What Is Data Science?

There's a joke that says a data scientist is someone who knows more statistics than a computer scientist and more computer science than a statistician. (I didn't say it was a good joke.) In fact, some data scientists are — for all practical purposes — statisticians, while others are pretty much indistinguishable from software engineers. Some are machine-learning experts, while others couldn't machine-learn their way out of kindergarten. Some are PhDs with impressive publication records, while others have never read an academic paper (shame on them, though). In short, pretty much no matter how you define data science, you'll find practitioners for whom the definition is totally, absolutely wrong.

Nonetheless, we won't let that stop us from trying. We'll say that a data scientist is someone who extracts insights from messy data. Today's world is full of people trying to turn data into insight.

For instance, the dating site OkCupid asks its members to answer thousands of questions in order to find the most appropriate matches for them. But it also analyzes these results to figure out innocuous-sounding questions you can ask someone to find out how likely someone is to sleep with you on the first date.

Facebook asks you to list your hometown and your current location, ostensibly to make it easier for your friends to find and connect with you. But it also analyzes these locations to identify global migration patterns and where the fanbases of different football teams live.

As a large retailer, Target tracks your purchases and interactions, both online and in-store. And it uses the data to predictively model which of its customers are pregnant, to better market baby-related purchases to them.

In 2012, the Obama campaign employed dozens of data scientists who data-mined and experimented their way to identifying voters who needed extra attention, choosing optimal donor-specific fundraising appeals and programs, and focusing get-out-the-vote efforts where they were most likely to be useful. It is generally agreed that these efforts played an important role in the president's re-election, which means it is a safe bet that political campaigns of the future will become more and more data-driven, resulting in a never-ending arms race of data science and data collection.

Now, before you start feeling too jaded: some data scientists also occasionally use their skills for good — using data to make government more effective, to help the homeless, and to improve public health. But it certainly won't hurt your career if you like figuring out the best way to get people to click on advertisements.

Motivating Hypothetical: DataSciencester

Congratulations! You've just been hired to lead the data science efforts at DataSciencester, *the* social network for data scientists.

Despite being *for* data scientists, DataSciencester has never actually invested in building its own data science practice. (In fairness, DataSciencester has never really invested in building its product either.) That will be your job! Throughout the book, we'll be learning about data science concepts by solving problems that you encounter at work. Sometimes we'll look at data explicitly supplied by users, sometimes we'll look at data generated through their interactions with the site, and sometimes we'll even look at data from experiments that we'll design.

And because DataSciencester has a strong "not-invented-here" mentality, we'll be building our own tools from scratch. At the end, you'll have a pretty solid understanding of the fundamentals of data science. And you'll be ready to apply your skills at a company with a less shaky premise, or to any other problems that happen to interest you.

Welcome aboard, and good luck! (You're allowed to wear jeans on Fridays, and the bathroom is down the hall on the right.)

Finding Key Connectors

It's your first day on the job at DataSciencester, and the VP of Networking is full of questions about your users. Until now he's had no one to ask, so he's very excited to have you aboard.

In particular, he wants you to identify who the "key connectors" are among data scientists. To this end, he gives you a dump of the entire DataSciencester network. (In real life, people don't typically hand you the data you need. Chapter 9 is devoted to getting data.)

What does this data dump look like? It consists of a list of users, each represented by a `dict` that contains for each user his or her `id` (which is a number) and `name` (which, in one of the great cosmic coincidences, rhymes with the user's `id`):

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

He also gives you the "friendship" data, represented as a list of pairs of IDs:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

For example, the tuple `(0, 1)` indicates that the data scientist with `id` 0 (Hero) and the data scientist with `id` 1 (Dunn) are friends. The network is illustrated in Figure 1-1.

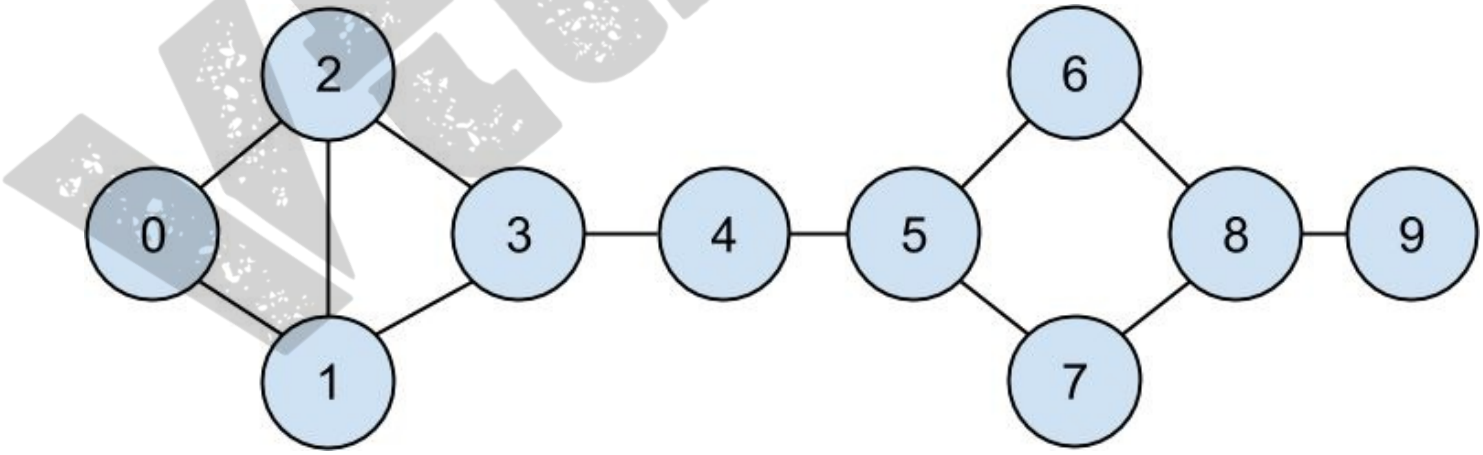

Figure 1-1. The DataSciencester network

Since we represented our users as `dicts`, it's easy to augment them with extra data.

> **NOTE**
>
> Don't get too hung up on the details of the code right now. In Chapter 2, we'll take you through a crash course in Python. For now just try to get the general flavor of what we're doing.

For example, we might want to add a list of friends to each user. First we set each user's `friends` property to an empty list:

```
for user in users:
    user["friends"] = []
```

And then we populate the lists using the `friendships` data:

```
for i, j in friendships:
    # this works because users[i] is the user whose id is i
    users[i]["friends"].append(users[j]) # add i as a friend of j
    users[j]["friends"].append(users[i]) # add j as a friend of i
```

Once each `user` `dict` contains a list of friends, we can easily ask questions of our graph, like "what's the average number of connections?"

First we find the *total* number of connections, by summing up the lengths of all the `friends` lists:

```
def number_of_friends(user):
    """how many friends does _user_ have?"""
    return len(user["friends"])                    # length of friend_ids list

total_connections = sum(number_of_friends(user)
                        for user in users)         # 24
```

And then we just divide by the number of users:

```
from __future__ import division                    # integer division is lame
num_users = len(users)                             # length of the users list
avg_connections = total_connections / num_users    # 2.4
```

It's also easy to find the most connected people — they're the people who have the largest number of friends.

Since there aren't very many users, we can sort them from "most friends" to "least friends":

```
# create a list (user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

sorted(num_friends_by_id,                                  # get it sorted
       key=lambda (user_id, num_friends): num_friends,     # by num_friends
       reverse=True)                                       # largest to smallest

# each pair is (user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

One way to think of what we've done is as a way of identifying people who are somehow central to the network. In fact, what we've just computed is the network metric *degree centrality* (Figure 1-2).

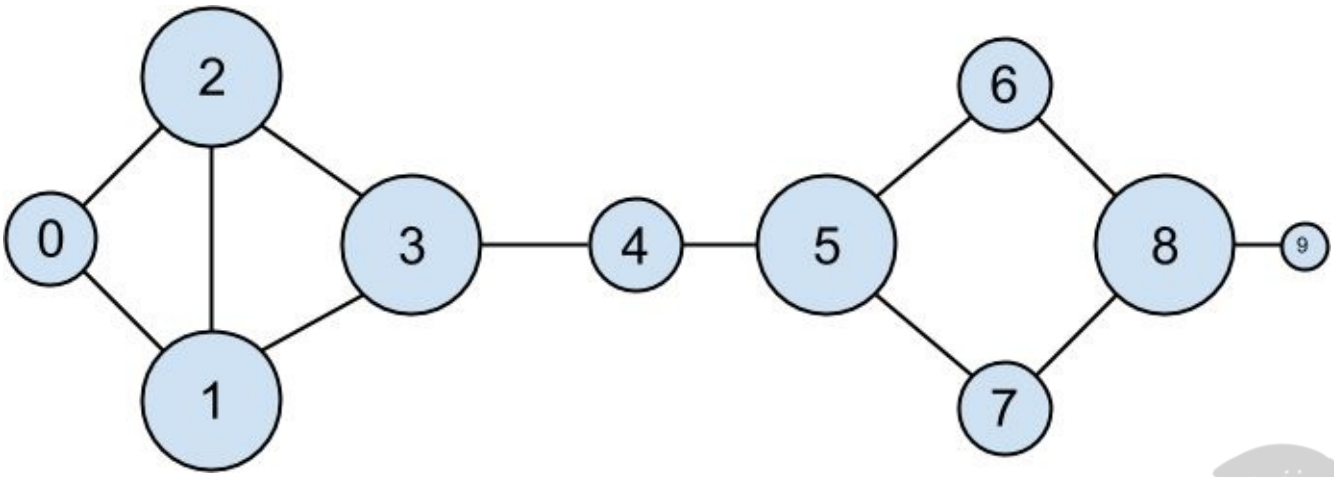

Figure 1-2. The DataSciencester network sized by degree

This has the virtue of being pretty easy to calculate, but it doesn't always give the results you'd want or expect. For example, in the DataSciencester network Thor (`id` 4) only has two connections while Dunn (`id` 1) has three. Yet looking at the network it intuitively seems like Thor should be more central. In Chapter 21, we'll investigate networks in more detail, and we'll look at more complex notions of centrality that may or may not accord better with our intuition.

Data Scientists You May Know

While you're still filling out new-hire paperwork, the VP of Fraternization comes by your desk. She wants to encourage more connections among your members, and she asks you to design a "Data Scientists You May Know" suggester.

Your first instinct is to suggest that a user might know the friends of friends. These are easy to compute: for each of a user's friends, iterate over that person's friends, and collect all the results:

```
def friends_of_friend_ids_bad(user):
    # "foaf" is short for "friend of a friend"
    return [foaf["id"]
            for friend in user["friends"]     # for each of user's friends
            for foaf in friend["friends"]]    # get each of _their_ friends
```

When we call this on `users[0]` (Hero), it produces:

```
[0, 2, 3, 0, 1, 3]
```

It includes user 0 (twice), since Hero is indeed friends with both of his friends. It includes users 1 and 2, although they are both friends with Hero already. And it includes user 3 twice, as Chi is reachable through two different friends:

```
print [friend["id"] for friend in users[0]["friends"]]  # [1, 2]
print [friend["id"] for friend in users[1]["friends"]]  # [0, 2, 3]
print [friend["id"] for friend in users[2]["friends"]]  # [0, 1, 3]
```

Knowing that people are friends-of-friends in multiple ways seems like interesting information, so maybe instead we should produce a *count* of mutual friends. And we definitely should use a helper function to exclude people already known to the user:

```
from collections import Counter                     # not loaded by default

def not_the_same(user, other_user):
    """two users are not the same if they have different ids"""
    return user["id"] != other_user["id"]

def not_friends(user, other_user):
    """other_user is not a friend if he's not in user["friends"];
    that is, if he's not_the_same as all the people in user["friends"]"""
    return all(not_the_same(friend, other_user)
               for friend in user["friends"])

def friends_of_friend_ids(user):
    return Counter(foaf["id"]
                   for friend in user["friends"]    # for each of my friends
                   for foaf in friend["friends"]    # count *their* friends
                   if not_the_same(user, foaf)      # who aren't me
                   and not_friends(user, foaf))     # and aren't my friends

print friends_of_friend_ids(users[3])               # Counter({0: 2, 5: 1})
```

This correctly tells Chi (`id` 3) that she has two mutual friends with Hero (`id` 0) but only one mutual friend with Clive (`id` 5).

As a data scientist, you know that you also might enjoy meeting users with similar

interests. (This is a good example of the "substantive expertise" aspect of data science.) After asking around, you manage to get your hands on this data, as a list of pairs `(user_id, interest)`:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

For example, Thor (`id` 4) has no friends in common with Devin (`id` 7), but they share an interest in machine learning.

It's easy to build a function that finds users with a certain interest:

```
def data_scientists_who_like(target_interest):
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

This works, but it has to examine the whole list of interests for every search. If we have a lot of users and interests (or if we just want to do a lot of searches), we're probably better off building an index from interests to users:

```
from collections import defaultdict

# keys are interests, values are lists of user_ids with that interest
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

And another from users to interests:

```
# keys are user_ids, values are lists of interests for that user_id
interests_by_user_id = defaultdict(list)

for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

Now it's easy to find who has the most interests in common with a given user:

- Iterate over the user's interests.
- For each interest, iterate over the other users with that interest.
- Keep count of how many times we see each other user.

```
def most_common_interests_with(user):
    return Counter(interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"])
```

We could then use this to build a richer “Data Scientists You Should Know” feature based on a combination of mutual friends and mutual interests. We’ll explore these kinds of applications in Chapter 22.

Salaries and Experience

Right as you're about to head to lunch, the VP of Public Relations asks if you can provide some fun facts about how much data scientists earn. Salary data is of course sensitive, but he manages to provide you an anonymous data set containing each user's `salary` (in dollars) and `tenure` as a data scientist (in years):

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

The natural first step is to plot the data (which we'll see how to do in Chapter 3). You can see the results in Figure 1-3.

Figure 1-3. Salary by years of experience

It seems pretty clear that people with more experience tend to earn more. How can you turn this into a fun fact? Your first idea is to look at the average salary for each tenure:

```
# keys are years, values are lists of the salaries for each tenure
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)
```

```
# keys are years, each value is average salary for that tenure
average_salary_by_tenure = {
    tenure : sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

This turns out to be not particularly useful, as none of the users have the same tenure, which means we're just reporting the individual users' salaries:

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

It might be more helpful to bucket the tenures:

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

Then group together the salaries corresponding to each bucket:

```
# keys are tenure buckets, values are lists of salaries for that bucket
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

And finally compute the average salary for each group:

```
# keys are tenure buckets, values are average salary for that bucket
average_salary_by_bucket = {
  tenure_bucket : sum(salaries) / len(salaries)
  for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
```

which is more interesting:

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

And you have your soundbite: "Data scientists with more than five years experience earn 65% more than data scientists with little or no experience!"

But we chose the buckets in a pretty arbitrary way. What we'd really like is to make some sort of statement about the salary effect — on average — of having an additional year of

experience. In addition to making for a snappier fun fact, this allows us to *make predictions* about salaries that we don't know. We'll explore this idea in Chapter 14.

Paid Accounts

When you get back to your desk, the VP of Revenue is waiting for you. She wants to better understand which users pay for accounts and which don't. (She knows their names, but that's not particularly actionable information.)

You notice that there seems to be a correspondence between years of experience and paid accounts:

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6   unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10  paid
```

Users with very few and very many years of experience tend to pay; users with average amounts of experience don't.

Accordingly, if you wanted to create a model — though this is definitely not enough data to base a model on — you might try to predict "paid" for users with very few and very many years of experience, and "unpaid" for users with middling amounts of experience:

```
def predict_paid_or_unpaid(years_experience):
  if years_experience < 3.0:
    return "paid"
  elif years_experience < 8.5:
    return "unpaid"
  else:
    return "paid"
```

Of course, we totally eyeballed the cutoffs.

With more data (and more mathematics), we could build a model predicting the likelihood that a user would pay, based on his years of experience. We'll investigate this sort of problem in Chapter 16.

Topics of Interest

As you're wrapping up your first day, the VP of Content Strategy asks you for data about what topics users are most interested in, so that she can plan out her blog calendar accordingly. You already have the raw data from the friend-suggester project:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

One simple (if not particularly exciting) way to find the most popular interests is simply to count the words:

1. Lowercase each interest (since different users may or may not capitalize their interests).
2. Split it into words.
3. Count the results.

In code:

```
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

This makes it easy to list out the words that occur more than once:

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print word, count
```

which gives the results you'd expect (unless you expect "scikit-learn" to get split into two words, in which case it doesn't give the results you expect):

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
```

```
networks 2
machine 2
neural 2
scikit-learn 2
r 2
```

We'll look at more sophisticated ways to extract topics from data in Chapter 20.

Visualizing Data

> I believe that visualization is one of the most powerful means of achieving personal goals.
>
> Harvey Mackay

A fundamental part of the data scientist's toolkit is data visualization. Although it is very easy to create visualizations, it's much harder to produce *good* ones.

There are two primary uses for data visualization:

- To *explore* data
- To *communicate* data

In this chapter, we will concentrate on building the skills that you'll need to start exploring your own data and to produce the visualizations we'll be using throughout the rest of the book. Like most of our chapter topics, data visualization is a rich field of study that deserves its own book. Nonetheless, we'll try to give you a sense of what makes for a good visualization and what doesn't.

matplotlib

A wide variety of tools exists for visualizing data. We will be using the `matplotlib` library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the Web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

In particular, we will be using the `matplotlib.pyplot` module. In its simplest use, `pyplot` maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it (with `savefig()`) or display it (with `show()`).

For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```

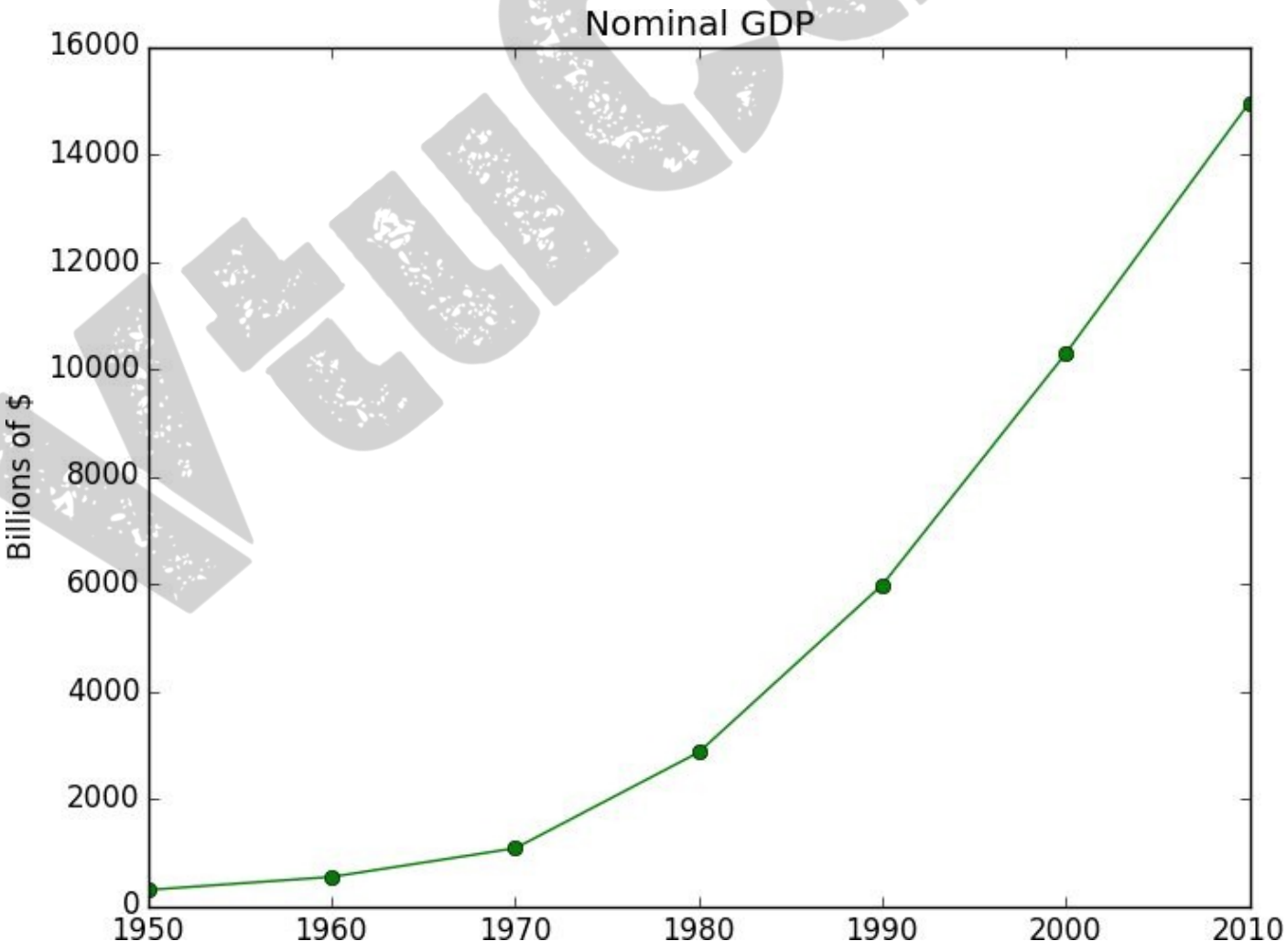

Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with (for example) axis labels, line styles, and point markers. Rather than attempt a comprehensive treatment of these options, we'll just use (and call attention to) some of them in our examples.

> **NOTE**
>
> Although we won't be using much of this functionality, `matplotlib` is capable of producing complicated plots within plots, sophisticated formatting, and interactive visualizations. Check out its documentation if you want to go deeper than we do in this book.

Bar Charts

A bar chart is a good choice when you want to show how some quantity varies among some *discrete* set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# bars are by default width 0.8, so we'll add 0.1 to the left coordinates
# so that each bar is centered
xs = [i + 0.1 for i, _ in enumerate(movies)]

# plot bars with left x-coordinates [xs], heights [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")

# label x-axis with movie names at bar centers
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

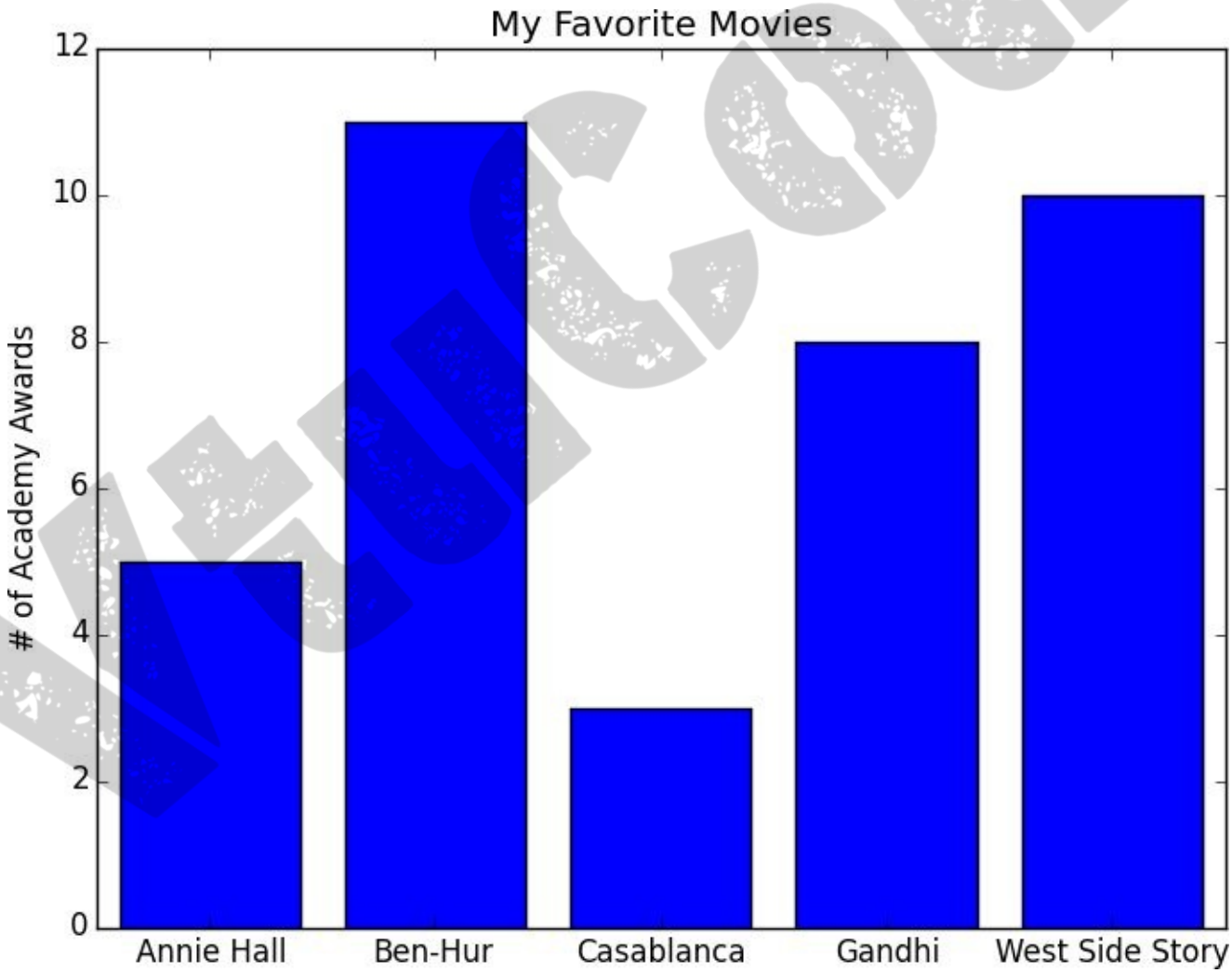

Figure 3-2. A simple bar chart

A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are *distributed*, as in Figure 3-3:

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # shift each bar to the left by 4
        histogram.values(),                # give each bar its correct height
        8)                                 # give each bar a width of 8

plt.axis([-5, 105, 0, 5])                  # x-axis from -5 to 105,
                                           # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])    # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```

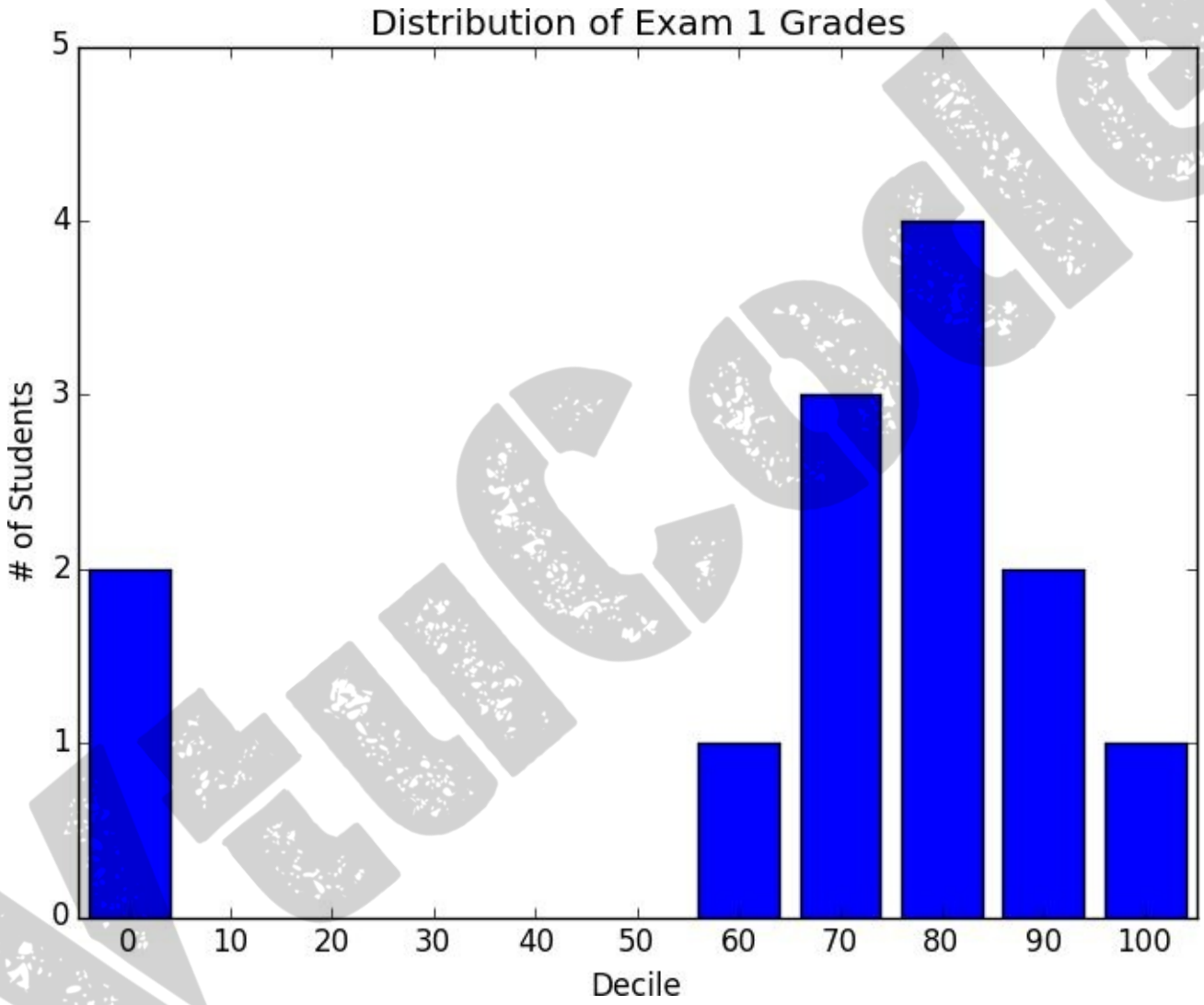

Figure 3-3. Using a bar chart for a histogram

The third argument to `plt.bar` specifies the bar width. Here we chose a width of 8 (which leaves a small gap between bars, since our buckets have width 10). And we shifted the bar left by 4, so that (for example) the "80" bar has its left and right sides at 76 and 84, and (hence) its center at 80.

The call to `plt.axis` indicates that we want the x-axis to range from -5 to 105 (so that the "0" and "100" bars are fully shown), and that the y-axis should range from 0 to 5. And the call to `plt.xticks` puts x-axis labels at 0, 10, 20, …, 100.

Be judicious when using `plt.axis()`. When creating bar charts it is considered especially bad form for your y-axis not to start at 0, since this is an easy way to mislead people (Figure 3-4):

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("# of times I heard someone say 'data science'")

# if you don't do this, matplotlib will label the x-axis 0, 1
# and then add a +2.013e3 off in the corner (bad matplotlib!)
plt.ticklabel_format(useOffset=False)

# misleading y-axis only shows the part above 500
plt.axis([2012.5,2014.5,499,506])
plt.title("Look at the 'Huge' Increase!")
plt.show()
```

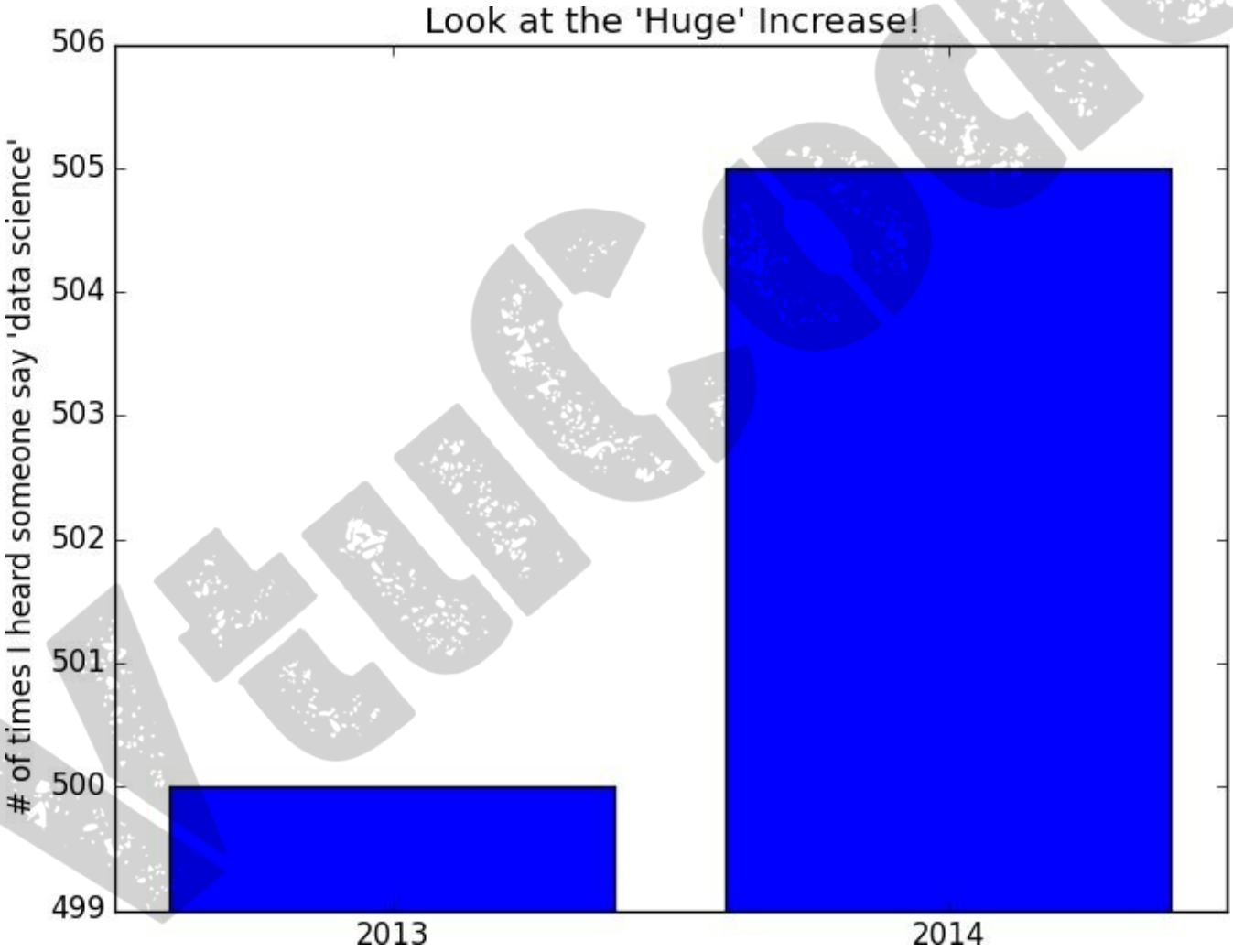

Figure 3-4. A chart with a misleading y-axis

In Figure 3-5, we use more-sensible axes, and it looks far less impressive:

```
plt.axis([2012.5,2014.5,0,550])
plt.title("Not So Huge Anymore")
plt.show()
```

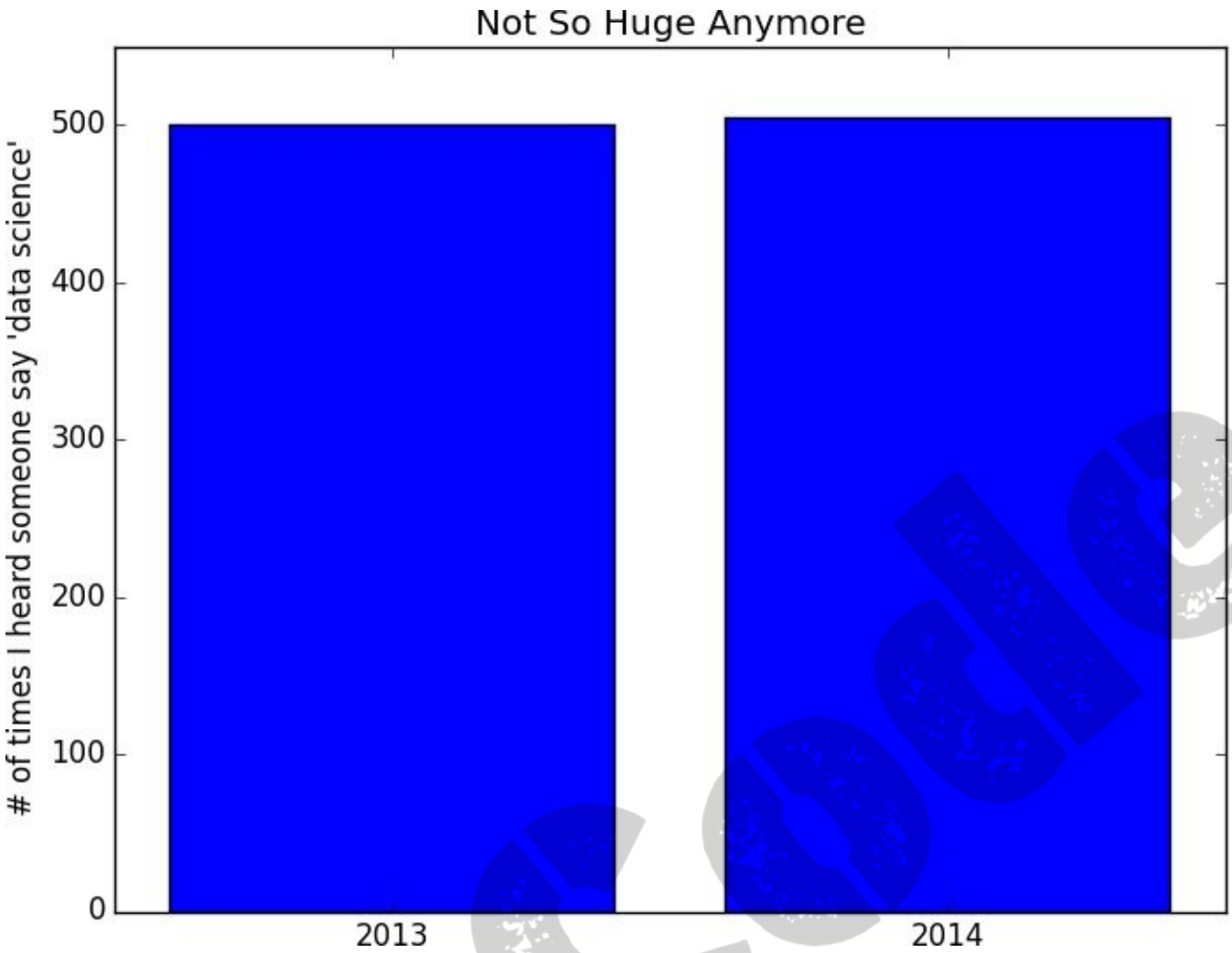

Figure 3-5. The same chart with a nonmisleading y-axis

Line Charts

As we saw already, we can make line charts using `plt.plot()`. These are a good choice for showing *trends*, as illustrated in Figure 3-6:

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# we can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line

# because we've assigned labels to each series
# we can get a legend for free
# loc=9 means "top center"
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff")
plt.show()
```

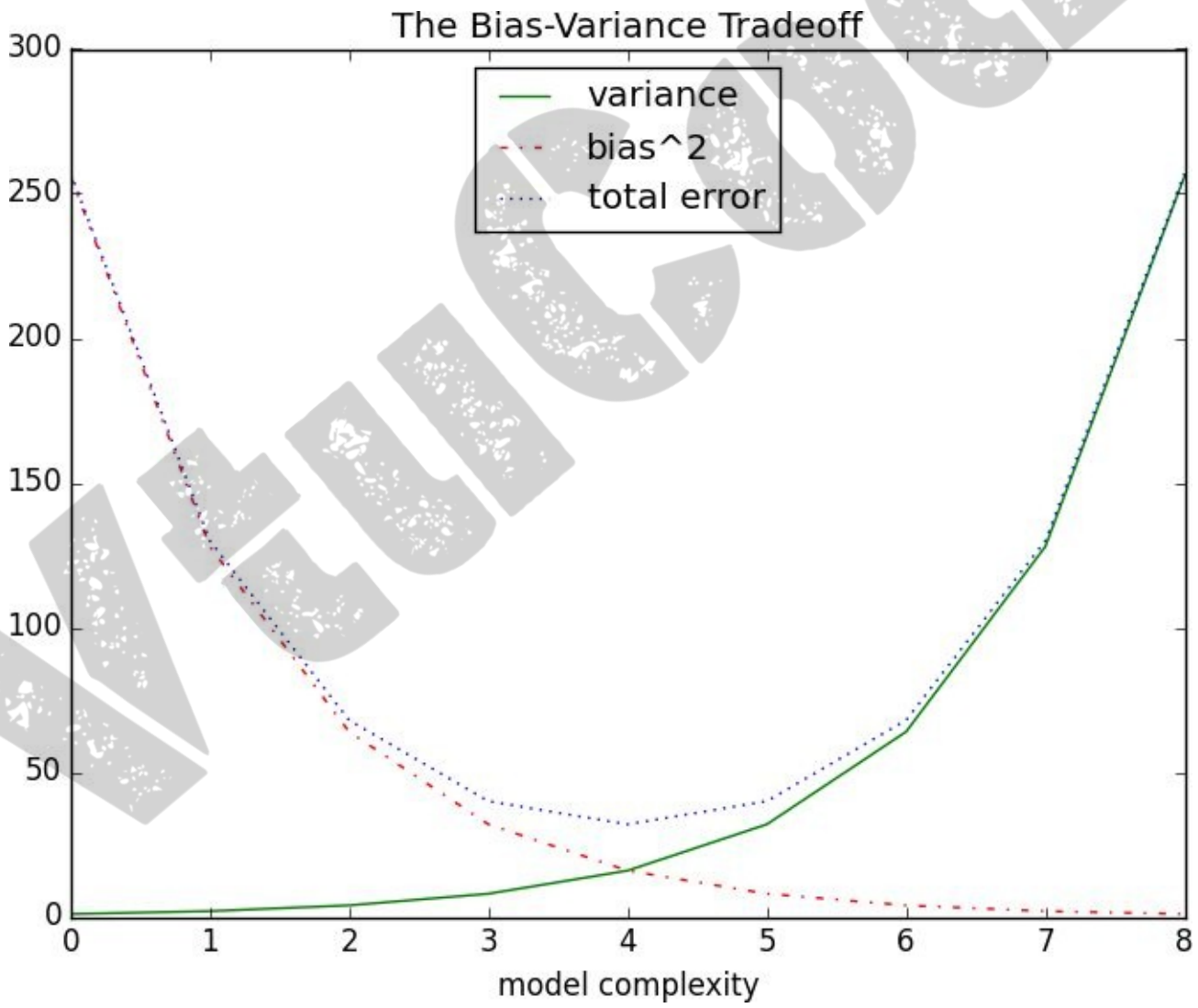

Figure 3-6. Several line charts with a legend

Scatterplots

A scatterplot is the right choice for visualizing the relationship between two paired sets of data. For example, Figure 3-7 illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# label each point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # put the label with its point
        xytext=(5, -5),                  # but slightly offset
        textcoords='offset points')

plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

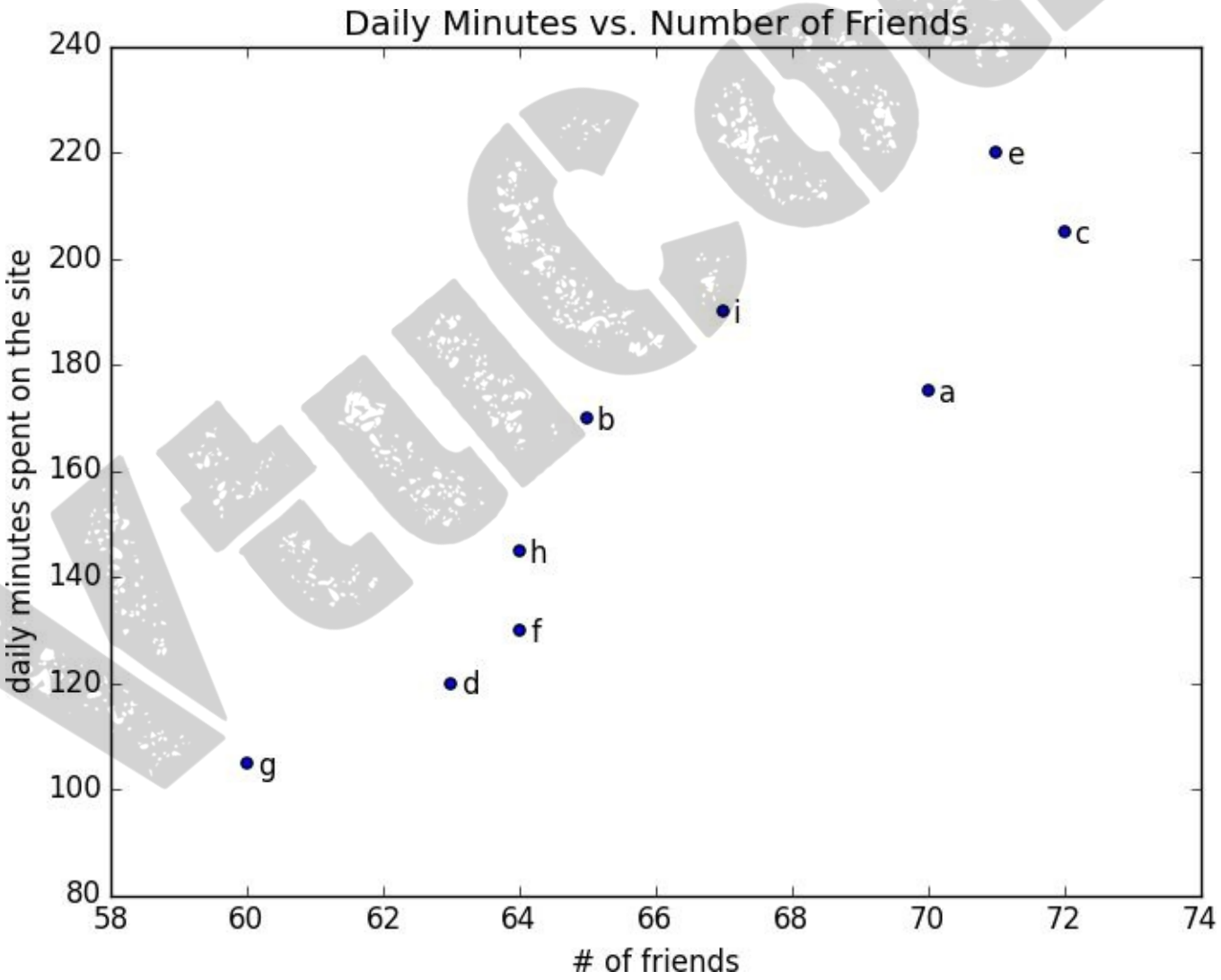

Figure 3-7. A scatterplot of friends and time on the site

If you're scattering comparable variables, you might get a misleading picture if you let `matplotlib` choose the scale, as in Figure 3-8:

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```

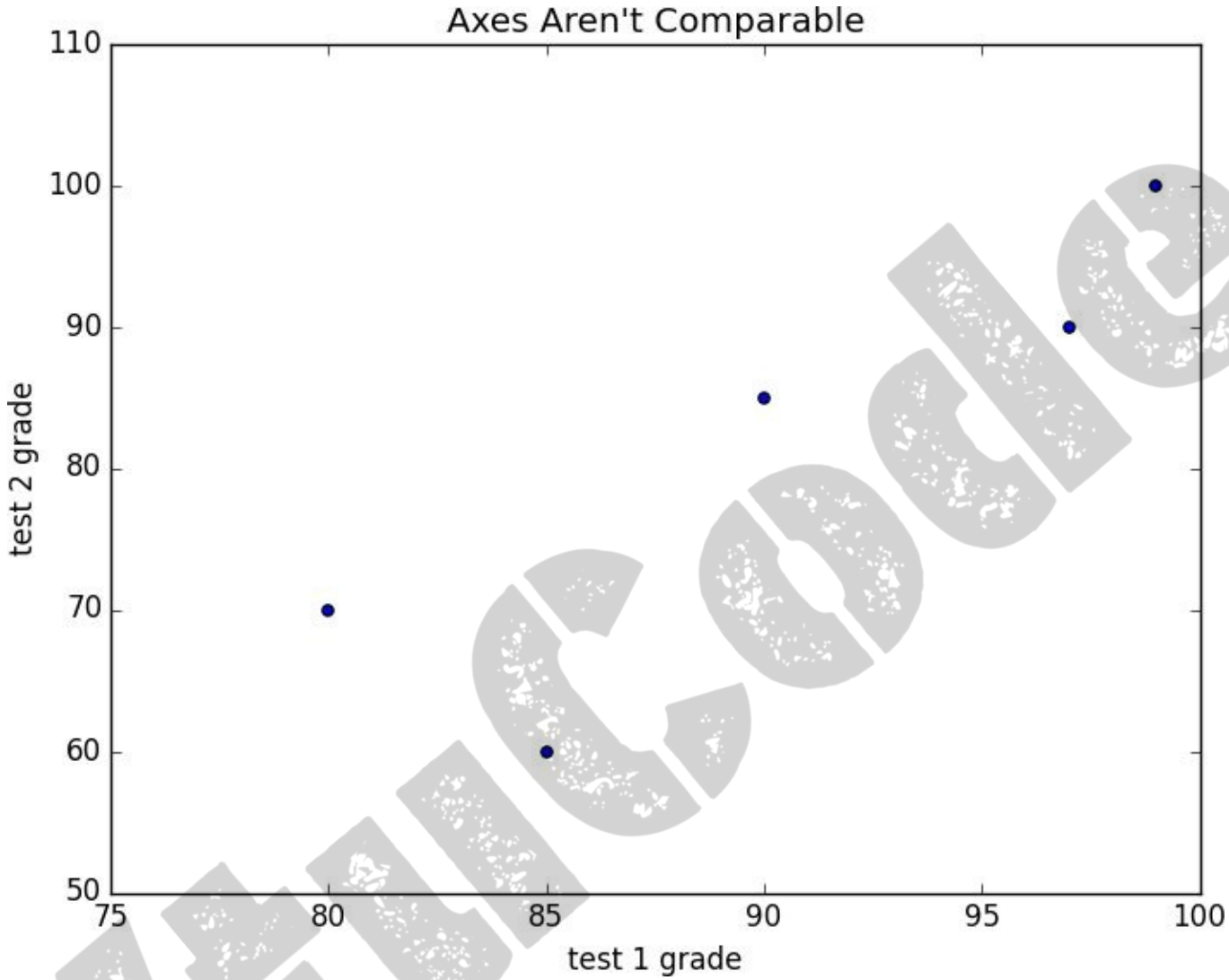

Figure 3-8. A scatterplot with uncomparable axes

If we include a call to `plt.axis("equal")`, the plot (Figure 3-9) more accurately shows that most of the variation occurs on test 2.

That's enough to get you started doing visualization. We'll learn much more about visualization throughout the book.

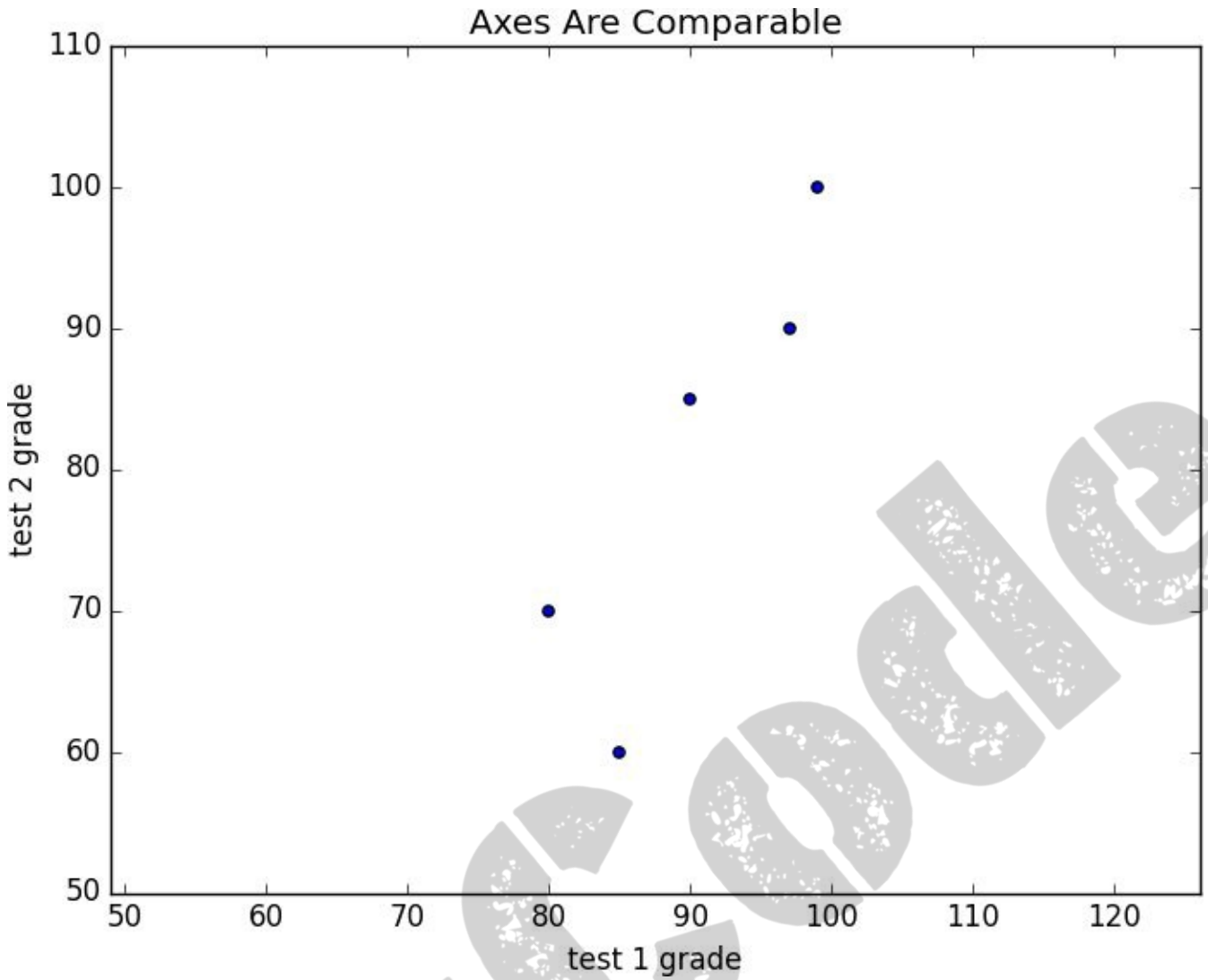

Figure 3-9. The same scatterplot with equal axes

Linear Algebra

> Is there anything more useless or less useful than Algebra?
>
> Billy Connolly

Linear algebra is the branch of mathematics that deals with *vector spaces*. Although I can't hope to teach you linear algebra in a brief chapter, it underpins a large number of data science concepts and techniques, which means I owe it to you to at least try. What we learn in this chapter we'll use heavily throughout the rest of the book.

Vectors

Abstractly, *vectors* are objects that can be added together (to form new vectors) and that can be multiplied by *scalars* (i.e., numbers), also to form new vectors.

Concretely (for us), vectors are points in some finite-dimensional space. Although you might not think of your data as vectors, they are a good way to represent numeric data.

For example, if you have the heights, weights, and ages of a large number of people, you can treat your data as three-dimensional vectors `(height, weight, age)`. If you're teaching a class with four exams, you can treat student grades as four-dimensional vectors `(exam1, exam2, exam3, exam4)`.

The simplest from-scratch approach is to represent vectors as lists of numbers. A list of three numbers corresponds to a vector in three-dimensional space, and vice versa:

```
height_weight_age = [70,  # inches,
                     170, # pounds,
                     40 ] # years

grades = [95,   # exam1
          80,   # exam2
          75,   # exam3
          62 ]  # exam4
```

One problem with this approach is that we will want to perform *arithmetic* on vectors. Because Python lists aren't vectors (and hence provide no facilities for vector arithmetic), we'll need to build these arithmetic tools ourselves. So let's start with that.

To begin with, we'll frequently need to add two vectors. Vectors add *componentwise*. This means that if two vectors `v` and `w` are the same length, their sum is just the vector whose first element is `v[0] + w[0]`, whose second element is `v[1] + w[1]`, and so on. (If they're not the same length, then we're not allowed to add them.)

For example, adding the vectors `[1, 2]` and `[2, 1]` results in `[1 + 2, 2 + 1]` or `[3, 3]`, as shown in Figure 4-1.

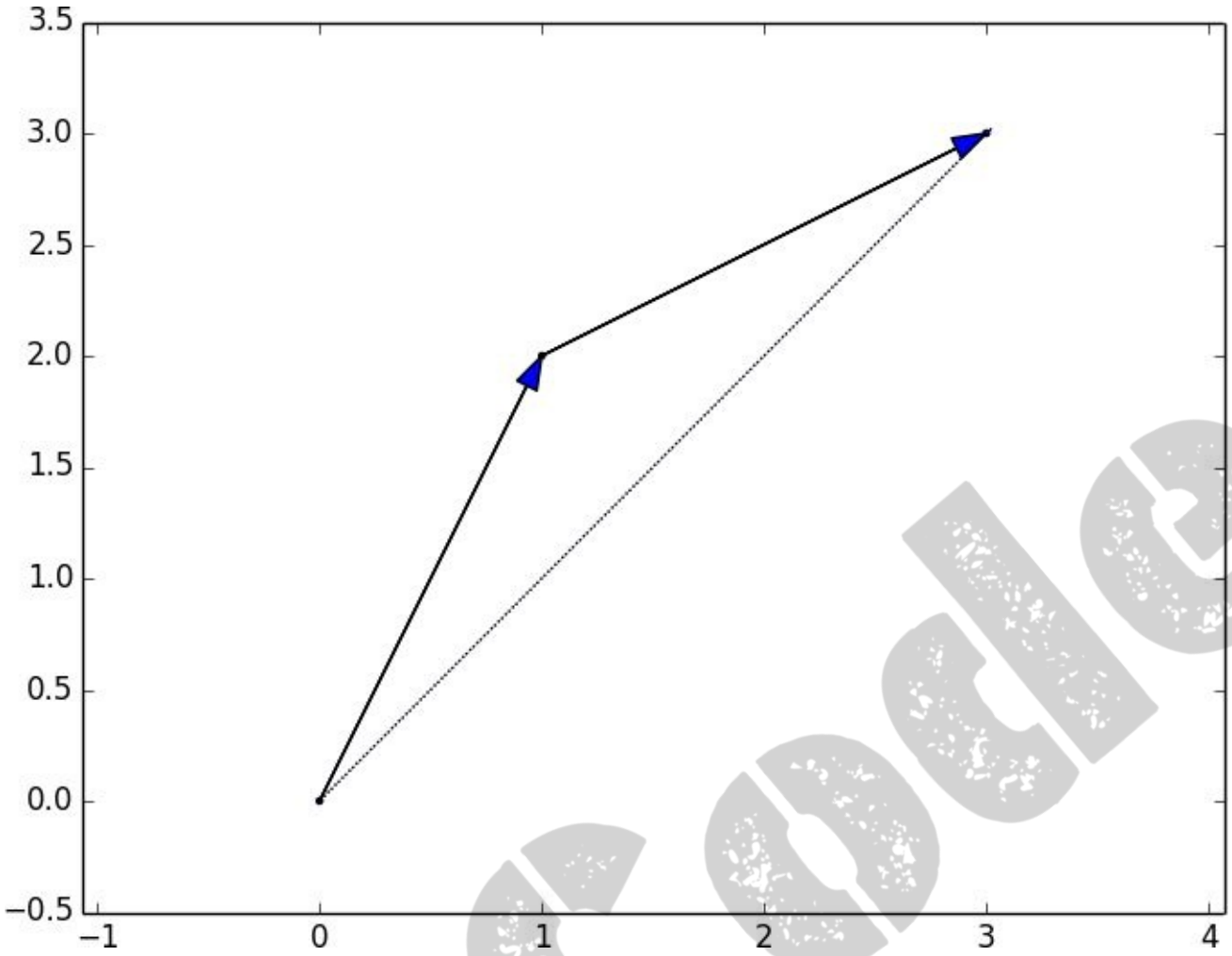

Figure 4-1. Adding two vectors

We can easily implement this by `zip`-ing the vectors together and using a list comprehension to add the corresponding elements:

```
def vector_add(v, w):
    """adds corresponding elements"""
    return [v_i + w_i
            for v_i, w_i in zip(v, w)]
```

Similarly, to subtract two vectors we just subtract corresponding elements:

```
def vector_subtract(v, w):
    """subtracts corresponding elements"""
    return [v_i - w_i
            for v_i, w_i in zip(v, w)]
```

We'll also sometimes want to componentwise sum a list of vectors. That is, create a new vector whose first element is the sum of all the first elements, whose second element is the sum of all the second elements, and so on. The easiest way to do this is by adding one vector at a time:

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0]                    # start with the first vector
    for vector in vectors[1:]:             # then loop over the others
        result = vector_add(result, vector)     # and add them to the result
    return result
```

If you think about it, we are just `reduce`-ing the list of vectors using `vector_add`, which means we can rewrite this more briefly using higher-order functions:

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

or even:

```
vector_sum = partial(reduce, vector_add)
```

although this last one is probably more clever than helpful.

We'll also need to be able to multiply a vector by a scalar, which we do simply by multiplying each element of the vector by that number:

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
```

This allows us to compute the componentwise means of a list of (same-sized) vectors:

```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the
    ith elements of the input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

A less obvious tool is the *dot product*. The dot product of two vectors is the sum of their componentwise products:

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

The dot product measures how far the vector *v* extends in the *w* direction. For example, if `w = [1, 0]` then `dot(v, w)` is just the first component of `v`. Another way of saying this is that it's the length of the vector you'd get if you *projected* *v* onto *w* (Figure 4-2).

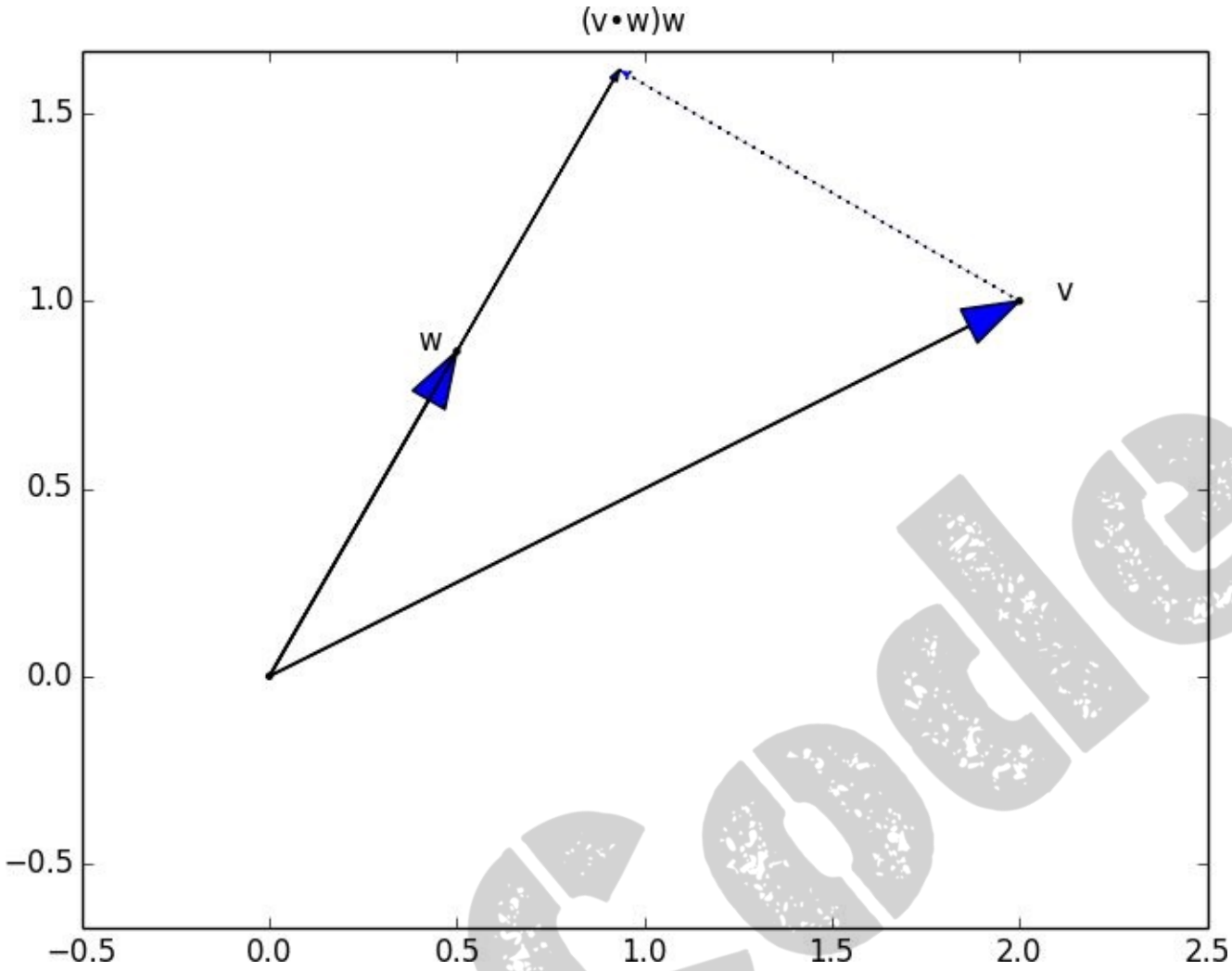

Figure 4-2. The dot product as vector projection

Using this, it's easy to compute a vector's *sum of squares*:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

Which we can use to compute its *magnitude* (or length):

```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))   # math.sqrt is square root function
```

We now have all the pieces we need to compute the distance between two vectors, defined as:

$$\sqrt{(v_1 - w_1)^2 + \ \dots + (v_n - w_n)^2}$$

```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
```

```
    return math.sqrt(squared_distance(v, w))
```

Which is possibly clearer if we write it as (the equivalent):

```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

That should be plenty to get us started. We'll be using these functions heavily throughout the book.

> **NOTE**
>
> Using lists as vectors is great for exposition but terrible for performance.
>
> In production code, you would want to use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included.

Matrices

A *matrix* is a two-dimensional collection of numbers. We will represent matrices as `lists` of `lists`, with each inner list having the same size and representing a *row* of the matrix. If `A` is a matrix, then `A[i][j]` is the element in the *i*th row and the *j*th column. Per mathematical convention, we will typically use capital letters to represent matrices. For example:

```
A = [[1, 2, 3],  # A has 2 rows and 3 columns
     [4, 5, 6]]

B = [[1, 2],     # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
```

> **NOTE**
>
> In mathematics, you would usually name the first row of the matrix "row 1" and the first column "column 1." Because we're representing matrices with Python `lists`, which are zero-indexed, we'll call the first row of a matrix "row 0" and the first column "column 0."

Given this list-of-lists representation, the matrix `A` has `len(A)` rows and `len(A[0])` columns, which we consider its `shape`:

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0   # number of elements in first row
    return num_rows, num_cols
```

If a matrix has *n* rows and *k* columns, we will refer to it as a $n \times k$ matrix. We can (and sometimes will) think of each row of a $n \times k$ matrix as a vector of length *k*, and each column as a vector of length *n*:

```
def get_row(A, i):
    return A[i]             # A[i] is already the ith row

def get_column(A, j):
    return [A_i[j]          # jth element of row A_i
            for A_i in A]   # for each row A_i
```

We'll also want to be able to create a matrix given its shape and a function for generating its elements. We can do this using a nested list comprehension:

```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix
    whose (i,j)th entry is entry_fn(i, j)"""
    return [[entry_fn(i, j)             # given i, create a list
             for j in range(num_cols)]  #   [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # create one list for each i
```

Given this function, you could make a 5 × 5 *identity matrix* (with 1s on the diagonal and 0s elsewhere) with:

```
def is_diagonal(i, j):
```

```
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

# [[1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 0],
#  [0, 0, 0, 0, 1]]
```

Matrices will be important to us for several reasons.

First, we can use a matrix to represent a data set consisting of multiple vectors, simply by considering each vector as a row of the matrix. For example, if you had the heights, weights, and ages of 1,000 people you could put them in a $1,000 \times 3$ matrix:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

Second, as we'll see later, we can use an $n \times k$ matrix to represent a linear function that maps *k*-dimensional vectors to *n*-dimensional vectors. Several of our techniques and concepts will involve such functions.

Third, matrices can be used to represent binary relationships. In Chapter 1, we represented the edges of a network as a collection of pairs `(i, j)`. An alternative representation would be to create a matrix *A* such that `A[i][j]` is 1 if nodes *i* and *j* are connected and 0 otherwise.

Recall that before we had:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

We could also represent this as:

```
#    user 0  1  2  3  4  5  6  7  8  9
#
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # user 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # user 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # user 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # user 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # user 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # user 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # user 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # user 9
```

If there are very few connections, this is a much more inefficient representation, since you end up having to store a lot of zeroes. However, with the matrix representation it is much quicker to check whether two nodes are connected — you just have to do a matrix lookup instead of (potentially) inspecting every edge:

```
friendships[0][2] == 1   # True, 0 and 2 are friends
friendships[0][8] == 1   # False, 0 and 8 are not friends
```

Similarly, to find the connections a node has, you only need to inspect the column (or the row) corresponding to that node:

```
friends_of_five = [i                                          # only need
                   for i, is_friend in enumerate(friendships[5])  # to look at
                   if is_friend]                              # one row
```

Previously we added a list of connections to each node object to speed up this process, but for a large, evolving graph that would probably be too expensive and difficult to maintain.

We'll revisit matrices throughout the book.

Statistics

> Facts are stubborn, but statistics are more pliable.
>
> Mark Twain

Statistics refers to the mathematics and techniques with which we understand data. It is a rich, enormous field, more suited to a shelf (or room) in a library rather than a chapter in a book, and so our discussion will necessarily not be a deep one. Instead, I'll try to teach you just enough to be dangerous, and pique your interest just enough that you'll go off and learn more.

Describing a Single Set of Data

Through a combination of word-of-mouth and luck, DataSciencester has grown to dozens of members, and the VP of Fundraising asks you for some sort of description of how many friends your members have that he can include in his elevator pitches.

Using techniques from Chapter 1, you are easily able to produce this data. But now you are faced with the problem of how to *describe* it.

One obvious description of any data set is simply the data itself:

```
num_friends = [100, 49, 41, 40, 25,
               # ... and lots more
              ]
```

For a small enough data set this might even be the best description. But for a larger data set, this is unwieldy and probably opaque. (Imagine staring at a list of 1 million numbers.) For that reason we use statistics to distill and communicate relevant features of our data.

As a first approach you put the friend counts into a histogram using `Counter` and `plt.bar()` (Figure 5-1):

```
friend_counts = Counter(num_friends)
xs = range(101)                         # largest value is 100
ys = [friend_counts[x] for x in xs]     # height is just # of friends
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()
```

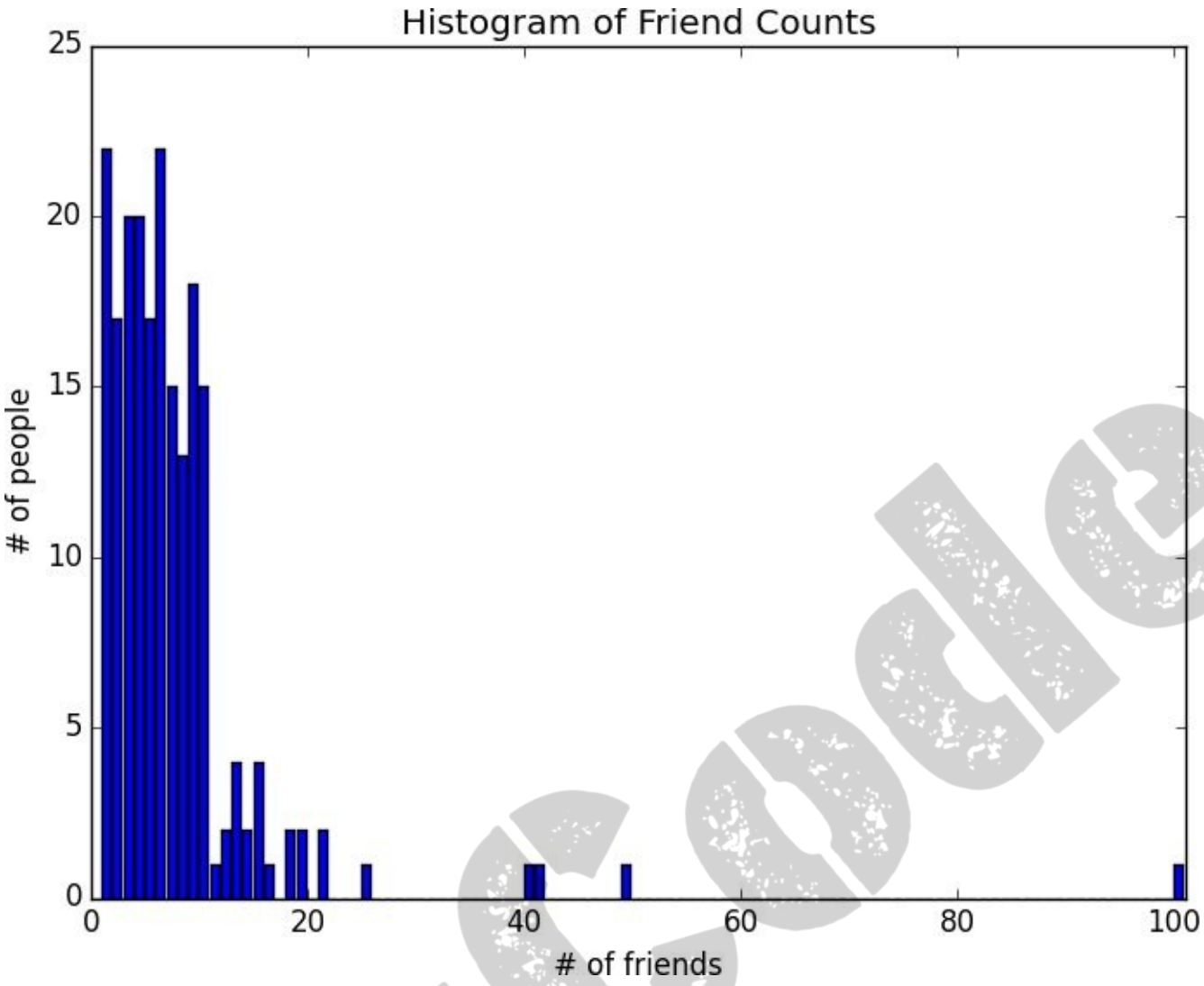

Figure 5-1. A histogram of friend counts

Unfortunately, this chart is still too difficult to slip into conversations. So you start generating some statistics. Probably the simplest statistic is simply the number of data points:

```
num_points = len(num_friends)              # 204
```

You're probably also interested in the largest and smallest values:

```
largest_value = max(num_friends)           # 100
smallest_value = min(num_friends)          # 1
```

which are just special cases of wanting to know the values in specific positions:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]          # 1
second_smallest_value = sorted_values[1]   # 1
second_largest_value = sorted_values[-2]   # 49
```

But we're only getting started.

Central Tendencies

Usually, we'll want some notion of where our data is centered. Most commonly we'll use the *mean* (or average), which is just the sum of the data divided by its count:

```
# this isn't right if you don't from __future__ import division
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

If you have two data points, the mean is simply the point halfway between them. As you add more points, the mean shifts around, but it always depends on the value of every point.

We'll also sometimes be interested in the *median*, which is the middle-most value (if the number of data points is odd) or the average of the two middle-most values (if the number of data points is even).

For instance, if we have five data points in a sorted vector `x`, the median is `x[5 // 2]` or `x[2]`. If we have six data points, we want the average of `x[2]` (the third point) and `x[3]` (the fourth point).

Notice that — unlike the mean — the median doesn't depend on every value in your data. For example, if you make the largest point larger (or the smallest point smaller), the middle points remain unchanged, which means so does the median.

The `median` function is slightly more complicated than you might expect, mostly because of the "even" case:

```
def median(v):
    """finds the 'middle-most' value of v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # if odd, return the middle value
        return sorted_v[midpoint]
    else:
        # if even, return the average of the middle values
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2

median(num_friends) # 6.0
```

Clearly, the mean is simpler to compute, and it varies smoothly as our data changes. If we have *n* data points and one of them increases by some small amount *e*, then necessarily the mean will increase by *e* / *n*. (This makes the mean amenable to all sorts of calculus tricks.) Whereas in order to find the median, we have to sort our data. And changing one of our data points by a small amount *e* might increase the median by *e*, by some number less than *e*, or not at all (depending on the rest of the data).

> **NOTE**
>
> There are, in fact, nonobvious tricks to efficiently compute medians without sorting the data. However, they are beyond the scope of this book, so *we* have to sort the data.

At the same time, the mean is very sensitive to outliers in our data. If our friendliest user had 200 friends (instead of 100), then the mean would rise to 7.82, while the median would stay the same. If outliers are likely to be bad data (or otherwise unrepresentative of whatever phenomenon we're trying to understand), then the mean can sometimes give us a misleading picture. For example, the story is often told that in the mid-1980s, the major at the University of North Carolina with the highest average starting salary was geography, mostly on account of NBA star (and outlier) Michael Jordan.

A generalization of the median is the *quantile*, which represents the value less than which a certain percentile of the data lies. (The median represents the value less than which 50% of the data lies.)

```
def quantile(x, p):
    """returns the pth-percentile value in x"""
    p_index = int(p * len(x))
    return sorted(x)[p_index]

quantile(num_friends, 0.10) # 1
quantile(num_friends, 0.25) # 3
quantile(num_friends, 0.75) # 9
quantile(num_friends, 0.90) # 13
```

Less commonly you might want to look at the *mode*, or most-common value[s]:

```
def mode(x):
    """returns a list, might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.iteritems()
            if count == max_count]

mode(num_friends)       # 1 and 6
```

But most frequently we'll just use the mean.

Dispersion

Dispersion refers to measures of how spread out our data is. Typically they're statistics for which values near zero signify *not spread out at all* and for which large values (whatever that means) signify *very spread out*. For instance, a very simple measure is the *range*, which is just the difference between the largest and smallest elements:

```
# "range" already means something in Python, so we'll use a different name
def data_range(x):
    return max(x) - min(x)

data_range(num_friends) # 99
```

The range is zero precisely when the `max` and `min` are equal, which can only happen if the elements of `x` are all the same, which means the data is as undispersed as possible. Conversely, if the range is large, then the `max` is much larger than the `min` and the data is more spread out.

Like the median, the range doesn't really depend on the whole data set. A data set whose points are all either 0 or 100 has the same range as a data set whose values are 0, 100, and lots of 50s. But it seems like the first data set "should" be more spread out.

A more complex measure of dispersion is the *variance*, which is computed as:

```
def de_mean(x):
    """translate x by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]

def variance(x):
    """assumes x has at least two elements"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)

variance(num_friends) # 81.54
```

> **NOTE**
>
> This looks like it is almost the average squared deviation from the mean, except that we're dividing by `n-1` instead of `n`. In fact, when we're dealing with a sample from a larger population, `x_bar` is only an *estimate* of the actual mean, which means that on average `(x_i - x_bar) ** 2` is an underestimate of `x_i`'s squared deviation from the mean, which is why we divide by `n-1` instead of `n`. See Wikipedia.

Now, whatever units our data is in (e.g., "friends"), all of our measures of central tendency are in that same unit. The range will similarly be in that same unit. The variance, on the other hand, has units that are the *square* of the original units (e.g., "friends squared"). As it can be hard to make sense of these, we often look instead at the *standard deviation*:

```
def standard_deviation(x):
    return math.sqrt(variance(x))

standard_deviation(num_friends) # 9.03
```

Both the range and the standard deviation have the same outlier problem that we saw

earlier for the mean. Using the same example, if our friendliest user had instead 200 friends, the standard deviation would be 14.89, more than 60% higher!

A more robust alternative computes the difference between the 75th percentile value and the 25th percentile value:

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)

interquartile_range(num_friends) # 6
```

which is quite plainly unaffected by a small number of outliers.

Correlation

DataSciencester's VP of Growth has a theory that the amount of time people spend on the site is related to the number of friends they have on the site (she's not a VP for nothing), and she's asked you to verify this.

After digging through traffic logs, you've come up with a list `daily_minutes` that shows how many minutes per day each user spends on DataSciencester, and you've ordered it so that its elements correspond to the elements of our previous `num_friends` list. We'd like to investigate the relationship between these two metrics.

We'll first look at *covariance*, the paired analogue of variance. Whereas variance measures how a single variable deviates from its mean, covariance measures how two variables vary in tandem from their means:

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

Recall that `dot` sums up the products of corresponding pairs of elements. When corresponding elements of `x` and `y` are either both above their means or both below their means, a positive number enters the sum. When one is above its mean and the other below, a negative number enters the sum. Accordingly, a "large" positive covariance means that `x` tends to be large when `y` is large and small when `y` is small. A "large" negative covariance means the opposite — that `x` tends to be small when `y` is large and vice versa. A covariance close to zero means that no such relationship exists.

Nonetheless, this number can be hard to interpret, for a couple of reasons:

- Its units are the product of the inputs' units (e.g., friend-minutes-per-day), which can be hard to make sense of. (What's a "friend-minute-per-day"?)

- If each user had twice as many friends (but the same number of minutes), the covariance would be twice as large. But in a sense the variables would be just as interrelated. Said differently, it's hard to say what counts as a "large" covariance.

For this reason, it's more common to look at the *correlation*, which divides out the standard deviations of both variables:

```
def correlation(x, y):
    stdev_x = standard_deviation(x)
    stdev_y = standard_deviation(y)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(x, y) / stdev_x / stdev_y
    else:
        return 0    # if no variation, correlation is zero

correlation(num_friends, daily_minutes) # 0.25
```

The `correlation` is unitless and always lies between -1 (perfect anti-correlation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

However, one thing we neglected to do was examine our data. Check out Figure 5-2.

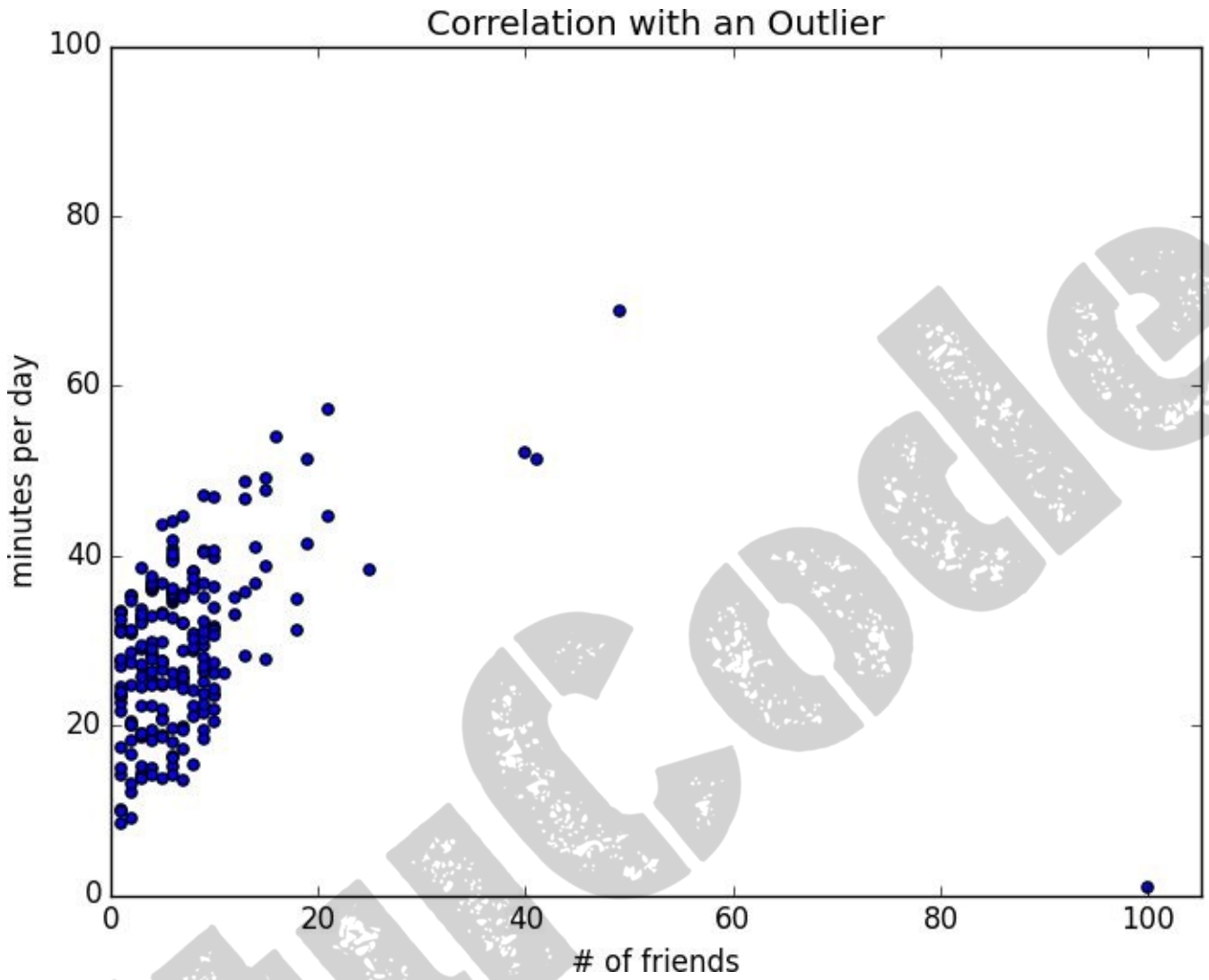

Figure 5-2. Correlation with an outlier

The person with 100 friends (who spends only one minute per day on the site) is a huge outlier, and correlation can be very sensitive to outliers. What happens if we ignore him?

```
outlier = num_friends.index(100)    # index of outlier

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

correlation(num_friends_good, daily_minutes_good) # 0.57
```

Without the outlier, there is a much stronger correlation (Figure 5-3).

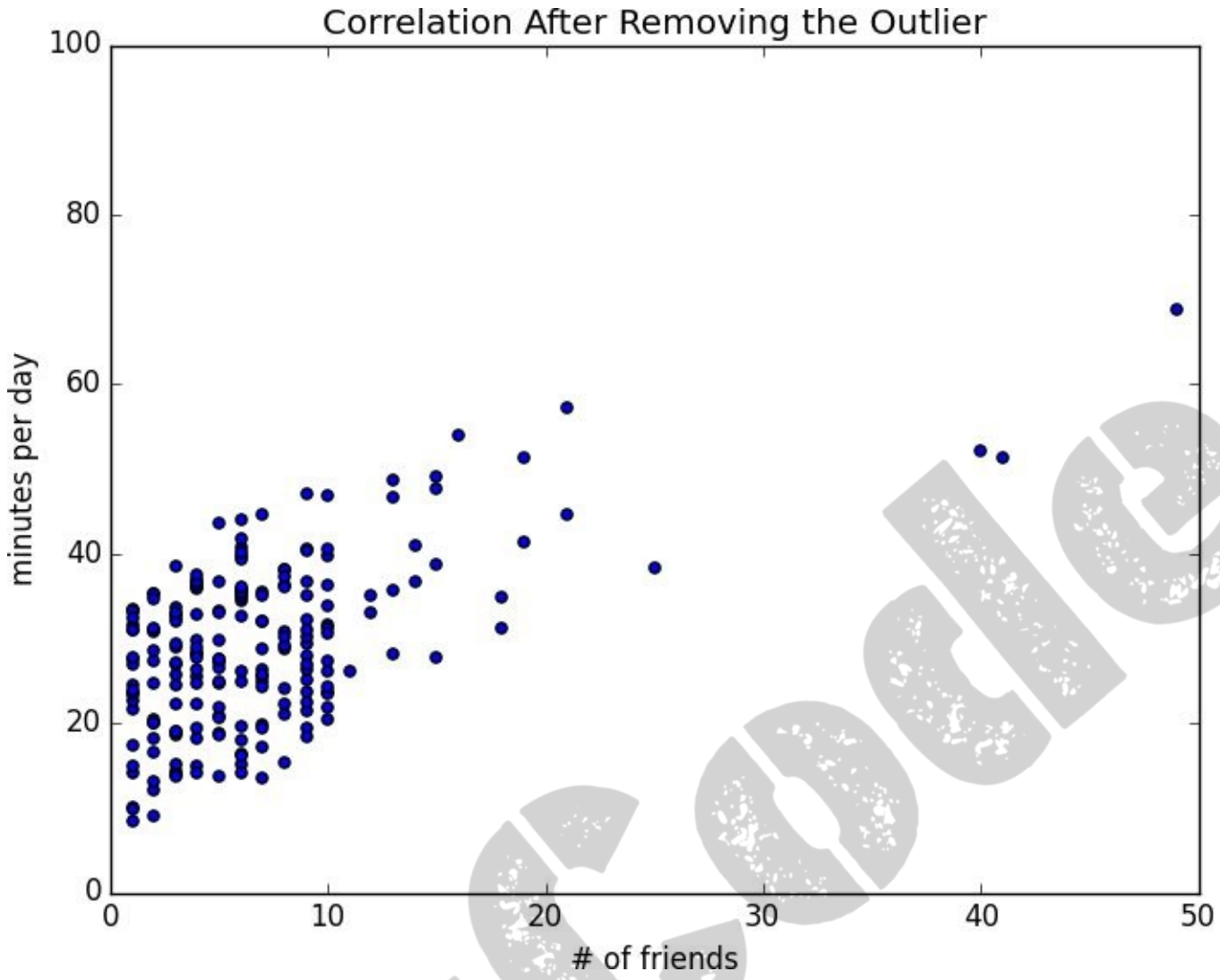

Figure 5-3. Correlation after removing the outlier

You investigate further and discover that the outlier was actually an internal *test* account that no one ever bothered to remove. So you feel pretty justified in excluding it.

Simpson's Paradox

One not uncommon surprise when analyzing data is Simpson's Paradox, in which correlations can be misleading when *confounding* variables are ignored.

For example, imagine that you can identify all of your members as either East Coast data scientists or West Coast data scientists. You decide to examine which coast's data scientists are friendlier:

coast	# of members	avg. # of friends
West Coast	101	8.2
East Coast	103	6.5

It certainly looks like the West Coast data scientists are friendlier than the East Coast data scientists. Your coworkers advance all sorts of theories as to why this might be: maybe it's the sun, or the coffee, or the organic produce, or the laid-back Pacific vibe?

When playing with the data you discover something very strange. If you only look at people with PhDs, the East Coast data scientists have more friends on average. And if you only look at people without PhDs, the East Coast data scientists also have more friends on average!

coast	degree	# of members	avg. # of friends
West Coast	PhD	35	3.1
East Coast	PhD	70	3.2
West Coast	no PhD	66	10.9
East Coast	no PhD	33	13.4

Once you account for the users' degrees, the correlation goes in the opposite direction! Bucketing the data as East Coast/West Coast disguised the fact that the East Coast data scientists skew much more heavily toward PhD types.

This phenomenon crops up in the real world with some regularity. The key issue is that correlation is measuring the relationship between your two variables *all else being equal*. If your data classes are assigned at random, as they might be in a well-designed experiment, "all else being equal" might not be a terrible assumption. But when there is a deeper pattern to class assignments, "all else being equal" can be an awful assumption.

The only real way to avoid this is by *knowing your data* and by doing what you can to make sure you've checked for possible confounding factors. Obviously, this is not always possible. If you didn't have the educational attainment of these 200 data scientists, you

might simply conclude that there was something inherently more sociable about the West Coast.

Some Other Correlational Caveats

A correlation of zero indicates that there is no linear relationship between the two variables. However, there may be other sorts of relationships. For example, if:

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

then `x` and `y` have zero correlation. But they certainly have a relationship — each element of `y` equals the absolute value of the corresponding element of `x`. What they don't have is a relationship in which knowing how `x_i` compares to `mean(x)` gives us information about how `y_i` compares to `mean(y)`. That is the sort of relationship that correlation looks for.

In addition, correlation tells you nothing about how large the relationship is. The variables:

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

are perfectly correlated, but (depending on what you're measuring) it's quite possible that this relationship isn't all that interesting.

Correlation and Causation

You have probably heard at some point that "correlation is not causation," most likely by someone looking at data that posed a challenge to parts of his worldview that he was reluctant to question. Nonetheless, this is an important point — if `x` and `y` are strongly correlated, that might mean that `x` causes `y`, that `y` causes `x`, that each causes the other, that some third factor causes both, or it might mean nothing.

Consider the relationship between `num_friends` and `daily_minutes`. It's possible that having more friends on the site *causes* DataSciencester users to spend more time on the site. This might be the case if each friend posts a certain amount of content each day, which means that the more friends you have, the more time it takes to stay current with their updates.

However, it's also possible that the more time you spend arguing in the DataSciencester forums, the more you encounter and befriend like-minded people. That is, spending more time on the site *causes* users to have more friends.

A third possibility is that the users who are most passionate about data science spend more time on the site (because they find it more interesting) and more actively collect data science friends (because they don't want to associate with anyone else).

One way to feel more confident about causality is by conducting randomized trials. If you can randomly split your users into two groups with similar demographics and give one of the groups a slightly different experience, then you can often feel pretty good that the different experiences are causing the different outcomes.

For instance, if you don't mind being angrily accused of experimenting on your users, you could randomly choose a subset of your users and show them content from only a fraction of their friends. If this subset subsequently spent less time on the site, this would give you some confidence that having more friends *causes* more time on the site.

Probability

> The laws of probability, so true in general, so fallacious in particular.
>
> Edward Gibbon

It is hard to do data science without some sort of understanding of *probability* and its mathematics. As with our treatment of statistics in Chapter 5, we'll wave our hands a lot and elide many of the technicalities.

For our purposes you should think of probability as a way of quantifying the uncertainty associated with *events* chosen from a some *universe* of events. Rather than getting technical about what these terms mean, think of rolling a die. The universe consists of all possible outcomes. And any subset of these outcomes is an event; for example, "the die rolls a one" or "the die rolls an even number."

Notationally, we write $P(E)$ to mean "the probability of the event *E*."

We'll use probability theory to build models. We'll use probability theory to evaluate models. We'll use probability theory all over the place.

One could, were one so inclined, get really deep into the philosophy of what probability theory *means*. (This is best done over beers.) We won't be doing that.

Dependence and Independence

Roughly speaking, we say that two events E and F are *dependent* if knowing something about whether E happens gives us information about whether F happens (and vice versa). Otherwise they are *independent*.

For instance, if we flip a fair coin twice, knowing whether the first flip is Heads gives us no information about whether the second flip is Heads. These events are independent. On the other hand, knowing whether the first flip is Heads certainly gives us information about whether both flips are Tails. (If the first flip is Heads, then definitely it's not the case that both flips are Tails.) These two events are dependent.

Mathematically, we say that two events E and F are independent if the probability that they both happen is the product of the probabilities that each one happens:

$$P(E, F) = P(E)P(F)$$

In the example above, the probability of "first flip Heads" is 1/2, and the probability of "both flips Tails" is 1/4, but the probability of "first flip Heads *and* both flips Tails" is 0.

Conditional Probability

When two events E and F are independent, then by definition we have:

$$P(E, F) = P(E)P(F)$$

If they are not necessarily independent (and if the probability of F is not zero), then we define the probability of E "conditional on F" as:

$$P(E \mid F) = P(E, F) / P(F)$$

You should think of this as the probability that E happens, given that we know that F happens.

We often rewrite this as:

$$P(E, F) = P(E \mid F)P(F)$$

When E and F are independent, you can check that this gives:

$$P(E \mid F) = P(E)$$

which is the mathematical way of expressing that knowing F occurred gives us no additional information about whether E occurred.

One common tricky example involves a family with two (unknown) children.

If we assume that:

1. Each child is equally likely to be a boy or a girl
2. The gender of the second child is independent of the gender of the first child

then the event "no girls" has probability 1/4, the event "one girl, one boy" has probability 1/2, and the event "two girls" has probability 1/4.

Now we can ask what is the probability of the event "both children are girls" (B) conditional on the event "the older child is a girl" (G)? Using the definition of conditional probability:

$$P(B \mid G) = P(B, G) / P(G) = P(B) / P(G) = 1 / 2$$

since the event B and G ("both children are girls *and* the older child is a girl") is just the event B. (Once you know that both children are girls, it's necessarily true that the older child is a girl.)

Most likely this result accords with your intuition.

We could also ask about the probability of the event “both children are girls” conditional on the event “at least one of the children is a girl” (L). Surprisingly, the answer is different from before!

As before, the event B and L (“both children are girls *and* at least one of the children is a girl”) is just the event B. This means we have:

$$P(B \mid L) = P(B, L) / P(L) = P(B) / P(L) = 1 / 3$$

How can this be the case? Well, if all you know is that at least one of the children is a girl, then it is twice as likely that the family has one boy and one girl than that it has both girls.

We can check this by “generating” a lot of families:

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1

print "P(both | older):", both_girls / older_girl      # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl   # 0.342 ~ 1/3
```

Bayes's Theorem

One of the data scientist's best friends is Bayes's Theorem, which is a way of "reversing" conditional probabilities. Let's say we need to know the probability of some event *E* conditional on some other event *F* occurring. But we only have information about the probability of *F* conditional on *E* occurring. Using the definition of conditional probability twice tells us that:

$$P(E \mid F) = P(E, F) / P(F) = P(F \mid E)P(E) / P(F)$$

The event *F* can be split into the two mutually exclusive events "*F* and *E*" and "*F* and not *E*." If we write $\neg E$ for "not *E*" (i.e., "*E* doesn't happen"), then:

$$P(F) = P(F, E) + P(F, \neg E)$$

so that:

$$P(E \mid F) = P(F \mid E)P(E) / [P(F \mid E)P(E) + P(F \mid \neg E)P(\neg E)]$$

which is how Bayes's Theorem is often stated.

This theorem often gets used to demonstrate why data scientists are smarter than doctors. Imagine a certain disease that affects 1 in every 10,000 people. And imagine that there is a test for this disease that gives the correct result ("diseased" if you have the disease, "nondiseased" if you don't) 99% of the time.

What does a positive test mean? Let's use *T* for the event "your test is positive" and *D* for the event "you have the disease." Then Bayes's Theorem says that the probability that you have the disease, conditional on testing positive, is:

$$P(D \mid T) = P(T \mid D)P(D) / [P(T \mid D)P(D) + P(T \mid \neg D)P(\neg D)]$$

Here we know that $P(T \mid D)$, the probability that someone with the disease tests positive, is 0.99. $P(D)$, the probability that any given person has the disease, is 1/10,000 = 0.0001. $P(T \mid \neg D)$, the probability that someone without the disease tests positive, is 0.01. And $P(\neg D)$, the probability that any given person doesn't have the disease, is 0.9999. If you substitute these numbers into Bayes's Theorem you find

$$P(D \mid T) = 0.98\ \%$$

That is, less than 1% of the people who test positive actually have the disease.

> **NOTE**
>
> This assumes that people take the test more or less at random. If only people with certain symptoms take the test we would instead have to condition on the event "positive test *and* symptoms" and the number would likely be a lot higher.

While this is a simple calculation for a data scientist, most doctors will guess that $P(D \mid T)$ is approximately 2.

A more intuitive way to see this is to imagine a population of 1 million people. You'd expect 100 of them to have the disease, and 99 of those 100 to test positive. On the other hand, you'd expect 999,900 of them not to have the disease, and 9,999 of those to test positive. Which means that you'd expect only 99 out of (99 + 9999) positive testers to actually have the disease.

Random Variables

A *random variable* is a variable whose possible values have an associated probability distribution. A very simple random variable equals 1 if a coin flip turns up heads and 0 if the flip turns up tails. A more complicated one might measure the number of heads observed when flipping a coin 10 times or a value picked from `range(10)` where each number is equally likely.

The associated distribution gives the probabilities that the variable realizes each of its possible values. The coin flip variable equals 0 with probability 0.5 and 1 with probability 0.5. The `range(10)` variable has a distribution that assigns probability 0.1 to each of the numbers from 0 to 9.

We will sometimes talk about the *expected value* of a random variable, which is the average of its values weighted by their probabilities. The coin flip variable has an expected value of 1/2 (= 0 * 1/2 + 1 * 1/2), and the `range(10)` variable has an expected value of 4.5.

Random variables can be *conditioned* on events just as other events can. Going back to the two-child example from "Conditional Probability", if X is the random variable representing the number of girls, X equals 0 with probability 1/4, 1 with probability 1/2, and 2 with probability 1/4.

We can define a new random variable Y that gives the number of girls conditional on at least one of the children being a girl. Then Y equals 1 with probability 2/3 and 2 with probability 1/3. And a variable Z that's the number of girls conditional on the older child being a girl equals 1 with probability 1/2 and 2 with probability 1/2.

For the most part, we will be using random variables *implicitly* in what we do without calling special attention to them. But if you look deeply you'll see them.

Continuous Distributions

A coin flip corresponds to a *discrete distribution* — one that associates positive probability with discrete outcomes. Often we'll want to model distributions across a continuum of outcomes. (For our purposes, these outcomes will always be real numbers, although that's not always the case in real life.) For example, the *uniform distribution* puts *equal weight* on all the numbers between 0 and 1.

Because there are infinitely many numbers between 0 and 1, this means that the weight it assigns to individual points must necessarily be zero. For this reason, we represent a continuous distribution with a *probability density function* (pdf) such that the probability of seeing a value in a certain interval equals the integral of the density function over the interval.

> **NOTE**
>
> If your integral calculus is rusty, a simpler way of understanding this is that if a distribution has density function f, then the probability of seeing a value between x and $x + h$ is approximately $h * f(x)$ if h is small.

The density function for the uniform distribution is just:

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

The probability that a random variable following that distribution is between 0.2 and 0.3 is 1/10, as you'd expect. Python's `random.random()` is a [pseudo]random variable with a uniform density.

We will often be more interested in the *cumulative distribution function* (cdf), which gives the probability that a random variable is less than or equal to a certain value. It's not hard to create the cumulative distribution function for the uniform distribution (Figure 6-1):

```
def uniform_cdf(x):
    "returns the probability that a uniform random variable is <= x"
    if x < 0:   return 0    # uniform random is never less than 0
    elif x < 1: return x    # e.g. P(X <= 0.4) = 0.4
    else:       return 1    # uniform random is always less than 1
```

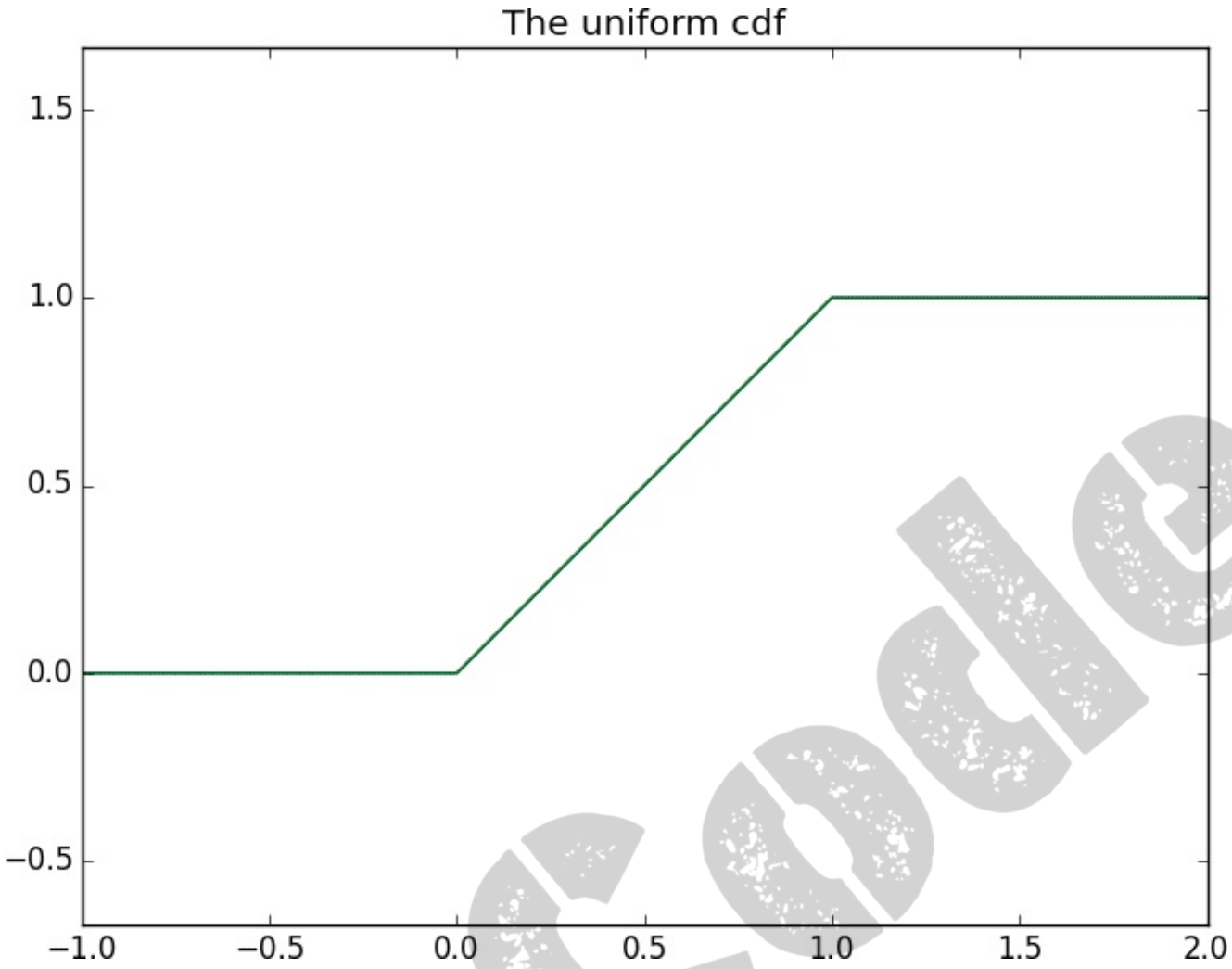

Figure 6-1. The uniform cdf

The Normal Distribution

The normal distribution is the king of distributions. It is the classic bell curve–shaped distribution and is completely determined by two parameters: its mean μ (mu) and its standard deviation σ (sigma). The mean indicates where the bell is centered, and the standard deviation how "wide" it is.

It has the distribution function:

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

which we can implement as:

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

In Figure 6-2, we plot some of these pdfs to see what they look like:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

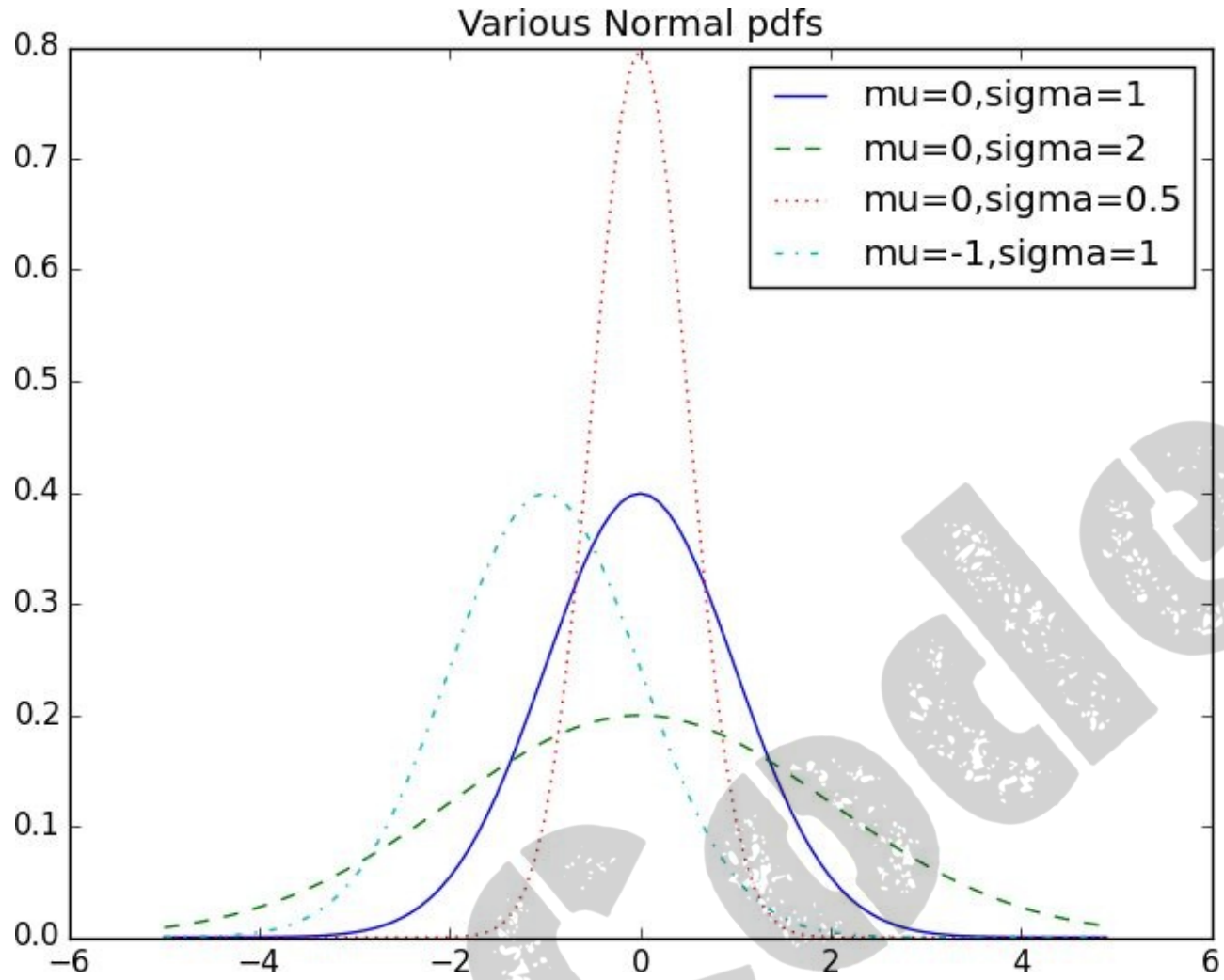

Figure 6-2. Various normal pdfs

When $\mu = 0$ and $\sigma = 1$, it's called the *standard normal distribution*. If *Z* is a standard normal random variable, then it turns out that:

$$X = \sigma Z + \mu$$

is also normal but with mean μ and standard deviation σ. Conversely, if *X* is a normal random variable with mean μ and standard deviation σ,

$$Z = (X - \mu) / \sigma$$

is a standard normal variable.

The cumulative distribution function for the normal distribution cannot be written in an "elementary" manner, but we can write it using Python's `math.erf`:

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Again, in Figure 6-3, we plot a few:

```
xs = [x / 10.0 for x in range(-50, 50)]
```

```
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```

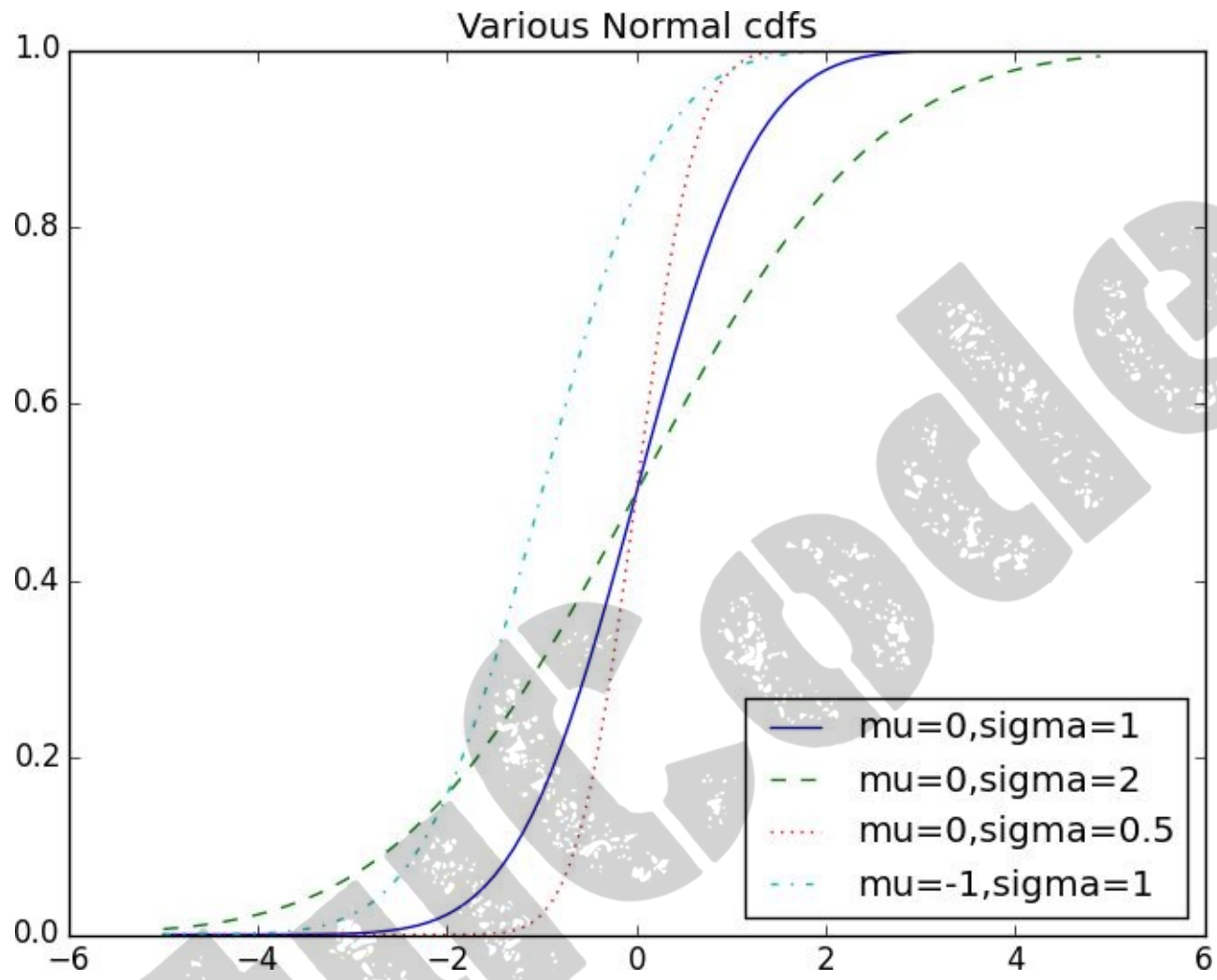

Figure 6-3. Various normal cdfs

Sometimes we'll need to invert `normal_cdf` to find the value corresponding to a specified probability. There's no simple way to compute its inverse, but `normal_cdf` is continuous and strictly increasing, so we can use a binary search:

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """find approximate inverse using binary search"""

    # if not standard, compute standard and rescale
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z, low_p = -10.0, 0            # normal_cdf(-10) is (very close to) 0
    hi_z,  hi_p  =  10.0, 1            # normal_cdf(10)  is (very close to) 1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2     # consider the midpoint
        mid_p = normal_cdf(mid_z)      # and the cdf's value there
        if mid_p < p:
            # midpoint is still too low, search above it
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            # midpoint is still too high, search below it
            hi_z, hi_p = mid_z, mid_p
        else:
```

```
            break
    return mid_z
```

The function repeatedly bisects intervals until it narrows in on a Z that's close enough to the desired probability.

The Central Limit Theorem

One reason the normal distribution is so useful is the *central limit theorem*, which says (in essence) that a random variable defined as the average of a large number of independent and identically distributed random variables is itself approximately normally distributed.

In particular, if $x_1, \ldots, x_n$ are random variables with mean μ and standard deviation σ, and if *n* is large, then:

$$\frac{1}{n}(x_1 + \ldots + x_n)$$

is approximately normally distributed with mean μ and standard deviation $\sigma / \sqrt{n}$. Equivalently (but often more usefully),

$$\frac{(x_1 + \ldots + x_n) - \mu n}{\sigma\sqrt{n}}$$

is approximately normally distributed with mean 0 and standard deviation 1.

An easy way to illustrate this is by looking at *binomial* random variables, which have two parameters *n* and *p*. A Binomial(n,p) random variable is simply the sum of *n* independent Bernoulli(p) random variables, each of which equals 1 with probability *p* and 0 with probability $1 - p$:

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0

def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

The mean of a Bernoulli(p) variable is *p*, and its standard deviation is $\sqrt{p(1-p)}$. The central limit theorem says that as *n* gets large, a Binomial(n,p) variable is approximately a normal random variable with mean $\mu = np$ and standard deviation $\sigma = \sqrt{np(1-p)}$. If we plot both, you can easily see the resemblance:

```
def make_hist(p, n, num_points):

    data = [binomial(n, p) for _ in range(num_points)]

    # use a bar chart to show the actual binomial samples
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))
```

```
    # use a line chart to show the normal approximation
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("Binomial Distribution vs. Normal Approximation")
    plt.show()
```

For example, when you call `make_hist(0.75, 100, 10000)`, you get the graph in Figure 6-4.

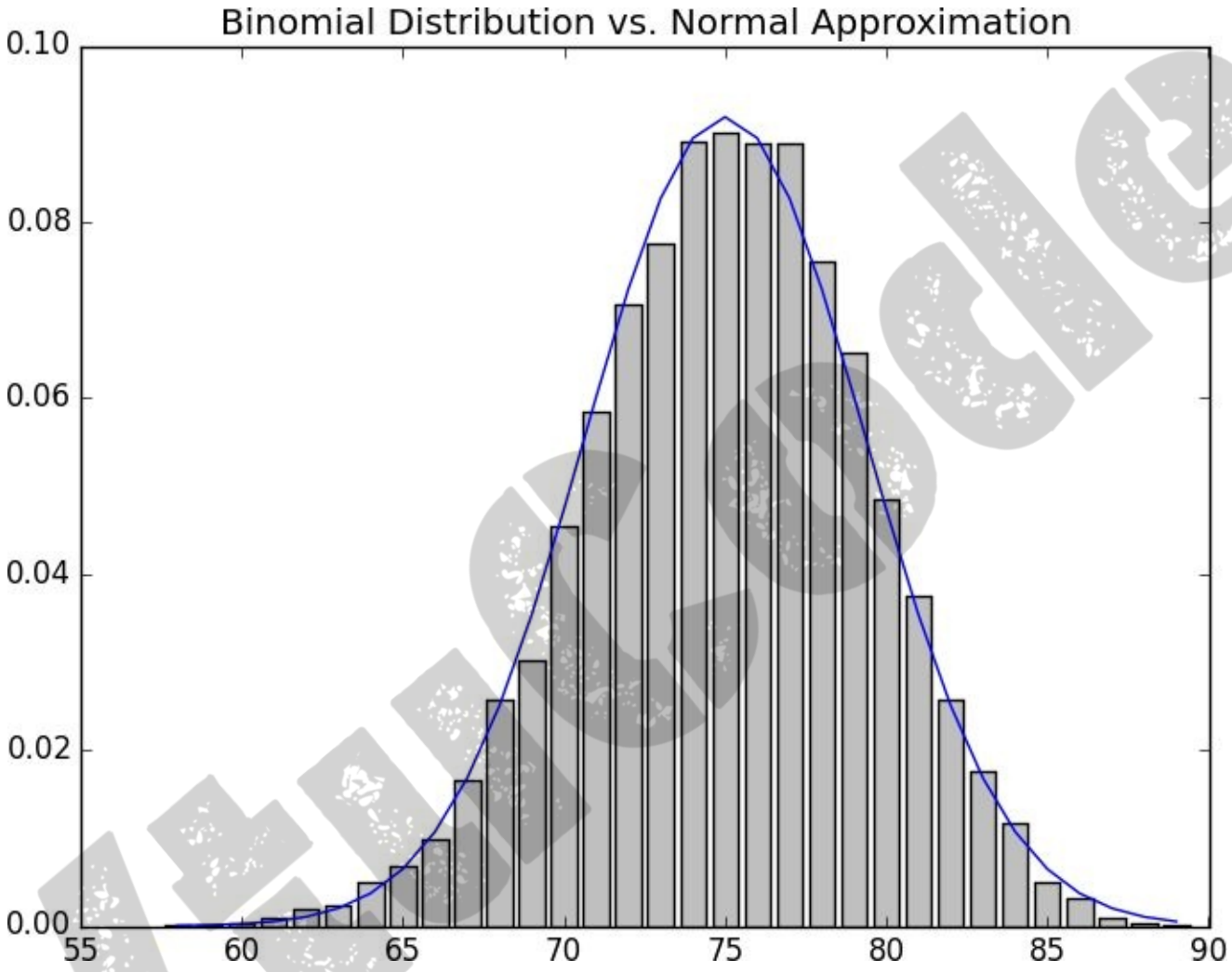

Figure 6-4. The output from `make_hist`

The moral of this approximation is that if you want to know the probability that (say) a fair coin turns up more than 60 heads in 100 flips, you can estimate it as the probability that a Normal(50,5) is greater than 60, which is easier than computing the Binomial(100,0.5) cdf. (Although in most applications you'd probably be using statistical software that would gladly compute whatever probabilities you want.)