k* additional spams containing the word and *k* additional spams not containing the word.

For example, if "data" occurs in 0/98 spam documents, and if *k* is 1, we estimate $P(\text{"data"} \mid S)$ as 1/100 = 0.01, which allows our classifier to still assign some nonzero spam probability to messages that contain the word "data."

Implementation

Now we have all the pieces we need to build our classifier. First, let's create a simple function to tokenize messages into distinct words. We'll first convert each message to lowercase; use `re.findall()` to extract "words" consisting of letters, numbers, and apostrophes; and finally use `set()` to get just the distinct words:

```
def tokenize(message):
    message = message.lower()                       # convert to lowercase
    all_words = re.findall("[a-z0-9']+", message)   # extract the words
    return set(all_words)                           # remove duplicates
```

Our second function will count the words in a labeled training set of messages. We'll have it return a dictionary whose keys are words, and whose values are two-element lists `[spam_count, non_spam_count]` corresponding to how many times we saw that word in both spam and nonspam messages:

```
def count_words(training_set):
    """training set consists of pairs (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

Our next step is to turn these counts into estimated probabilities using the smoothing we described before. Our function will return a list of triplets containing each word, the probability of seeing that word in a spam message, and the probability of seeing that word in a nonspam message:

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """turn the word_counts into a list of triplets
    w, p(w | spam) and p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

The last piece is to use these word probabilities (and our Naive Bayes assumptions) to assign probabilities to messages:

```
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # iterate through each word in our vocabulary
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # if *word* appears in the message,
        # add the log probability of seeing it
        if word in message_words:
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_not_spam += math.log(prob_if_not_spam)

        # if *word* doesn't appear in the message
        # add the log probability of _not_ seeing it
        # which is log(1 - probability of seeing it)
        else:
```

```
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

We can put this all together into our Naive Bayes Classifier:

```
class NaiveBayesClassifier:

    def __init__(self, k=0.5):
        self.k = k
        self.word_probs = []

    def train(self, training_set):

        # count spam and non-spam messages
        num_spams = len([is_spam
                         for message, is_spam in training_set
                         if is_spam])
        num_non_spams = len(training_set) - num_spams

        # run training data through our "pipeline"
        word_counts = count_words(training_set)
        self.word_probs = word_probabilities(word_counts,
                                             num_spams,
                                             num_non_spams,
                                             self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

Testing Our Model

A good (if somewhat old) data set is the SpamAssassin public corpus. We'll look at the files prefixed with *20021010*. (On Windows, you might need a program like 7-Zip to decompress and extract them.)

After extracting the data (to, say, *C:\spam*) you should have three folders: *spam*, *easy_ham*, and *hard_ham*. Each folder contains many emails, each contained in a single file. To keep things *really* simple, we'll just look at the subject lines of each email.

How do we identify the subject line? Looking through the files, they all seem to start with "Subject:". So we'll look for that:

```
import glob, re

# modify the path with wherever you've put the files
path = r"C:\spam\*\*"

data = []

# glob.glob returns every filename that matches the wildcarded path
for fn in glob.glob(path):
    is_spam = "ham" not in fn

    with open(fn,'r') as file:
        for line in file:
            if line.startswith("Subject:"):
                # remove the leading "Subject: " and keep what's left
                subject = re.sub(r"^Subject: ", "", line).strip()
                data.append((subject, is_spam))
```

Now we can split the data into training data and test data, and then we're ready to build a classifier:

```
random.seed(0)      # just so you get the same answers as me
train_data, test_data = split_data(data, 0.75)

classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

And then we can check how our model does:

```
# triplets (subject, actual is_spam, predicted spam probability)
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]

# assume that spam_probability > 0.5 corresponds to spam prediction
# and count the combinations of (actual is_spam, predicted is_spam)
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

This gives 101 true positives (spam classified as "spam"), 33 false positives (ham classified as "spam"), 704 true negatives (ham classified as "ham"), and 38 false negatives (spam classified as "ham"). This means our precision is 101 / (101 + 33) = 75%, and our recall is 101 / (101 + 38) = 73%, which are not bad numbers for such a simple model.

It's also interesting to look at the most misclassified:

```
# sort by spam_probability from smallest to largest
classified.sort(key=lambda row: row[2])

# the highest predicted spam probabilities among the non-spams
spammiest_hams = filter(lambda row: not row[1], classified)[-5:]

# the lowest predicted spam probabilities among the actual spams
hammiest_spams = filter(lambda row: row[1], classified)[:5]
```

The two spammiest hams both have the words “needed” (77 times more likely to appear in spam), “insurance” (30 times more likely to appear in spam), and “important” (10 times more likely to appear in spam).

The hammiest spam is too short (“Re: girls”) to make much of a judgment, and the second-hammiest is a credit card solicitation most of whose words weren’t in the training set.

We can similarly look at the spammiest *words*:

```
def p_spam_given_word(word_prob):
    """uses bayes's theorem to compute p(spam | message contains word)"""

    # word_prob is one of the triplets produced by word_probabilities
    word, prob_if_spam, prob_if_not_spam = word_prob
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)

words = sorted(classifier.word_probs, key=p_spam_given_word)

spammiest_words = words[-5:]
hammiest_words = words[:5]
```

The spammiest words are “money,” “systemworks,” “rates,” “sale,” and “year,” all of which seem related to trying to get people to buy things. And the hammiest words are “spambayes,” “users,” “razor,” “zzzzteana,” and “sadev,” most of which seem related to spam prevention, oddly enough.

How could we get better performance? One obvious way would be to get more data to train on. There are a number of ways to improve the model as well. Here are some possibilities that you might try:

- Look at the message content, not just the subject line. You’ll have to be careful how you deal with the message headers.

- Our classifier takes into account every word that appears in the training set, even words that appear only once. Modify the classifier to accept an optional `min_count` threshhold and ignore tokens that don’t appear at least that many times.

- The tokenizer has no notion of similar words (e.g., “cheap” and “cheapest”). Modify the classifier to take an optional `stemmer` function that converts words to *equivalence classes* of words. For example, a really simple stemmer function might be:

```
def drop_final_s(word):
    return re.sub("s$", "", word)
```

Creating a good stemmer function is hard. People frequently use the Porter Stemmer.

- Although our features are all of the form "message contains word w_i," there's no reason why this has to be the case. In our implementation, we could add extra features like "message contains a number" by creating phony tokens like *contains:number* and modifying the `tokenizer` to emit them when appropriate.

Simple Linear Regression

> Art, like morality, consists in drawing the line somewhere.
>
> G. K. Chesterton

In Chapter 5, we used the `correlation` function to measure the strength of the linear relationship between two variables. For most applications, knowing that such a linear relationship exists isn't enough. We'll want to be able to understand the nature of the relationship. This is where we'll use simple linear regression.

The Model

Recall that we were investigating the relationship between a DataSciencester user's number of friends and the amount of time he spent on the site each day. Let's assume that you've convinced yourself that having more friends *causes* people to spend more time on the site, rather than one of the alternative explanations we discussed.

The VP of Engagement asks you to build a model describing this relationship. Since you found a pretty strong linear relationship, a natural place to start is a linear model.

In particular, you hypothesize that there are constants α (alpha) and β (beta) such that:

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

where y_i is the number of minutes user *i* spends on the site daily, x_i is the number of friends user *i* has, and ε_i is a (hopefully small) error term representing the fact that there are other factors not accounted for by this simple model.

Assuming we've determined such an `alpha` and `beta`, then we make predictions simply with:

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

How do we choose `alpha` and `beta`? Well, any choice of `alpha` and `beta` gives us a predicted output for each input `x_i`. Since we know the actual output `y_i` we can compute the error for each pair:

```
def error(alpha, beta, x_i, y_i):
    """the error from predicting beta * x_i + alpha
    when the actual value is y_i"""
    return y_i - predict(alpha, beta, x_i)
```

What we'd really like to know is the total error over the entire data set. But we don't want to just add the errors — if the prediction for `x_1` is too high and the prediction for `x_2` is too low, the errors may just cancel out.

So instead we add up the *squared* errors:

```
def sum_of_squared_errors(alpha, beta, x, y):
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

The *least squares solution* is to choose the `alpha` and `beta` that make `sum_of_squared_errors` as small as possible.

Using calculus (or tedious algebra), the error-minimizing alpha and beta are given by:

```
def least_squares_fit(x, y):
```

```
    """given training values for x and y,
    find the least-squares values of alpha and beta"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

Without going through the exact mathematics, let's think about why this might be a reasonable solution. The choice of `alpha` simply says that when we see the average value of the independent variable `x`, we predict the average value of the dependent variable `y`.

The choice of `beta` means that when the input value increases by `standard_deviation(x)`, the prediction increases by `correlation(x, y) * standard_deviation(y)`. In the case when `x` and `y` are perfectly correlated, a one standard deviation increase in `x` results in a one-standard-deviation-of-`y` increase in the prediction. When they're perfectly anticorrelated, the increase in `x` results in a *decrease* in the prediction. And when the correlation is zero, `beta` is zero, which means that changes in `x` don't affect the prediction at all.

It's easy to apply this to the outlierless data from Chapter 5:

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

This gives values of alpha = 22.95 and beta = 0.903. So our model says that we expect a user with `n` friends to spend `22.95 + n * 0.903` minutes on the site each day. That is, we predict that a user with no friends on DataSciencester would still spend about 23 minutes a day on the site. And for each additional friend, we expect a user to spend almost a minute more on the site each day.

In Figure 14-1, we plot the prediction line to get a sense of how well the model fits the observed data.

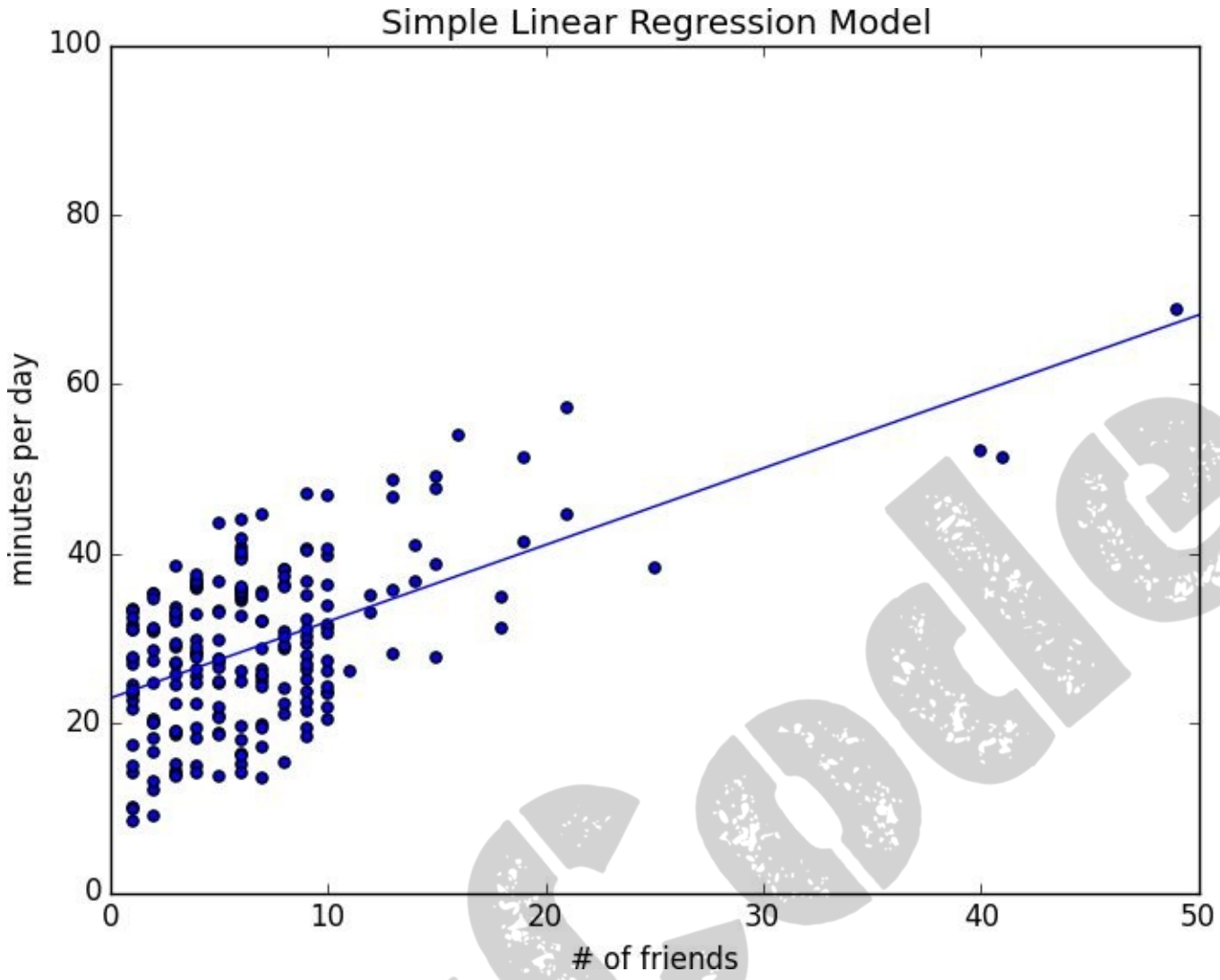

Figure 14-1. Our simple linear model

Of course, we need a better way to figure out how well we've fit the data than staring at the graph. A common measure is the *coefficient of determination* (or *R-squared*), which measures the fraction of the total variation in the dependent variable that is captured by the model:

```
def total_sum_of_squares(y):
    """the total squared variation of y_i's from their mean"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
    """the fraction of variation in y captured by the model, which equals
    1 - the fraction of variation in y not captured by the model"""

    return 1.0 - (sum_of_squared_errors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good)      # 0.329
```

Now, we chose the `alpha` and `beta` that minimized the sum of the squared prediction errors. One linear model we could have chosen is "always predict `mean(y)`" (corresponding to `alpha = mean(y)` and `beta = 0`), whose sum of squared errors exactly equals its total sum of squares. This means an R-squared of zero, which indicates a model that (obviously, in this case) performs no better than just predicting the mean.

Clearly, the least squares model must be at least as good as that one, which means that the sum of the squared errors is *at most* the total sum of squares, which means that the R-squared must be at least zero. And the sum of squared errors must be at least 0, which means that the R-squared can be at most 1.

The higher the number, the better our model fits the data. Here we calculate an R-squared of 0.329, which tells us that our model is only sort of okay at fitting the data, and that clearly there are other factors at play.

Using Gradient Descent

If we write `theta = [alpha, beta]`, then we can also solve this using gradient descent:

```
def squared_error(x_i, y_i, theta):
    alpha, beta = theta
    return error(alpha, beta, x_i, y_i) ** 2

def squared_error_gradient(x_i, y_i, theta):
    alpha, beta = theta
    return [-2 * error(alpha, beta, x_i, y_i),       # alpha partial derivative
            -2 * error(alpha, beta, x_i, y_i) * x_i] # beta partial derivative

# choose random value to start
random.seed(0)
theta = [random.random(), random.random()]
alpha, beta = minimize_stochastic(squared_error,
                                  squared_error_gradient,
                                  num_friends_good,
                                  daily_minutes_good,
                                  theta,
                                  0.0001)
print alpha, beta
```

Using the same data we get alpha = 22.93, beta = 0.905, which are very close to the exact answers.

Maximum Likelihood Estimation

Why choose least squares? One justification involves *maximum likelihood estimation*.

Imagine that we have a sample of data $v_1, \ldots, v_n$ that comes from a distribution that depends on some unknown parameter θ:

$$p(v_1, \ldots, v_n \mid \theta)$$

If we didn't know theta, we could turn around and think of this quantity as the *likelihood* of θ given the sample:

$$L(\theta \mid v_1, \ldots, v_n)$$

Under this approach, the most likely θ is the value that maximizes this likelihood function; that is, the value that makes the observed data the most probable. In the case of a continuous distribution, in which we have a probability distribution function rather than a probability mass function, we can do the same thing.

Back to regression. One assumption that's often made about the simple regression model is that the regression errors are normally distributed with mean 0 and some (known) standard deviation σ. If that's the case, then the likelihood based on seeing a pair `(x_i, y_i)` is:

$$L(\alpha, \beta \mid x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-(y_i - \alpha - \beta x_i)^2 / 2\sigma^2\right)$$

The likelihood based on the entire data set is the product of the individual likelihoods, which is largest precisely when `alpha` and `beta` are chosen to minimize the sum of squared errors. That is, in this case (and with these assumptions), minimizing the sum of squared errors is equivalent to maximizing the likelihood of the observed data.

Multiple Regression

> I don't look at a problem and put variables in there that don't affect it.
>
> Bill Parcells

Although the VP is pretty impressed with your predictive model, she thinks you can do better. To that end, you've collected additional data: for each of your users, you know how many hours he works each day, and whether he has a PhD. You'd like to use this additional data to improve your model.

Accordingly, you hypothesize a linear model with more independent variables:

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \beta_3 \text{phd} + \varepsilon$$

Obviously, whether a user has a PhD is not a number, but — as we mentioned in Chapter 11 — we can introduce a *dummy variable* that equals 1 for users with PhDs and 0 for users without, after which it's just as numeric as the other variables.

The Model

Recall that in Chapter 14 we fit a model of the form:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Now imagine that each input x_i is not a single number but rather a vector of *k* numbers $x_{i1}, \ldots, x_{ik}$. The multiple regression model assumes that:

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

In multiple regression the vector of parameters is usually called β. We'll want this to include the constant term as well, which we can achieve by adding a column of ones to our data:

```
beta = [alpha, beta_1, ..., beta_k]
```

and:

```
x_i = [1, x_i1, ..., x_ik]
```

Then our model is just:

```
def predict(x_i, beta):
    """assumes that the first element of each x_i is 1"""
    return dot(x_i, beta)
```

In this particular case, our independent variable `x` will be a list of vectors, each of which looks like this:

```
[1,    # constant term
 49,   # number of friends
 4,    # work hours per day
 0]    # doesn't have PhD
```

Further Assumptions of the Least Squares Model

There are a couple of further assumptions that are required for this model (and our solution) to make sense.

The first is that the columns of *x* are *linearly independent* — that there's no way to write any one as a weighted sum of some of the others. If this assumption fails, it's impossible to estimate `beta`. To see this in an extreme case, imagine we had an extra field `num_acquaintances` in our data that for every user was exactly equal to `num_friends`.

Then, starting with any `beta`, if we add *any* amount to the `num_friends` coefficient and subtract that same amount from the `num_acquaintances` coefficient, the model's predictions will remain unchanged. Which means that there's no way to find *the* coefficient for `num_friends`. (Usually violations of this assumption won't be so obvious.)

The second important assumption is that the columns of *x* are all uncorrelated with the errors ε. If this fails to be the case, our estimates of `beta` will be systematically wrong.

For instance, in Chapter 14, we built a model that predicted that each additional friend was associated with an extra 0.90 daily minutes on the site.

Imagine that it's also the case that:

- People who work more hours spend less time on the site.
- People with more friends tend to work more hours.

That is, imagine that the "actual" model is:

$$\text{minutes} = \alpha + \beta_1 \text{ friends} + \beta_2 \text{ work hours} + \varepsilon$$

and that work hours and friends are positively correlated. In that case, when we minimize the errors of the single variable model:

$$\text{minutes} = \alpha + \beta_1 \text{ friends} + \varepsilon$$

we will underestimate β_1.

Think about what would happen if we made predictions using the single variable model with the "actual" value of β_1. (That is, the value that arises from minimizing the errors of what we called the "actual" model.) The predictions would tend to be too small for users who work many hours and too large for users who work few hours, because $\beta_2 > 0$ and we "forgot" to include it. Because work hours is positively correlated with number of friends, this means the predictions tend to be too small for users with many friends and too large for users with few friends.

The result of this is that we can reduce the errors (in the single-variable model) by

decreasing our estimate of β_1, which means that the error-minimizing β_1 is smaller than the "actual" value. That is, in this case the single-variable least squares solution is biased to underestimate β_1. And, in general, whenever the independent variables are correlated with the errors like this, our least squares solution will give us a biased estimate of β.

Fitting the Model

As we did in the simple linear model, we'll choose `beta` to minimize the sum of squared errors. Finding an exact solution is not simple to do by hand, which means we'll need to use gradient descent. We'll start by creating an error function to minimize. For stochastic gradient descent, we'll just want the squared error corresponding to a single prediction:

```
def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2
```

If you know calculus, you can compute:

```
def squared_error_gradient(x_i, y_i, beta):
    """the gradient (with respect to beta)
    corresponding to the ith squared error term"""
    return [-2 * x_ij * error(x_i, y_i, beta)
            for x_ij in x_i]
```

Otherwise, you'll need to take my word for it.

At this point, we're ready to find the optimal beta using stochastic gradient descent:

```
def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)

random.seed(0)
beta = estimate_beta(x, daily_minutes_good) # [30.63, 0.972, -1.868, 0.911]
```

This means our model looks like:

$$\text{minutes} = 30.63 + 0.972\ \text{friends} - 1.868\ \text{work hours} + 0.911\ \text{phd}$$

Interpreting the Model

You should think of the coefficients of the model as representing all-else-being-equal estimates of the impacts of each factor. All else being equal, each additional friend corresponds to an extra minute spent on the site each day. All else being equal, each additional hour in a user's workday corresponds to about two fewer minutes spent on the site each day. All else being equal, having a PhD is associated with spending an extra minute on the site each day.

What this doesn't (directly) tell us is anything about the interactions among the variables. It's possible that the effect of work hours is different for people with many friends than it is for people with few friends. This model doesn't capture that. One way to handle this case is to introduce a new variable that is the *product* of "friends" and "work hours." This effectively allows the "work hours" coefficient to increase (or decrease) as the number of friends increases.

Or it's possible that the more friends you have, the more time you spend on the site *up to a point*, after which further friends cause you to spend less time on the site. (Perhaps with too many friends the experience is just too overwhelming?) We could try to capture this in our model by adding another variable that's the *square* of the number of friends.

Once we start adding variables, we need to worry about whether their coefficients "matter." There are no limits to the numbers of products, logs, squares, and higher powers we could add.

Goodness of Fit

Again we can look at the R-squared, which has now increased to 0.68:

```
def multiple_r_squared(x, y, beta):
    sum_of_squared_errors = sum(error(x_i, y_i, beta) ** 2
                                for x_i, y_i in zip(x, y))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

Keep in mind, however, that adding new variables to a regression will *necessarily* increase the R-squared. After all, the simple regression model is just the special case of the multiple regression model where the coefficients on “work hours” and “PhD” both equal 0. The optimal multiple regression model will necessarily have an error at least as small as that one.

Because of this, in a multiple regression, we also need to look at the *standard errors* of the coefficients, which measure how certain we are about our estimates of each β_i. The regression as a whole may fit our data very well, but if some of the independent variables are correlated (or irrelevant), their coefficients might not *mean* much.

The typical approach to measuring these errors starts with another assumption — that the errors ε_i are independent normal random variables with mean 0 and some shared (unknown) standard deviation σ. In that case, we (or, more likely, our statistical software) can use some linear algebra to find the standard error of each coefficient. The larger it is, the less sure our model is about that coefficient. Unfortunately, we’re not set up to do that kind of linear algebra from scratch.

Digression: The Bootstrap

Imagine we have a sample of *n* data points, generated by some (unknown to us) distribution:

```
data = get_sample(num_points=n)
```

In Chapter 5, we wrote a function to compute the `median` of the observed data, which we can use as an estimate of the median of the distribution itself.

But how confident can we be about our estimate? If all the data in the sample are very close to 100, then it seems likely that the actual median is close to 100. If approximately half the data in the sample is close to 0 and the other half is close to 200, then we can't be nearly as certain about the median.

If we could repeatedly get new samples, we could compute the median of each and look at the distribution of those medians. Usually we can't. What we can do instead is *bootstrap* new data sets by choosing *n* data points *with replacement* from our data and then compute the medians of those synthetic data sets:

```
def bootstrap_sample(data):
    """randomly samples len(data) elements with replacement"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data, stats_fn, num_samples):
    """evaluates stats_fn on num_samples bootstrap samples from data"""
    return [stats_fn(bootstrap_sample(data))
            for _ in range(num_samples)]
```

For example, consider the two following data sets:

```
# 101 points all very close to 100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# 101 points, 50 of them near 0, 50 of them near 200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

If you compute the `median` of each, both will be very close to 100. However, if you look at:

```
bootstrap_statistic(close_to_100, median, 100)
```

you will mostly see numbers really close to 100. Whereas if you look at:

```
bootstrap_statistic(far_from_100, median, 100)
```

you will see a lot of numbers close to 0 and a lot of numbers close to 200.

The `standard_deviation` of the first set of medians is close to 0, while the `standard_deviation` of the second set of medians is close to 100. (This extreme a case

would be pretty easy to figure out by manually inspecting the data, but in general that won’t be true.)

Standard Errors of Regression Coefficients

We can take the same approach to estimating the standard errors of our regression coefficients. We repeatedly take a `bootstrap_sample` of our data and estimate `beta` based on that sample. If the coefficient corresponding to one of the independent variables (say `num_friends`) doesn't vary much across samples, then we can be confident that our estimate is relatively tight. If the coefficient varies greatly across samples, then we can't be at all confident in our estimate.

The only subtlety is that, before sampling, we'll need to `zip` our `x` data and `y` data to make sure that corresponding values of the independent and dependent variables are sampled together. This means that `bootstrap_sample` will return a list of pairs `(x_i, y_i)`, which we'll need to reassemble into an `x_sample` and a `y_sample`:

```
def estimate_sample_beta(sample):
    """sample is a list of pairs (x_i, y_i)"""
    x_sample, y_sample = zip(*sample) # magic unzipping trick
    return estimate_beta(x_sample, y_sample)

random.seed(0) # so that you get the same results as me

bootstrap_betas = bootstrap_statistic(zip(x, daily_minutes_good),
                                      estimate_sample_beta,
                                      100)
```

After which we can estimate the standard deviation of each coefficient:

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

# [1.174,    # constant term, actual error = 1.19
#  0.079,    # num_friends,   actual error = 0.080
#  0.131,    # unemployed,    actual error = 0.127
#  0.990]    # phd,           actual error = 0.998
```

We can use these to test hypotheses such as "does β_i equal zero?" Under the null hypothesis $\beta_i = 0$ (and with our other assumptions about the distribution of ε_i) the statistic:

$$t_j = \hat{\beta}_j / \hat{\sigma}_j$$

which is our estimate of β_j divided by our estimate of its standard error, follows a *Student's t-distribution* with "$n - k$ degrees of freedom."

If we had a `students_t_cdf` function, we could compute *p*-values for each least-squares coefficient to indicate how likely we would be to observe such a value if the actual coefficient were zero. Unfortunately, we don't have such a function. (Although we would if we weren't working from scratch.)

However, as the degrees of freedom get large, the *t*-distribution gets closer and closer to a standard normal. In a situation like this, where *n* is much larger than *k*, we can use `normal_cdf` and still feel good about ourselves:

```
def p_value(beta_hat_j, sigma_hat_j):
    if beta_hat_j > 0:
        # if the coefficient is positive, we need to compute twice the
        # probability of seeing an even *larger* value
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # otherwise twice the probability of seeing a *smaller* value
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

p_value(30.63, 1.174)    # ~0   (constant term)
p_value(0.972, 0.079)    # ~0   (num_friends)
p_value(-1.868, 0.131)   # ~0   (work_hours)
p_value(0.911, 0.990)    # 0.36 (phd)
```

(In a situation not like this, we would probably be using statistical software that knows how to compute the *t*-distribution, as well as how to compute the exact standard errors.)

While most of the coefficients have very small *p*-values (suggesting that they are indeed nonzero), the coefficient for "PhD" is not "significantly" different from zero, which makes it likely that the coefficient for "PhD" is random rather than meaningful.

In more elaborate regression scenarios, you sometimes want to test more elaborate hypotheses about the data, such as "at least one of the β_j is non-zero" or "β_1 equals β_2 *and* β_3 equals β_4," which you can do with an *F-test*, which, alas, falls outside the scope of this book.

Regularization

In practice, you'd often like to apply linear regression to data sets with large numbers of variables. This creates a couple of extra wrinkles. First, the more variables you use, the more likely you are to overfit your model to the training set. And second, the more nonzero coefficients you have, the harder it is to make sense of them. If the goal is to *explain* some phenomenon, a sparse model with three factors might be more useful than a slightly better model with hundreds.

Regularization is an approach in which we add to the error term a penalty that gets larger as `beta` gets larger. We then minimize the combined error and penalty. The more importance we place on the penalty term, the more we discourage large coefficients.

For example, in *ridge regression*, we add a penalty proportional to the sum of the squares of the `beta_i`. (Except that typically we don't penalize `beta_0`, the constant term.)

```
# alpha is a *hyperparameter* controlling how harsh the penalty is
# sometimes it's called "lambda" but that already means something in Python
def ridge_penalty(beta, alpha):
  return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x_i, y_i, beta, alpha):
    """estimate error plus ridge penalty on beta"""
    return error(x_i, y_i, beta) ** 2 + ridge_penalty(beta, alpha)
```

which you can then plug into gradient descent in the usual way:

```
def ridge_penalty_gradient(beta, alpha):
    """gradient of just the ridge penalty"""
    return [0] + [2 * alpha * beta_j for beta_j in beta[1:]]

def squared_error_ridge_gradient(x_i, y_i, beta, alpha):
    """the gradient corresponding to the ith squared error term
    including the ridge penalty"""
    return vector_add(squared_error_gradient(x_i, y_i, beta),
                      ridge_penalty_gradient(beta, alpha))

def estimate_beta_ridge(x, y, alpha):
    """use gradient descent to fit a ridge regression
    with penalty alpha"""
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(partial(squared_error_ridge, alpha=alpha),
                               partial(squared_error_ridge_gradient,
                                       alpha=alpha),
                               x, y,
                               beta_initial,
                               0.001)
```

With `alpha` set to zero, there's no penalty at all and we get the same results as before:

```
random.seed(0)
beta_0 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.0)
# [30.6, 0.97, -1.87, 0.91]
dot(beta_0[1:], beta_0[1:]) # 5.26
multiple_r_squared(x, daily_minutes_good, beta_0) # 0.680
```

As we increase `alpha`, the goodness of fit gets worse, but the size of `beta` gets smaller:

```
beta_0_01 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.01)
# [30.6, 0.97, -1.86, 0.89]
dot(beta_0_01[1:], beta_0_01[1:])  # 5.19
multiple_r_squared(x, daily_minutes_good, beta_0_01)  # 0.680

beta_0_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.1)
# [30.8, 0.95, -1.84, 0.54]
dot(beta_0_1[1:], beta_0_1[1:])  # 4.60
multiple_r_squared(x, daily_minutes_good, beta_0_1)  # 0.680

beta_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=1)
# [30.7, 0.90, -1.69, 0.085]
dot(beta_1[1:], beta_1[1:])  # 3.69
multiple_r_squared(x, daily_minutes_good, beta_1)  # 0.676

beta_10 = estimate_beta_ridge(x, daily_minutes_good, alpha=10)
# [28.3, 0.72, -0.91, -0.017]
dot(beta_10[1:], beta_10[1:])  # 1.36
multiple_r_squared(x, daily_minutes_good, beta_10)  # 0.573
```

In particular, the coefficient on “PhD” vanishes as we increase the penalty, which accords with our previous result that it wasn’t significantly different from zero.

> **NOTE**
>
> Usually you’d want to `rescale` your data before using this approach. After all, if you changed years of experience to centuries of experience, its least squares coefficient would increase by a factor of 100 and suddenly get penalized much more, even though it’s the same model.

Another approach is *lasso* regression, which uses the penalty:

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Whereas the ridge penalty shrank the coefficients overall, the lasso penalty tends to force coefficients to be zero, which makes it good for learning sparse models. Unfortunately, it’s not amenable to gradient descent, which means that we won’t be able to solve it from scratch.

Logistic Regression

> A lot of people say there's a fine line between genius and insanity. I don't think there's a fine line, I actually think there's a yawning gulf.
>
> Bill Bailey

In Chapter 1, we briefly looked at the problem of trying to predict which DataSciencester users paid for premium accounts. Here we'll revisit that problem.

The Problem

We have an anonymized data set of about 200 users, containing each user's salary, her years of experience as a data scientist, and whether she paid for a premium account (Figure 16-1). As is usual with categorical variables, we represent the dependent variable as either 0 (no premium account) or 1 (premium account).

As usual, our data is in a matrix where each row is a list `[experience, salary, paid_account]`. Let's turn it into the format we need:

```
x = [[1] + row[:2] for row in data]  # each element is [1, experience, salary]
y = [row[2] for row in data]         # each element is paid_account
```

An obvious first attempt is to use linear regression and find the best model:

$$\text{paid account} = \beta_0 + \beta_1 \text{experience} + \beta_2 \text{salary} + \varepsilon$$

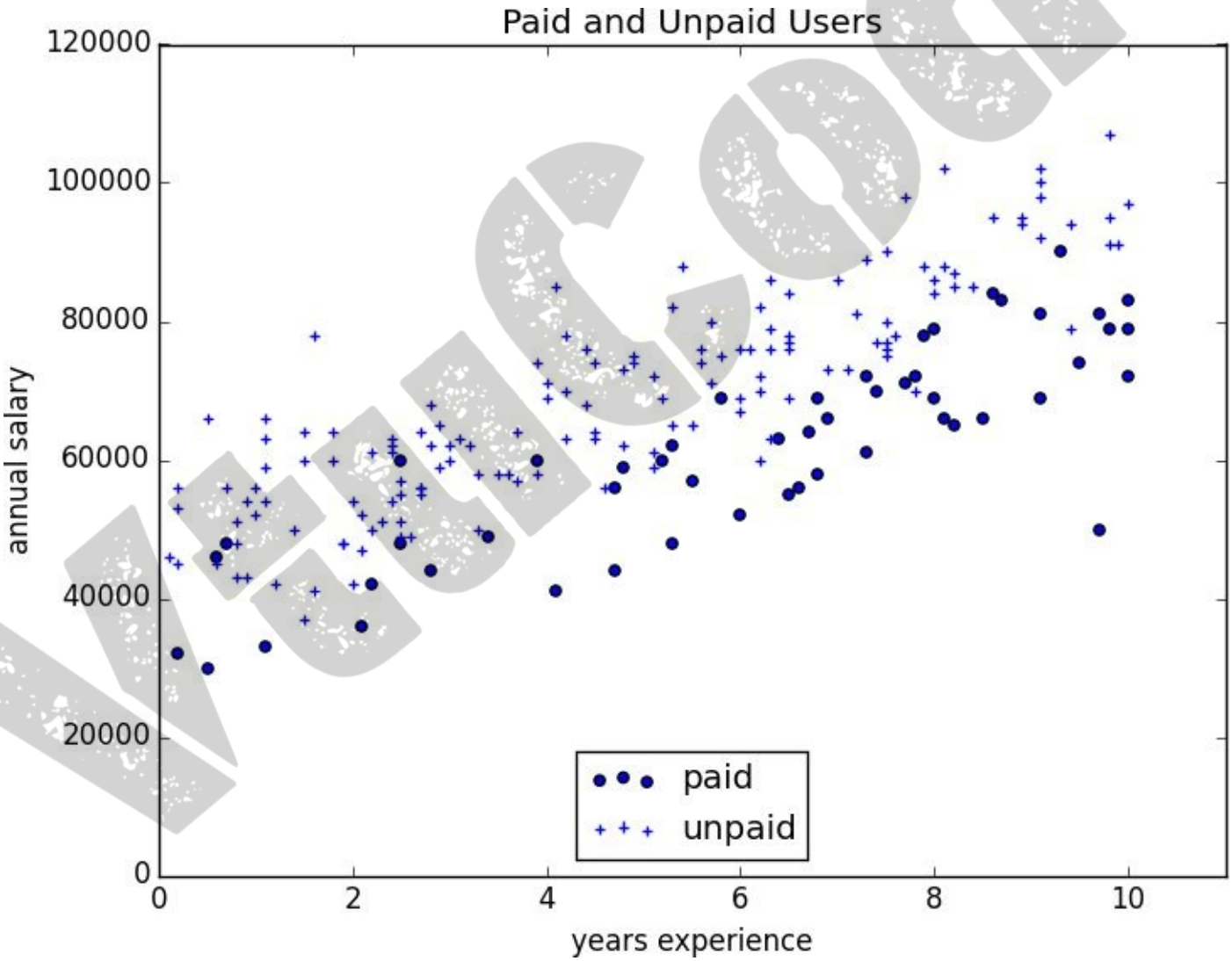

Figure 16-1. Paid and unpaid users

And certainly, there's nothing preventing us from modeling the problem this way. The results are shown in Figure 16-2:

```
rescaled_x = rescale(x)
```

```
beta = estimate_beta(rescaled_x, y)  # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]

plt.scatter(predictions, y)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

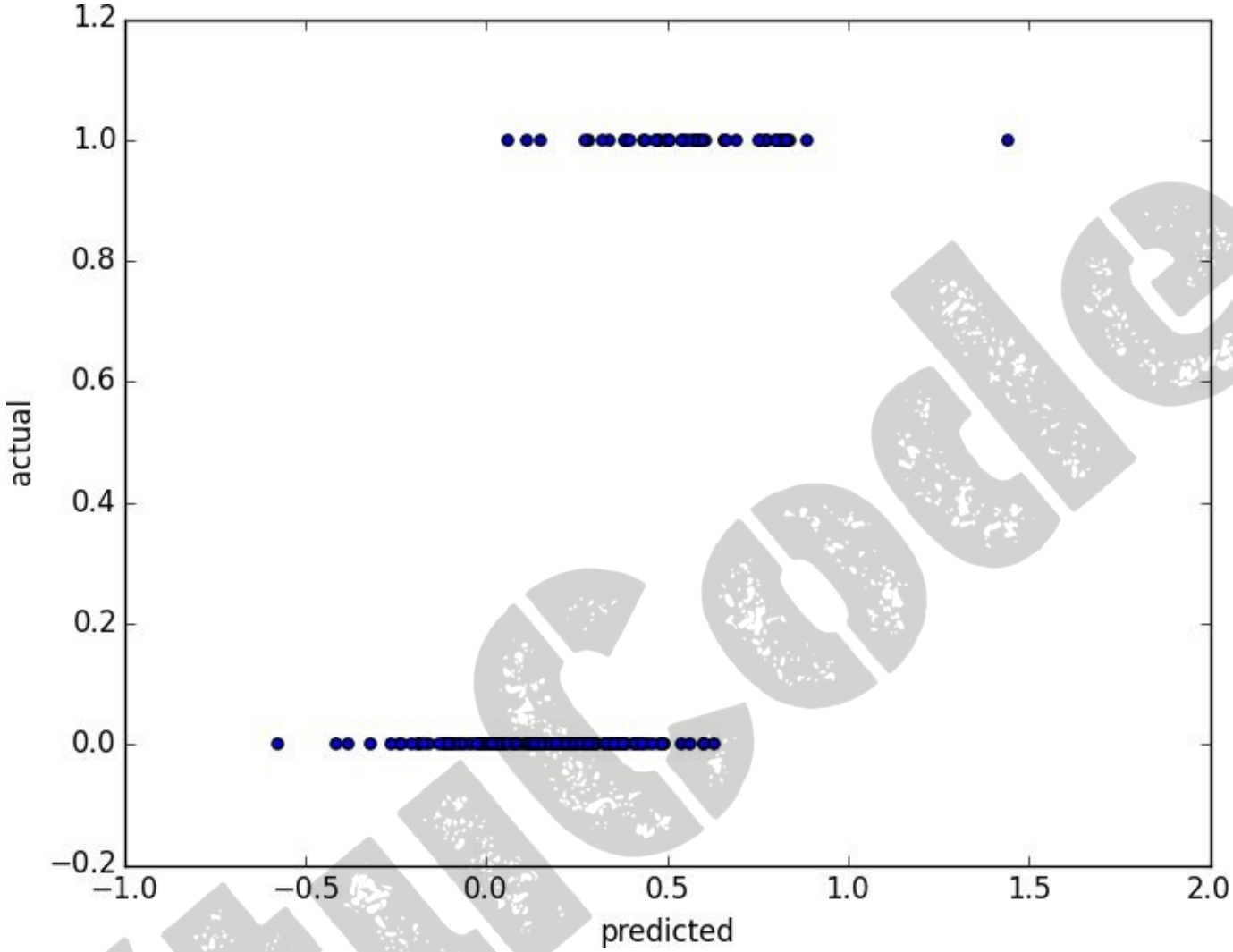

Figure 16-2. Using linear regression to predict premium accounts

But this approach leads to a couple of immediate problems:

- We'd like for our predicted outputs to be 0 or 1, to indicate class membership. It's fine if they're between 0 and 1, since we can interpret these as probabilities — an output of 0.25 could mean 25% chance of being a paid member. But the outputs of the linear model can be huge positive numbers or even negative numbers, which it's not clear how to interpret. Indeed, here a lot of our predictions were negative.

- The linear regression model assumed that the errors were uncorrelated with the columns of x. But here, the regression coefficent for `experience` is 0.43, indicating that more experience leads to a greater likelihood of a premium account. This means that our model outputs very large values for people with lots of experience. But we know that the actual values must be at most 1, which means that necessarily very large outputs (and therefore very large values of `experience`) correspond to very large negative values of the error term. Because this is the case, our estimate of `beta` is biased.

What we’d like instead is for large positive values of `dot(x_i, beta)` to correspond to probabilities close to 1, and for large negative values to correspond to probabilities close to 0. We can accomplish this by applying another function to the result.

The Logistic Function

In the case of logistic regression, we use the *logistic function*, pictured in Figure 16-3:

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

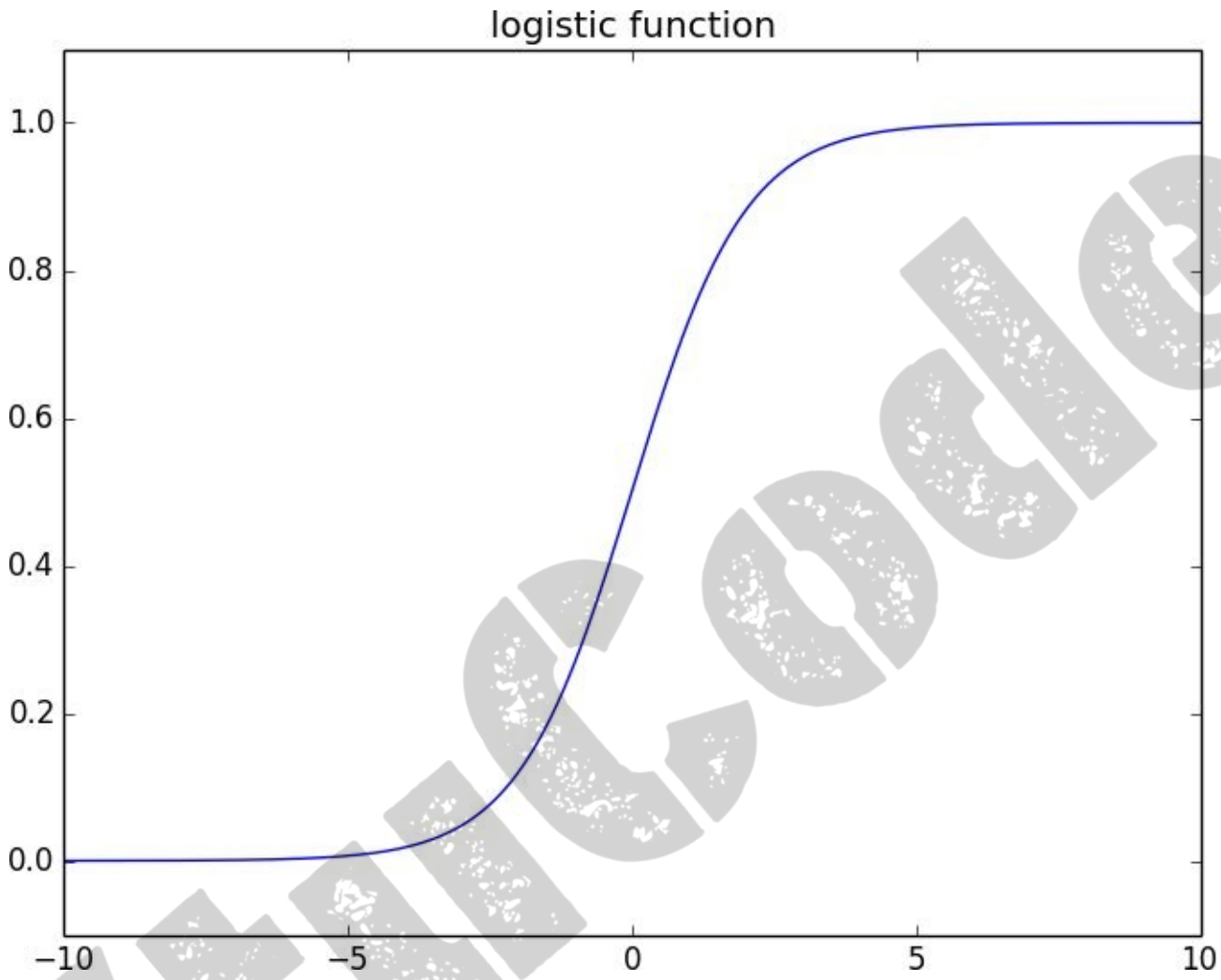

Figure 16-3. The logistic function

As its input gets large and positive, it gets closer and closer to 1. As its input gets large and negative, it gets closer and closer to 0. Additionally, it has the convenient property that its derivative is given by:

```
def logistic_prime(x):
    return logistic(x) * (1 - logistic(x))
```

which we'll make use of in a bit. We'll use this to fit a model:

$$y_i = f(x_i\beta) + \varepsilon_i$$

where *f* is the `logistic` function.

Recall that for linear regression we fit the model by minimizing the sum of squared errors,

which ended up choosing the β that maximized the likelihood of the data.

Here the two aren't equivalent, so we'll use gradient descent to maximize the likelihood directly. This means we need to calculate the likelihood function and its gradient.

Given some β, our model says that each y_i should equal 1 with probability $f(x_i\beta)$ and 0 with probability $1 - f(x_i\beta)$.

In particular, the pdf for y_i can be written as:

$$p(y_i \mid x_i, \beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1-y_i}$$

since if y_i is 0, this equals:

$$1 - f(x_i\beta)$$

and if y_i is 1, it equals:

$$f(x_i\beta)$$

It turns out that it's actually simpler to maximize the *log likelihood*:

$$\log L(\beta \mid x_i, y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log (1 - f(x_i\beta))$$

Because log is strictly increasing function, any `beta` that maximizes the log likelihood also maximizes the likelihood, and vice versa.

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

If we assume different data points are independent from one another, the overall likelihood is just the product of the individual likelihoods. Which means the overall log likelihood is the sum of the individual log likelihoods:

```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

A little bit of calculus gives us the gradient:

```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """here i is the index of the data point,
    j the index of the derivative"""

    return (y_i - logistic(dot(x_i, beta))) * x_i[j]
```

```
def logistic_log_gradient_i(x_i, y_i, beta):
    """the gradient of the log likelihood
    corresponding to the ith data point"""

    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

at which point we have all the pieces we need.

Applying the Model

We'll want to split our data into a training set and a test set:

```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# want to maximize log likelihood on the training data
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# pick a random starting point
beta_0 = [random.random() for _ in range(3)]

# and maximize using gradient descent
beta_hat = maximize_batch(fn, gradient_fn, beta_0)
```

Alternatively, you could use stochastic gradient descent:

```
beta_hat = maximize_stochastic(logistic_log_likelihood_i,
                               logistic_log_gradient_i,
                               x_train, y_train, beta_0)
```

Either way we find approximately:

```
beta_hat = [-1.90, 4.05, -3.87]
```

These are coefficients for the `rescaled` data, but we can transform them back to the original data as well:

```
beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

Unfortunately, these are not as easy to interpret as linear regression coefficients. All else being equal, an extra year of experience adds 1.42 to the input of `logistic`. All else being equal, an extra $10,000 of salary subtracts 2.49 from the input of `logistic`.

The impact on the output, however, depends on the other inputs as well. If `dot(beta, x_i)` is already large (corresponding to a probability close to 1), increasing it even by a lot cannot affect the probability very much. If it's close to 0, increasing it just a little might increase the probability quite a bit.

What we can say is that — all else being equal — people with more experience are more likely to pay for accounts. And that — all else being equal — people with higher salaries are less likely to pay for accounts. (This was also somewhat apparent when we plotted the data.)

Goodness of Fit

We haven't yet used the test data that we held out. Let's see what happens if we predict *paid account* whenever the probability exceeds 0.5:

```
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))

    if y_i == 1 and predict >= 0.5:  # TP: paid and we predict paid
        true_positives += 1
    elif y_i == 1:                   # FN: paid and we predict unpaid
        false_negatives += 1
    elif predict >= 0.5:             # FP: unpaid and we predict paid
        false_positives += 1
    else:                            # TN: unpaid and we predict unpaid
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

This gives a precision of 93% ("when we predict *paid account* we're right 93% of the time") and a recall of 82% ("when a user has a paid account we predict *paid account* 82% of the time"), both of which are pretty respectable numbers.

We can also plot the predictions versus the actuals (Figure 16-4), which also shows that the model performs well:

```
predictions = [logistic(dot(beta_hat, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test)
plt.xlabel("predicted probability")
plt.ylabel("actual outcome")
plt.title("Logistic Regression Predicted vs. Actual")
plt.show()
```

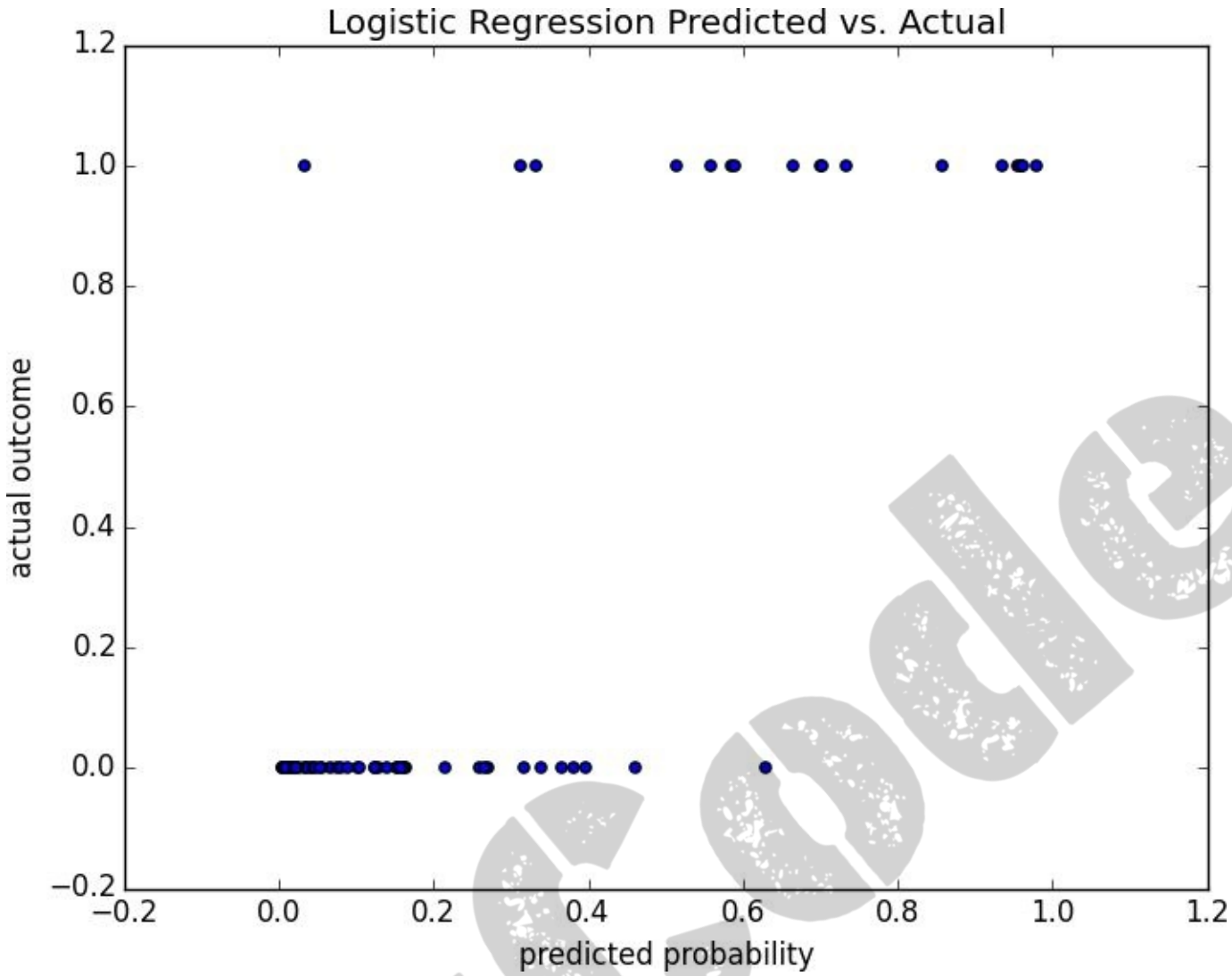

Figure 16-4. Logistic regression predicted versus actual

Support Vector Machines

The set of points where `dot(beta_hat, x_i)` equals 0 is the boundary between our classes. We can plot this to see exactly what our model is doing (Figure 16-5).

This boundary is a *hyperplane* that splits the parameter space into two half-spaces corresponding to *predict paid* and *predict unpaid*. We found it as a side-effect of finding the most likely logistic model.

An alternative approach to classification is to just look for the hyperplane that "best" separates the classes in the training data. This is the idea behind the *support vector machine*, which finds the hyperplane that maximizes the distance to the nearest point in each class (Figure 16-6).

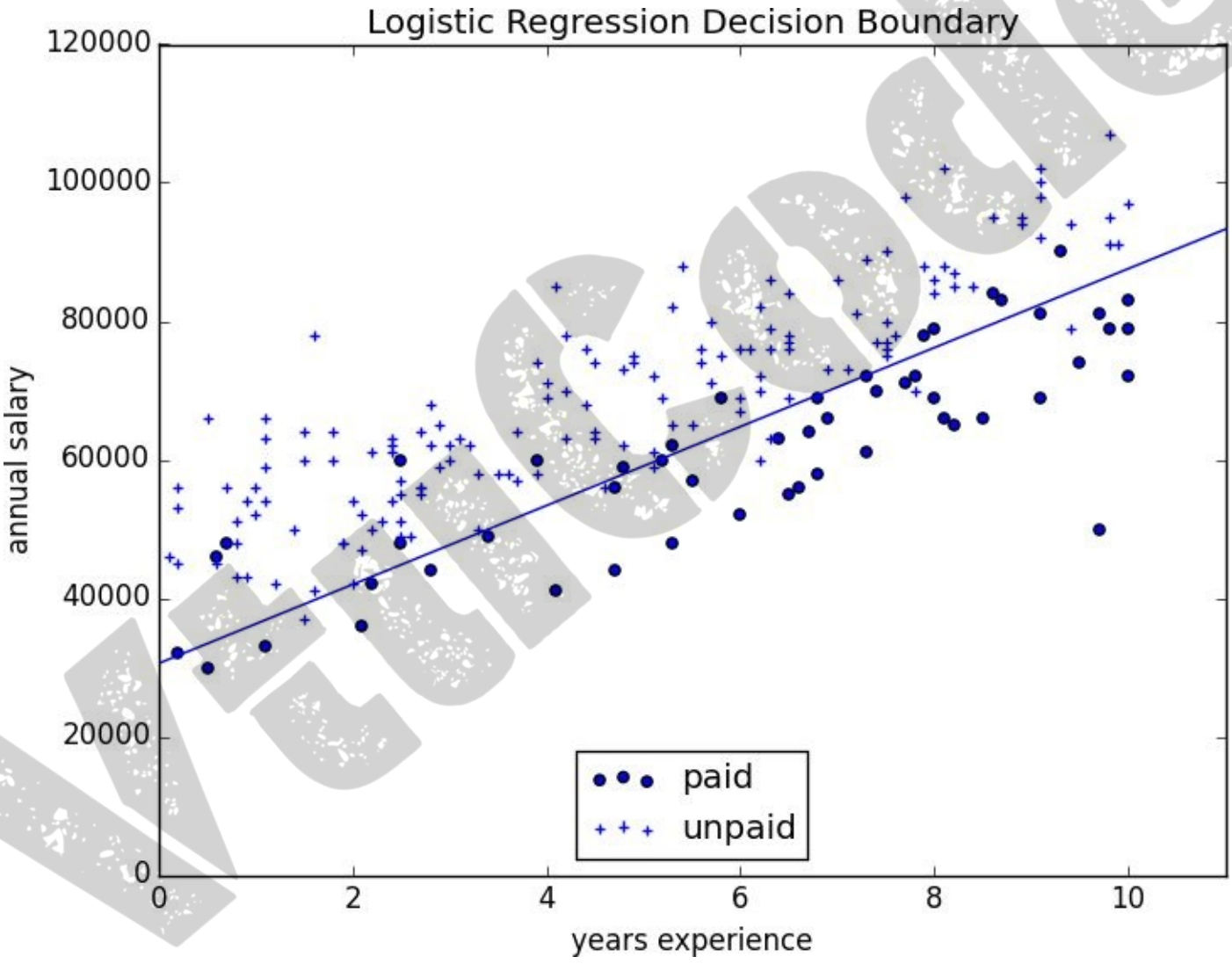

Figure 16-5. Paid and unpaid users with decision boundary

Finding such a hyperplane is an optimization problem that involves techniques that are too advanced for us. A different problem is that a separating hyperplane might not exist at all. In our "who pays?" data set there simply is no line that perfectly separates the paid users from the unpaid users.

We can (sometimes) get around this by transforming the data into a higher-dimensional space. For example, consider the simple one-dimensional data set shown in Figure 16-7.

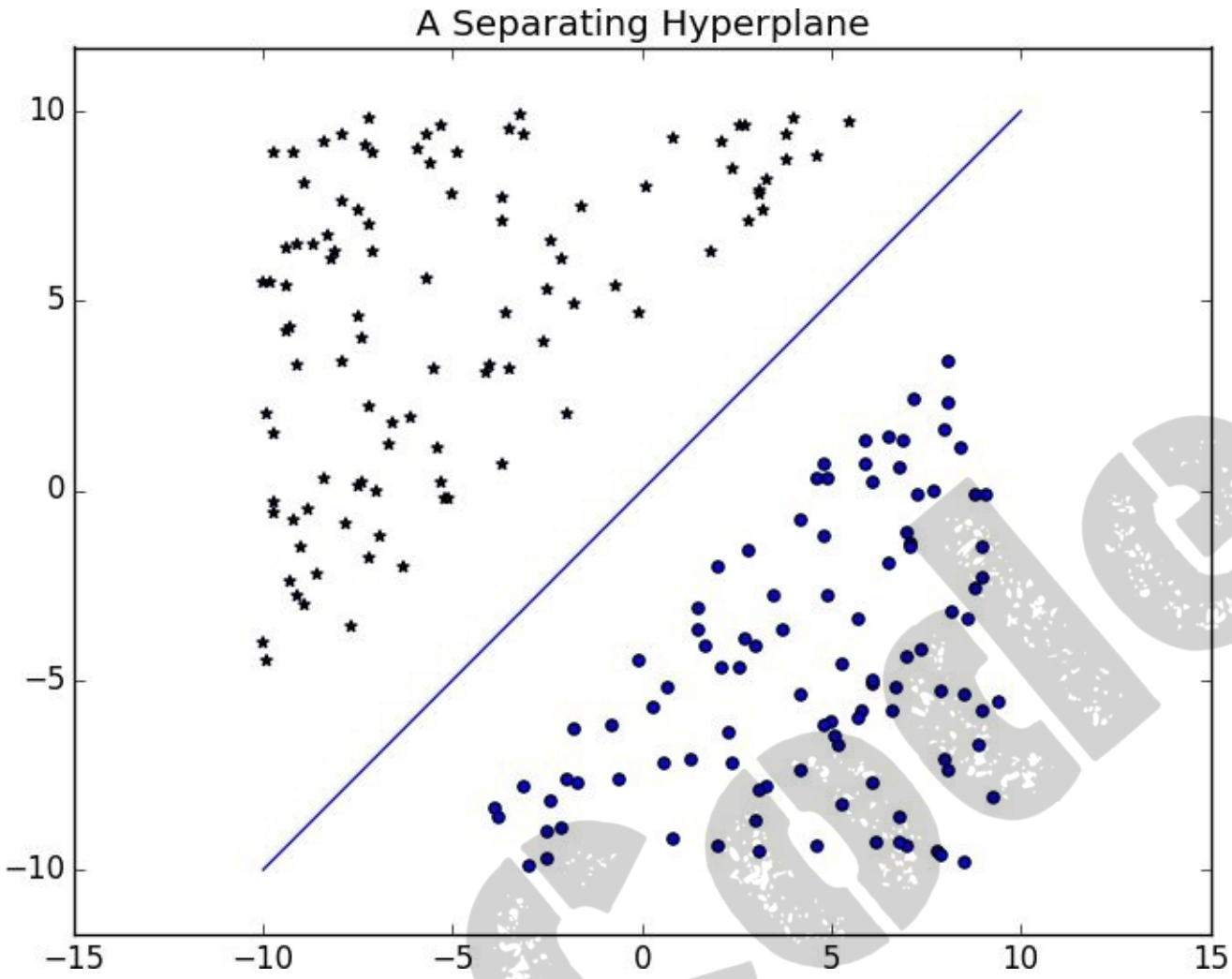

Figure 16-6. A separating hyperplane

It's clear that there's no hyperplane that separates the positive examples from the negative ones. However, look at what happens when we map this data set to two dimensions by sending the point `x` to `(x, x**2)`. Suddenly it's possible to find a hyperplane that splits the data (Figure 16-8).

This is usually called the *kernel trick* because rather than actually mapping the points into the higher-dimensional space (which could be expensive if there are a lot of points and the mapping is complicated), we can use a "kernel" function to compute dot products in the higher-dimensional space and use those to find a hyperplane.

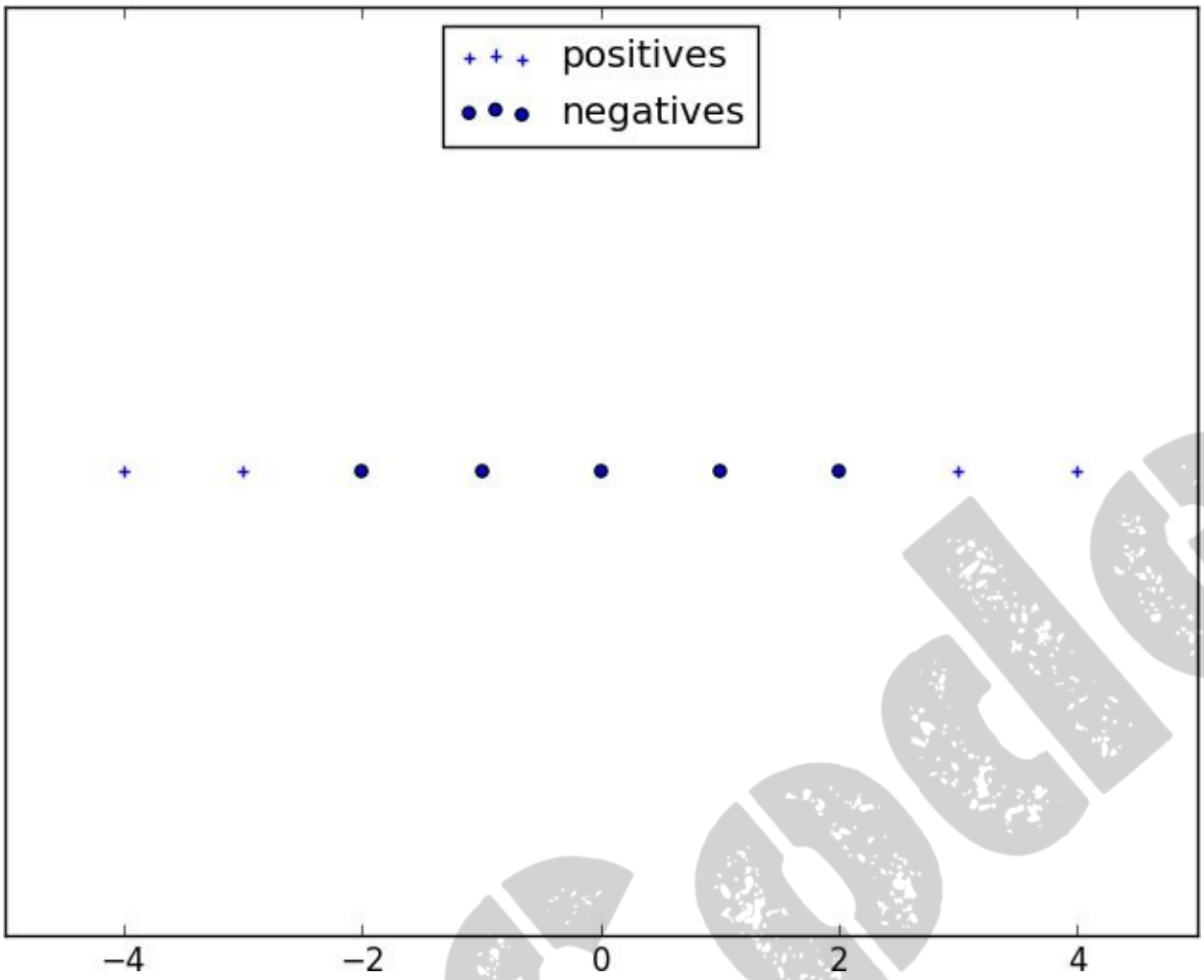

Figure 16-7. A nonseparable one-dimensional data set

It’s hard (and probably not a good idea) to *use* support vector machines without relying on specialized optimization software written by people with the appropriate expertise, so we’ll have to leave our treatment here.

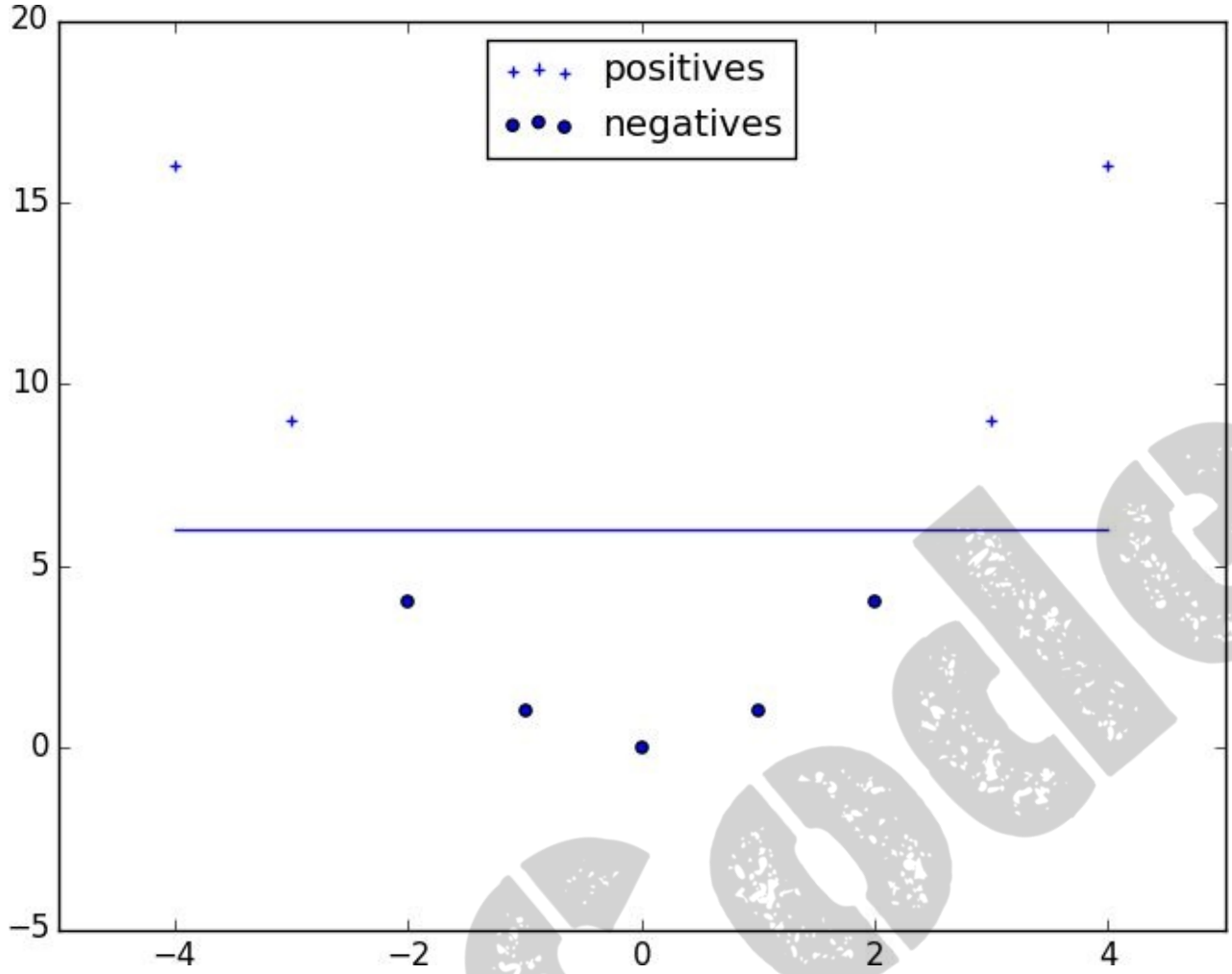

Figure 16-8. Data set becomes separable in higher dimensions