Decision Trees

A tree is an incomprehensible mystery.

Jim Woodring

DataSciencester's VP of Talent has interviewed a number of job candidates from the site, with varying degrees of success. He's collected a data set consisting of several (qualitative) attributes of each candidate, as well as whether that candidate interviewed well or poorly. Could you, he asks, use this data to build a model identifying which candidates will interview well, so that he doesn't have to waste time conducting interviews?

This seems like a good fit for a *decision tree*, another predictive modeling tool in the data scientist's kit.

What Is a Decision Tree?

A decision tree uses a tree structure to represent a number of possible *decision paths* and an outcome for each path.

If you have ever played the game Twenty Questions, then it turns out you are familiar with decision trees. For example:

- “I am thinking of an animal.”
- “Does it have more than five legs?”
- “No.”
- “Is it delicious?”
- “No.”
- “Does it appear on the back of the Australian five-cent coin?”
- “Yes.”
- “Is it an echidna?”
- “Yes, it is!”

This corresponds to the path:

“Not more than 5 legs” → “Not delicious” → “On the 5-cent coin” → “Echidna!”

in an idiosyncratic (and not very comprehensive) “guess the animal” decision tree (Figure 17-1).

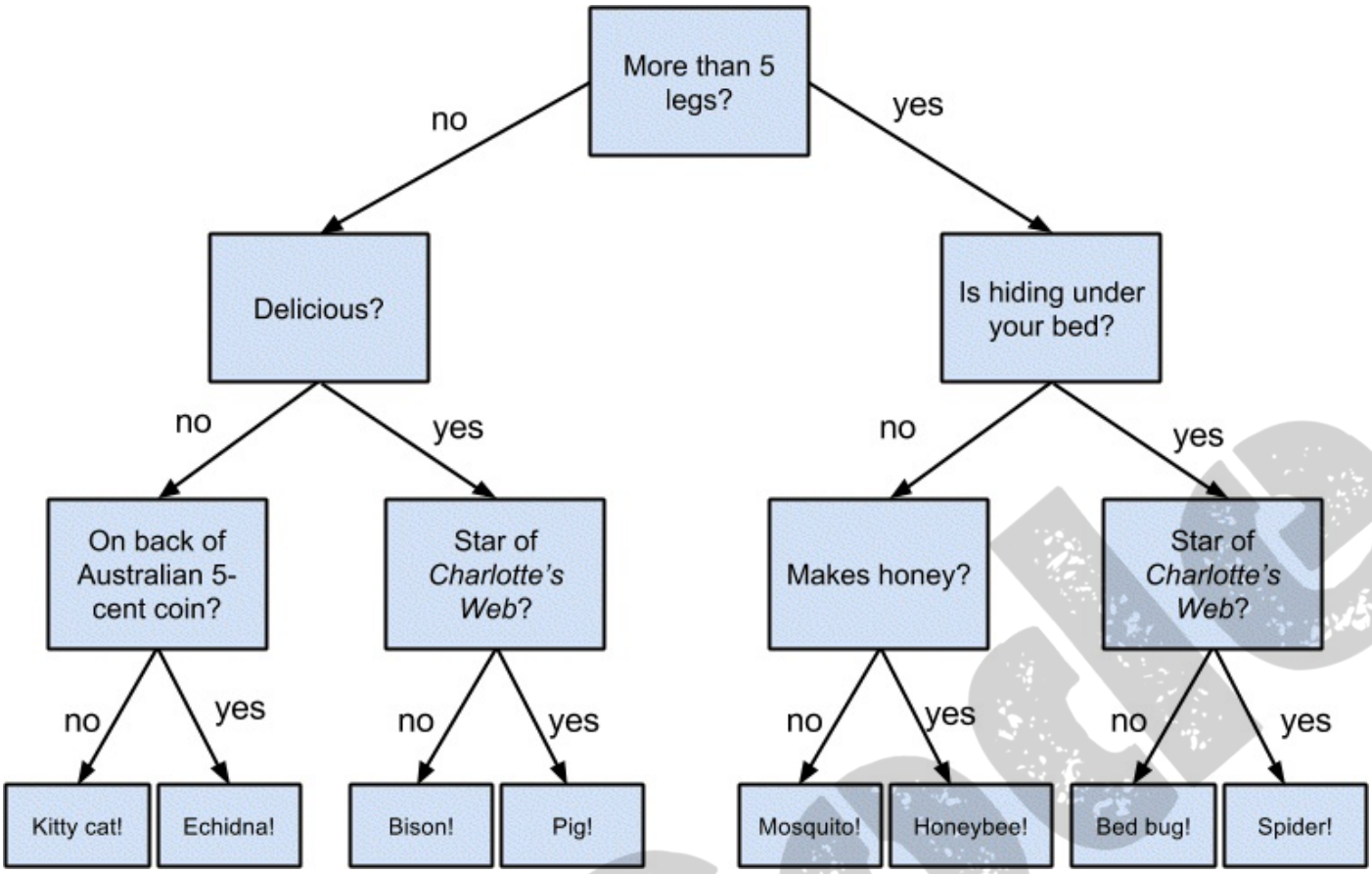

Figure 17-1. A "guess the animal" decision tree

Decision trees have a lot to recommend them. They're very easy to understand and interpret, and the process by which they reach a prediction is completely transparent. Unlike the other models we've looked at so far, decision trees can easily handle a mix of numeric (e.g., number of legs) and categorical (e.g., delicious/not delicious) attributes and can even classify data for which attributes are missing.

At the same time, finding an "optimal" decision tree for a set of training data is computationally a very hard problem. (We will get around this by trying to build a good-enough tree rather than an optimal one, although for large data sets this can still be a lot of work.) More important, it is very easy (and very bad) to build decision trees that are *overfitted* to the training data, and that don't generalize well to unseen data. We'll look at ways to address this.

Most people divide decision trees into *classification trees* (which produce categorical outputs) and *regression trees* (which produce numeric outputs). In this chapter, we'll focus on classification trees, and we'll work through the ID3 algorithm for learning a decision tree from a set of labeled data, which should help us understand how decision trees actually work. To make things simple, we'll restrict ourselves to problems with binary outputs like "should I hire this candidate?" or "should I show this website visitor advertisement A or advertisement B?" or "will eating this food I found in the office fridge make me sick?"

Entropy

In order to build a decision tree, we will need to decide what questions to ask and in what order. At each stage of the tree there are some possibilities we've eliminated and some that we haven't. After learning that an animal doesn't have more than five legs, we've eliminated the possibility that it's a grasshopper. We haven't eliminated the possibility that it's a duck. Every possible question partitions the remaining possibilities according to their answers.

Ideally, we'd like to choose questions whose answers give a lot of information about what our tree should predict. If there's a single yes/no question for which "yes" answers always correspond to `True` outputs and "no" answers to `False` outputs (or vice versa), this would be an awesome question to pick. Conversely, a yes/no question for which neither answer gives you much new information about what the prediction should be is probably not a good choice.

We capture this notion of "how much information" with *entropy*. You have probably heard this used to mean disorder. We use it to represent the uncertainty associated with data.

Imagine that we have a set S of data, each member of which is labeled as belonging to one of a finite number of classes $C_1, \ldots, C_n$. If all the data points belong to a single class, then there is no real uncertainty, which means we'd like there to be low entropy. If the data points are evenly spread across the classes, there is a lot of uncertainty and we'd like there to be high entropy.

In math terms, if p_i is the proportion of data labeled as class c_i, we define the entropy as:

$$H(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$

with the (standard) convention that $0 \log 0 = 0$.

Without worrying too much about the grisly details, each term $-p_i \log_2 p_i$ is non-negative and is close to zero precisely when p_i is either close to zero or close to one (Figure 17-2).

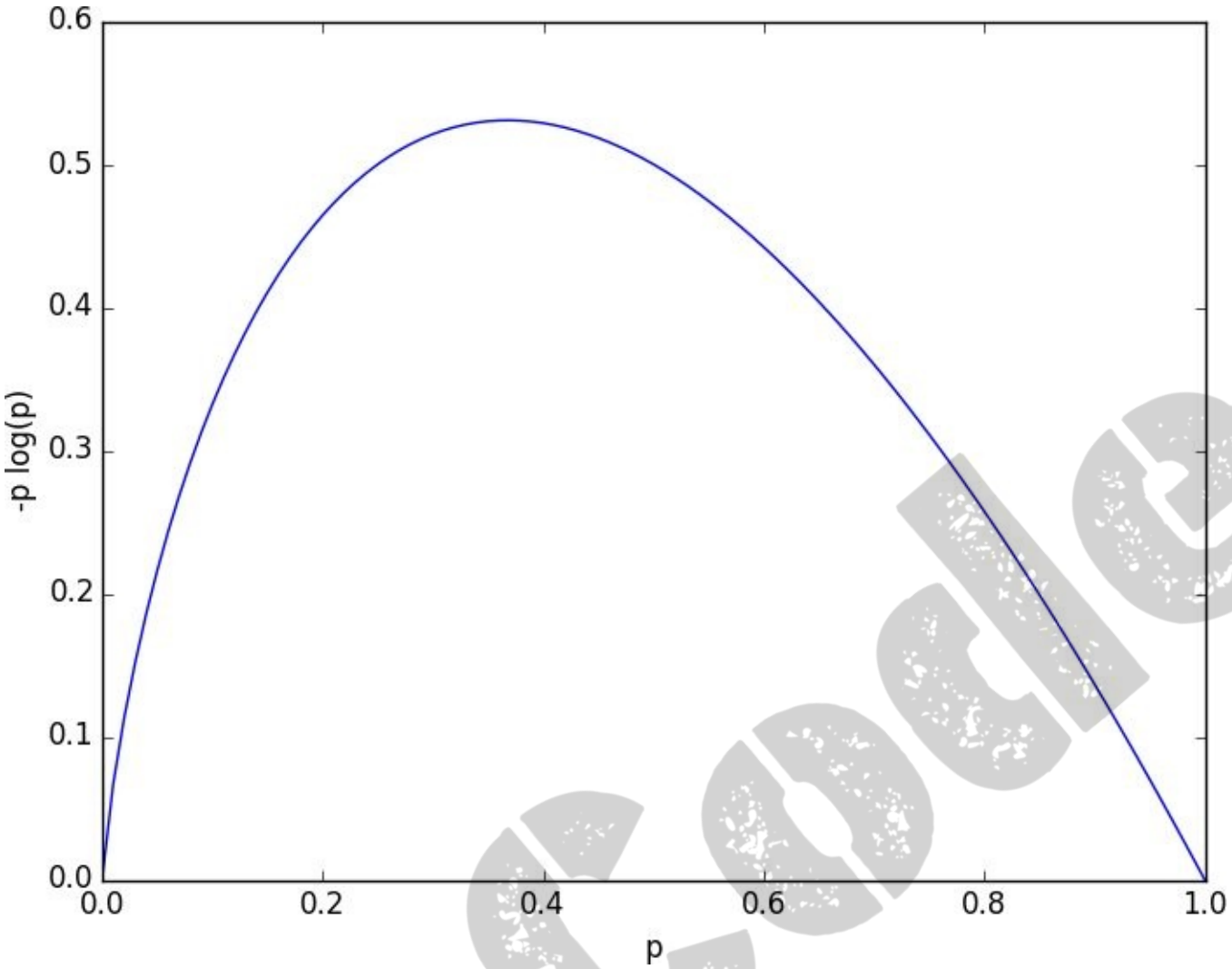

Figure 17-2. A graph of -p log p

This means the entropy will be small when every p_i is close to 0 or 1 (i.e., when most of the data is in a single class), and it will be larger when many of the p_i's are not close to 0 (i.e., when the data is spread across multiple classes). This is exactly the behavior we desire.

It is easy enough to roll all of this into a function:

```
def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p)                          # ignore zero probabilities
```

Our data will consist of pairs `(input, label)`, which means that we'll need to compute the class probabilities ourselves. Observe that we don't actually care which label is associated with each probability, only what the probabilities are:

```
def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
```

```
    return entropy(probabilities)
```

The Entropy of a Partition

What we've done so far is compute the entropy (think "uncertainty") of a single set of labeled data. Now, each stage of a decision tree involves asking a question whose answer partitions data into one or (hopefully) more subsets. For instance, our "does it have more than five legs?" question partitions animals into those who have more than five legs (e.g., spiders) and those that don't (e.g., echidnas).

Correspondingly, we'd like some notion of the entropy that results from partitioning a set of data in a certain way. We want a partition to have low entropy if it splits the data into subsets that themselves have low entropy (i.e., are highly certain), and high entropy if it contains subsets that (are large and) have high entropy (i.e., are highly uncertain).

For example, my "Australian five-cent coin" question was pretty dumb (albeit pretty lucky!), as it partitioned the remaining animals at that point into S_1 = {echidna} and S_2 = {everything else}, where S_2 is both large and high-entropy. (S_1 has no entropy but it represents a small fraction of the remaining "classes.")

Mathematically, if we partition our data S into subsets $S_1, \ldots, S_m$ containing proportions $q_1, \ldots, q_m$ of the data, then we compute the entropy of the partition as a weighted sum:

$$H = q_1 H(S_1) + \ldots + q_m H(S_m)$$

which we can implement as:

```
def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets
    subsets is a list of lists of labeled data"""

    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )
```

NOTE

One problem with this approach is that partitioning by an attribute with many different values will result in a very low entropy due to overfitting. For example, imagine you work for a bank and are trying to build a decision tree to predict which of your customers are likely to default on their mortgages, using some historical data as your training set. Imagine further that the data set contains each customer's Social Security number. Partitioning on SSN will produce one-person subsets, each of which necessarily has zero entropy. But a model that relies on SSN is *certain* not to generalize beyond the training set. For this reason, you should probably try to avoid (or bucket, if appropriate) attributes with large numbers of possible values when creating decision trees.

Creating a Decision Tree

The VP provides you with the interviewee data, consisting of (per your specification) pairs `(input, label)`, where each `input` is a `dict` of candidate attributes, and each label is either `True` (the candidate interviewed well) or `False` (the candidate interviewed poorly). In particular, you are provided with each candidate's level, her preferred language, whether she is active on Twitter, and whether she has a PhD:

```
inputs = [
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'},    False),
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'},   False),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'},      True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'},   True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'},     False),
    ({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'},         True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'},  False),
    ({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'},  True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'}, True),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'},     True),
    ({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, False)
]
```

Our tree will consist of *decision nodes* (which ask a question and direct us differently depending on the answer) and *leaf nodes* (which give us a prediction). We will build it using the relatively simple *ID3* algorithm, which operates in the following manner. Let's say we're given some labeled data, and a list of attributes to consider branching on.

- If the data all have the same label, then create a leaf node that predicts that label and then stop.
- If the list of attributes is empty (i.e., there are no more possible questions to ask), then create a leaf node that predicts the most common label and then stop.
- Otherwise, try partitioning the data by each of the attributes
- Choose the partition with the lowest partition entropy
- Add a decision node based on the chosen attribute
- Recur on each partitioned subset using the remaining attributes

This is what's known as a "greedy" algorithm because, at each step, it chooses the most immediately best option. Given a data set, there may be a better tree with a worse-looking first move. If so, this algorithm won't find it. Nonetheless, it is relatively easy to understand and implement, which makes it a good place to begin exploring decision trees.

Let's manually go through these steps on the interviewee data set. The data set has both `True` and `False` labels, and we have four attributes we can split on. So our first step will be to find the partition with the least entropy. We'll start by writing a function that does

the partitioning:

```
def partition_by(inputs, attribute):
    """each input is a pair (attribute_dict, label).
    returns a dict : attribute_value -> inputs"""
    groups = defaultdict(list)
    for input in inputs:
        key = input[0][attribute]   # get the value of the specified attribute
        groups[key].append(input)   # then add this input to the correct list
    return groups
```

and one that uses it to compute entropy:

```
def partition_entropy_by(inputs, attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

Then we just need to find the minimum-entropy partition for the whole data set:

```
for key in ['level','lang','tweets','phd']:
    print key, partition_entropy_by(inputs, key)

# level 0.693536138896
# lang 0.860131712855
# tweets 0.788450457308
# phd 0.892158928262
```

The lowest entropy comes from splitting on `level`, so we'll need to make a subtree for each possible `level` value. Every `Mid` candidate is labeled `True`, which means that the `Mid` subtree is simply a leaf node predicting `True`. For `Senior` candidates, we have a mix of `True`s and `False`s, so we need to split again:

```
senior_inputs = [(input, label)
                 for input, label in inputs if input["level"] == "Senior"]

for key in ['lang', 'tweets', 'phd']:
    print key, partition_entropy_by(senior_inputs, key)

# lang 0.4
# tweets 0.0
# phd 0.950977500433
```

This shows us that our next split should be on `tweets`, which results in a zero-entropy partition. For these Senior-level candidates, "yes" tweets always result in `True` while "no" tweets always result in `False`.

Finally, if we do the same thing for the `Junior` candidates, we end up splitting on `phd`, after which we find that no PhD always results in `True` and PhD always results in `False`.

Figure 17-3 shows the complete decision tree.

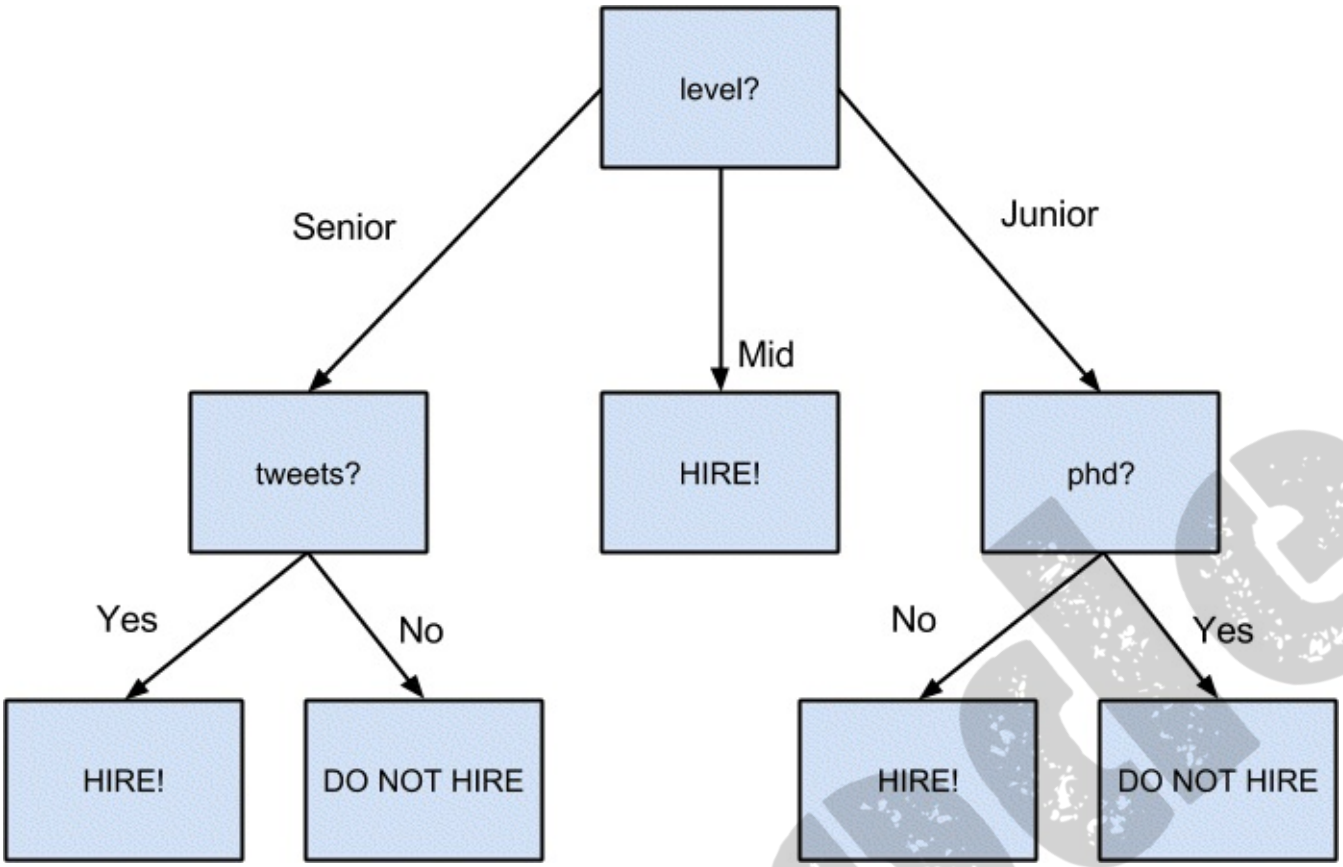

Figure 17-3. The decision tree for hiring

Putting It All Together

Now that we've seen how the algorithm works, we would like to implement it more generally. This means we need to decide how we want to represent trees. We'll use pretty much the most lightweight representation possible. We define a *tree* to be one of the following:

- `True`
- `False`
- a tuple `(attribute, subtree_dict)`

Here `True` represents a leaf node that returns `True` for any input, `False` represents a leaf node that returns `False` for any input, and a tuple represents a decision node that, for any input, finds its `attribute` value, and classifies the input using the corresponding subtree.

With this representation, our hiring tree would look like:

```
('level',
 {'Junior': ('phd', {'no': True, 'yes': False}),
  'Mid': True,
  'Senior': ('tweets', {'no': False, 'yes': True})})
```

There's still the question of what to do if we encounter an unexpected (or missing) attribute value. What should our hiring tree do if it encounters a candidate whose `level` is "Intern"? We'll handle this case by adding a `None` key that just predicts the most common label. (Although this would be a bad idea if `None` is actually a value that appears in the data.)

Given such a representation, we can classify an input with:

```
def classify(tree, input):
    """classify the input using the given decision tree"""

    # if this is a leaf node, return its value
    if tree in [True, False]:
        return tree

    # otherwise this tree consists of an attribute to split on
    # and a dictionary whose keys are values of that attribute
    # and whose values of are subtrees to consider next
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)    # None if input is missing attribute

    if subtree_key not in subtree_dict:   # if no subtree for key,
        subtree_key = None                # we'll use the None subtree

    subtree = subtree_dict[subtree_key]   # choose the appropriate subtree
    return classify(subtree, input)       # and use it to classify the input
```

All that's left is to build the tree representation from our training data:

```
def build_tree_id3(inputs, split_candidates=None):
```

```
    # if this is our first pass,
    # all keys of the first input are split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # count Trues and Falses in the inputs
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0: return False     # no Trues? return a "False" leaf
    if num_falses == 0: return True     # no Falses? return a "True" leaf

    if not split_candidates:            # if no split candidates left
        return num_trues >= num_falses  # return the majority leaf

    # otherwise, split on the best attribute
    best_attribute = min(split_candidates,
                         key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # recursively build the subtrees
    subtrees = { attribute_value : build_tree_id3(subset, new_candidates)
                 for attribute_value, subset in partitions.iteritems() }

    subtrees[None] = num_trues > num_falses      # default case

    return (best_attribute, subtrees)
```

In the tree we built, every leaf consisted entirely of `True` inputs or entirely of `False` inputs. This means that the tree predicts perfectly on the training data set. But we can also apply it to new data that wasn't in the training set:

```
tree = build_tree_id3(inputs)

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "no"} )        # True

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "yes"} )       # False
```

And also to data with missing or unexpected values:

```
classify(tree, { "level" : "Intern" } ) # True
classify(tree, { "level" : "Senior" } ) # False
```

> **NOTE**
>
> Since our goal was mainly to demonstrate *how* to build a tree, we built the tree using the entire data set. As always, if we were really trying to create a good model for something, we would have (collected more data and) split the data into train/validation/test subsets.

Random Forests

Given how closely decision trees can fit themselves to their training data, it's not surprising that they have a tendency to overfit. One way of avoiding this is a technique called *random forests*, in which we build multiple decision trees and let them vote on how to classify inputs:

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

Our tree-building process was deterministic, so how do we get random trees?

One piece involves bootstrapping data (recall "Digression: The Bootstrap"). Rather than training each tree on all the `inputs` in the training set, we train each tree on the result of `bootstrap_sample(inputs)`. Since each tree is built using different data, each tree will be different from every other tree. (A side benefit is that it's totally fair to use the nonsampled data to test each tree, which means you can get away with using all of your data as the training set if you are clever in how you measure performance.) This technique is known as *bootstrap aggregating* or *bagging*.

A second source of randomness involves changing the way we chose the `best_attribute` to split on. Rather than looking at all the remaining attributes, we first choose a random subset of them and then split on whichever of those is best:

```
    # if there's already few enough split candidates, look at all of them
    if len(split_candidates) <= self.num_split_candidates:
        sampled_split_candidates = split_candidates
    # otherwise pick a random sample
    else:
        sampled_split_candidates = random.sample(split_candidates,
                                                 self.num_split_candidates)

    # now choose the best attribute only from those candidates
    best_attribute = min(sampled_split_candidates,
        key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
```

This is an example of a broader technique called *ensemble learning* in which we combine several *weak learners* (typically high-bias, low-variance models) in order to produce an overall strong model.

Random forests are one of the most popular and versatile models around.

Neural Networks

> I like nonsense; it wakes up the brain cells.
>
> Dr. Seuss

An *artificial neural network* (or neural network for short) is a predictive model motivated by the way the brain operates. Think of the brain as a collection of neurons wired together. Each neuron looks at the outputs of the other neurons that feed into it, does a calculation, and then either fires (if the calculation exceeds some threshhold) or doesn't (if it doesn't).

Accordingly, artificial neural networks consist of artificial neurons, which perform similar calculations over their inputs. Neural networks can solve a wide variety of problems like handwriting recognition and face detection, and they are used heavily in deep learning, one of the trendiest subfields of data science. However, most neural networks are "black boxes" — inspecting their details doesn't give you much understanding of *how* they're solving a problem. And large neural networks can be difficult to train. For most problems you'll encounter as a budding data scientist, they're probably not the right choice. Someday, when you're trying to build an artificial intelligence to bring about the Singularity, they very well might be.

Perceptrons

Pretty much the simplest neural network is the *perceptron*, which approximates a single neuron with *n* binary inputs. It computes a weighted sum of its inputs and "fires" if that weighted sum is zero or greater:

```
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights, bias, x):
    """returns 1 if the perceptron 'fires', 0 if not"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

The perceptron is simply distinguishing between the half spaces separated by the hyperplane of points `x` for which:

```
dot(weights,x) + bias == 0
```

With properly chosen weights, perceptrons can solve a number of simple problems (Figure 18-1). For example, we can create an *AND gate* (which returns 1 if both its inputs are 1 but returns 0 if one of its inputs is 0) with:

```
weights = [2, 2]
bias = -3
```

If both inputs are 1, the `calculation` equals 2 + 2 - 3 = 1, and the output is 1. If only one of the inputs is 1, the `calculation` equals 2 + 0 - 3 = -1, and the output is 0. And if both of the inputs are 0, the `calculation` equals -3, and the output is 0.

Similarly, we could build an *OR gate* with:

```
weights = [2, 2]
bias = -1
```

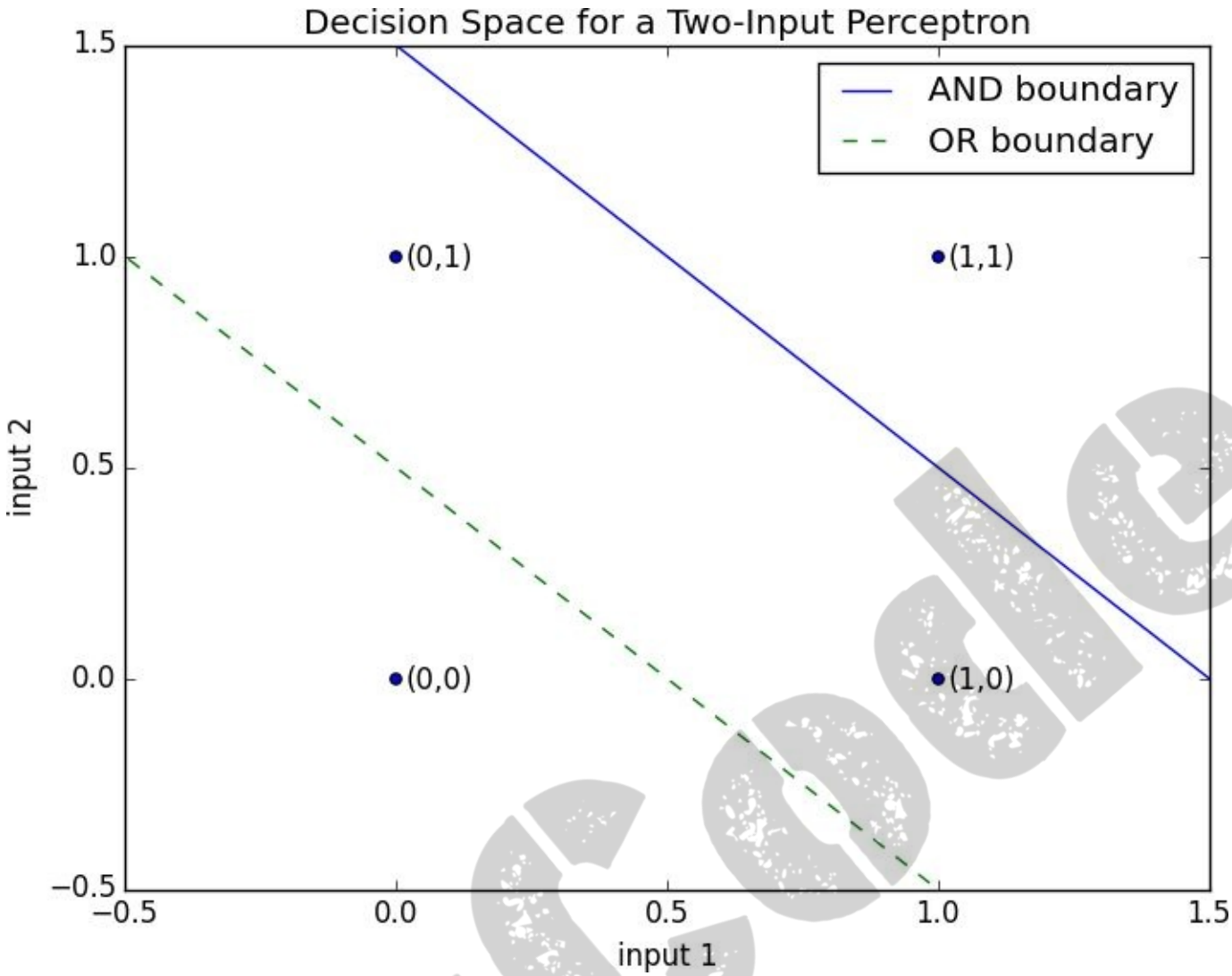

Figure 18-1. Decision space for a two-input perceptron

And we could build a *NOT gate* (which has one input and converts 1 to 0 and 0 to 1) with:

```
weights = [-2]
bias = 1
```

However, there are some problems that simply can't be solved by a single perceptron. For example, no matter how hard you try, you cannot use a perceptron to build an *XOR gate* that outputs 1 if exactly one of its inputs is 1 and 0 otherwise. This is where we start needing more-complicated neural networks.

Of course, you don't need to approximate a neuron in order to build a logic gate:

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Like real neurons, artificial neurons start getting more interesting when you start connecting them together.

Feed-Forward Neural Networks

The topology of the brain is enormously complicated, so it's common to approximate it with an idealized *feed-forward* neural network that consists of discrete *layers* of neurons, each connected to the next. This typically entails an input layer (which receives inputs and feeds them forward unchanged), one or more "hidden layers" (each of which consists of neurons that take the outputs of the previous layer, performs some calculation, and passes the result to the next layer), and an output layer (which produces the final outputs).

Just like the perceptron, each (noninput) neuron has a weight corresponding to each of its inputs and a bias. To make our representation simpler, we'll add the bias to the end of our weights vector and give each neuron a *bias input* that always equals 1.

As with the perceptron, for each neuron we'll sum up the products of its inputs and its weights. But here, rather than outputting the `step_function` applied to that product, we'll output a smooth approximation of the step function. In particular, we'll use the `sigmoid` function (Figure 18-2):

```
def sigmoid(t):
    return 1 / (1 + math.exp(-t))
```

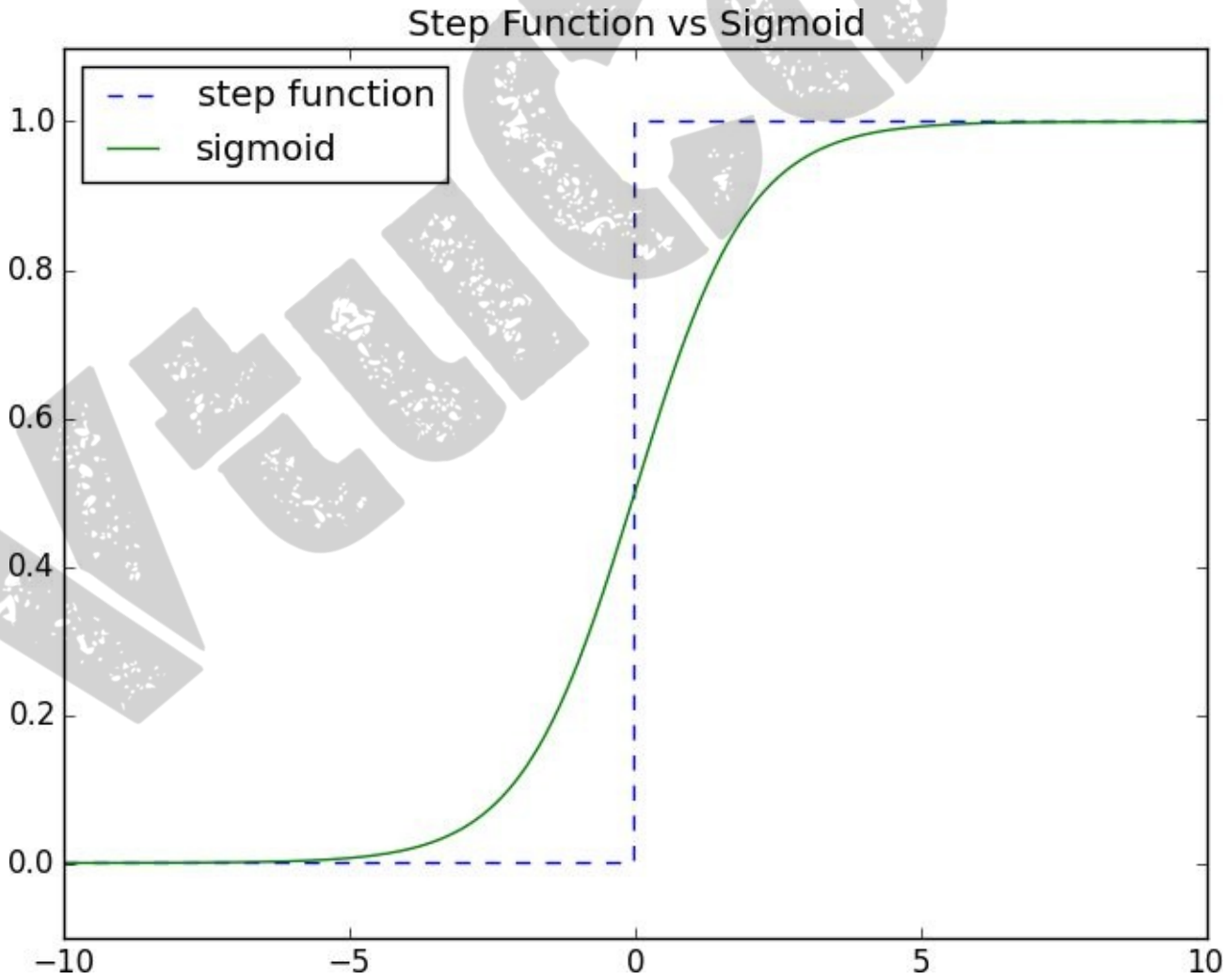

Figure 18-2. The sigmoid function

Why use `sigmoid` instead of the simpler `step_function`? In order to train a neural network, we'll need to use calculus, and in order to use calculus, we need *smooth* functions. The step function isn't even continuous, and sigmoid is a good smooth approximation of it.

> **NOTE**
>
> You may remember `sigmoid` from Chapter 16, where it was called `logistic`. Technically "sigmoid" refers to the *shape* of the function, "logistic" to this particular function although people often use the terms interchangeably.

We then calculate the output as:

```
def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))
```

Given this function, we can represent a neuron simply as a list of weights whose length is one more than the number of inputs to that neuron (because of the bias weight). Then we can represent a neural network as a list of (noninput) *layers*, where each layer is just a list of the neurons in that layer.

That is, we'll represent a neural network as a list (layers) of lists (neurons) of lists (weights).

Given such a representation, using the neural network is quite simple:

```
def feed_forward(neural_network, input_vector):
    """takes in a neural network
    (represented as a list of lists of lists of weights)
    and returns the output from forward-propagating the input"""

    outputs = []

    # process one layer at a time
    for layer in neural_network:
        input_with_bias = input_vector + [1]              # add a bias input
        output = [neuron_output(neuron, input_with_bias)  # compute the output
                  for neuron in layer]                    # for each neuron
        outputs.append(output)                            # and remember it

        # then the input to the next layer is the output of this one
        input_vector = output

    return outputs
```

Now it's easy to build the XOR gate that we couldn't build with a single perceptron. We just need to scale the weights up so that the `neuron_outputs` are either really close to 0 or really close to 1:

```
xor_network = [# hidden layer
               [[20, 20, -30],      # 'and' neuron
                [20, 20, -10]],     # 'or'  neuron
               # output layer
               [[-60, 60, -30]]]    # '2nd input but not 1st input' neuron

for x in [0, 1]:
    for y in [0, 1]:
        # feed_forward produces the outputs of every neuron
        # feed_forward[-1] is the outputs of the output-layer neurons
```

```
        print x, y, feed_forward(xor_network,[x, y])[-1]

# 0 0 [9.38314668300676e-14]
# 0 1 [0.9999999999999059]
# 1 0 [0.9999999999999059]
# 1 1 [9.383146683006828e-14]
```

By using a hidden layer, we are able to feed the output of an “and” neuron and the output of an “or” neuron into a “second input but not first input” neuron. The result is a network that performs “or, but not and,” which is precisely XOR (Figure 18-3).

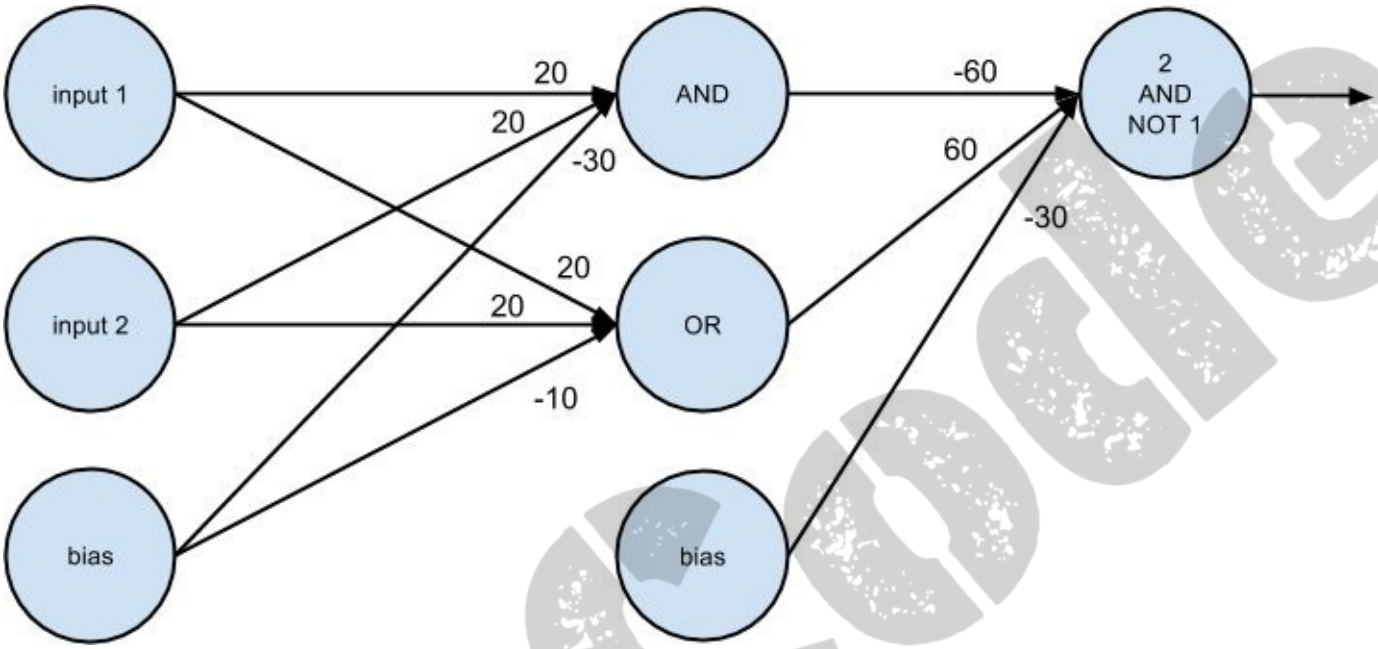

Figure 18-3. A neural network for XOR

Backpropagation

Usually we don't build neural networks by hand. This is in part because we use them to solve much bigger problems — an image recognition problem might involve hundreds or thousands of neurons. And it's in part because we usually won't be able to "reason out" what the neurons should be.

Instead (as usual) we use data to *train* neural networks. One popular approach is an algorithm called *backpropagation* that has similarities to the gradient descent algorithm we looked at earlier.

Imagine we have a training set that consists of input vectors and corresponding target output vectors. For example, in our previous `xor_network` example, the input vector `[1, 0]` corresponded to the target output `[1]`. And imagine that our network has some set of weights. We then adjust the weights using the following algorithm:

1. Run `feed_forward` on an input vector to produce the outputs of all the neurons in the network.

2. This results in an error for each output neuron — the difference between its output and its target.

3. Compute the gradient of this error as a function of the neuron's weights, and adjust its weights in the direction that most decreases the error.

4. "Propagate" these output errors backward to infer errors for the hidden layer.

5. Compute the gradients of these errors and adjust the hidden layer's weights in the same manner.

Typically we run this algorithm many times for our entire training set until the network converges:

```
def backpropagate(network, input_vector, targets):

    hidden_outputs, outputs = feed_forward(network, input_vector)

    # the output * (1 - output) is from the derivative of sigmoid
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, targets)]

    # adjust weights for output layer, one neuron at a time
    for i, output_neuron in enumerate(network[-1]):
        # focus on the ith output layer neuron
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            # adjust the jth weight based on both
            # this neuron's delta and its jth input
            output_neuron[j] -= output_deltas[i] * hidden_output

    # back-propagate errors to hidden layer
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                     dot(output_deltas, [n[i] for n in output_layer])
                     for i, hidden_output in enumerate(hidden_outputs)]

    # adjust weights for hidden layer, one neuron at a time
    for i, hidden_neuron in enumerate(network[0]):
        for j, input in enumerate(input_vector + [1]):
```

```
            hidden_neuron[j] -= hidden_deltas[i] * input
```

This is pretty much doing the same thing as if you explicitly wrote the squared error as a function of the weights and used the `minimize_stochastic` function we built in Chapter 8.

In this case, explicitly writing out the gradient function turns out to be kind of a pain. If you know calculus and the chain rule, the mathematical details are relatively straightforward, but keeping the notation straight ("the partial derivative of the error function with respect to the weight that neuron *i* assigns to the input coming from neuron *j*") is not much fun.

Example: Defeating a CAPTCHA

To make sure that people registering for your site are actually people, the VP of Product Management wants to implement a CAPTCHA as part of the registration process. In particular, he'd like to show users a picture of a digit and require them to input that digit to prove they're human.

He doesn't believe you that computers can easily solve this problem, so you decide to convince him by creating a program that can easily solve the problem.

We'll represent each digit as a 5 × 5 image:

```
@@@@@  ..@..  @@@@@  @@@@@  @...@  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@
@...@  ..@..  ....@  ....@  @...@  @....  @....  ....@  @...@  @...@
@...@  ..@..  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
@...@  ..@..  @....  ....@  ....@  ....@  @...@  ....@  @...@  ....@
@@@@@  ..@..  @@@@@  @@@@@  ....@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
```

Our neural network wants an input to be a vector of numbers. So we'll transform each image to a vector of length 25, whose elements are either 1 ("this pixel is in the image") or 0 ("this pixel is not in the image").

For instance, the zero digit would be represented as:

```
zero_digit = [1,1,1,1,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,1,1,1,1]
```

We'll want our output to indicate which digit the neural network thinks it is, so we'll need 10 outputs. The correct output for digit 4, for instance, will be:

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

Then, assuming our `inputs` are correctly ordered from 0 to 9, our targets will be:

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]
```

so that (for example) `targets[4]` is the correct output for digit 4.

At which point we're ready to build our neural network:

```
random.seed(0)      # to get repeatable results
input_size = 25     # each input is a vector of length 25
num_hidden = 5      # we'll have 5 neurons in the hidden layer
output_size = 10    # we need 10 outputs for each input

# each hidden neuron has one weight per input, plus a bias weight
hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

# each output neuron has one weight per hidden neuron, plus a bias weight
output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]
```

```
# the network starts out with random weights
network = [hidden_layer, output_layer]
```

And we can train it using the backpropagation algorithm:

```
# 10,000 iterations seems enough to converge
for __ in range(10000):
    for input_vector, target_vector in zip(inputs, targets):
        backpropagate(network, input_vector, target_vector)
```

It works well on the training set, obviously:

```
def predict(input):
    return feed_forward(network, input)[-1]

predict(inputs[7])
# [0.026, 0.0, 0.0, 0.018, 0.001, 0.0, 0.0, 0.967, 0.0, 0.0]
```

Which indicates that the digit 7 output neuron produces 0.97, while all the other output neurons produce very small numbers.

But we can also apply it to differently drawn digits, like my stylized 3:

```
predict([0,1,1,1,0,  # .@@@.
         0,0,0,1,1,  # ...@@
         0,0,1,1,0,  # ..@@.
         0,0,0,1,1,  # ...@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.92, 0.0, 0.0, 0.0, 0.01, 0.0, 0.12]
```

The network still thinks it looks like a 3, whereas my stylized 8 gets votes for being a 5, an 8, and a 9:

```
predict([0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.0, 0.0, 0.55, 0.0, 0.0, 0.93, 1.0]
```

Having a larger training set would probably help.

Although the network's operation is not exactly transparent, we can inspect the weights of the hidden layer to get a sense of what they're recognizing. In particular, we can plot the weights of each neuron as a 5 × 5 grid corresponding to the 5 × 5 inputs.

In real life you'd probably want to plot zero weights as white, with larger positive weights more and more (say) green and larger negative weights more and more (say) red. Unfortunately, that's hard to do in a black-and-white book.

Instead, we'll plot zero weights as white, with far-away-from-zero weights darker and darker. And we'll use crosshatching to indicate negative weights.

To do this we'll use `pyplot.imshow`, which we haven't seen before. With it we can plot

images pixel by pixel. Normally this isn't all that useful for data science, but here it's a good choice:

```
import matplotlib
weights = network[0][0]                 # first neuron in hidden layer
abs_weights = map(abs, weights)         # darkness only depends on absolute value


grid = [abs_weights[row:(row+5)]        # turn the weights into a 5x5 grid
        for row in range(0,25,5)]       # [weights[0:5], ..., weights[20:25]]

ax = plt.gca()                          # to use hatching, we'll need the axis

ax.imshow(grid,                         # here same as plt.imshow
          cmap=matplotlib.cm.binary,    # use white-black color scale
          interpolation='none')         # plot blocks as blocks

def patch(x, y, hatch, color):
    """return a matplotlib 'patch' object with the specified
    location, crosshatch pattern, and color"""
    return matplotlib.patches.Rectangle((x - 0.5, y - 0.5), 1, 1,
                                        hatch=hatch, fill=False, color=color)

# cross-hatch the negative weights
for i in range(5):                      # row
    for j in range(5):                  # column
        if weights[5*i + j] < 0:        # row i, column j = weights[5*i + j]
            # add black and white hatches, so visible whether dark or light
            ax.add_patch(patch(j, i, '/',  "white"))
            ax.add_patch(patch(j, i, '\\', "black"))

plt.show()
```

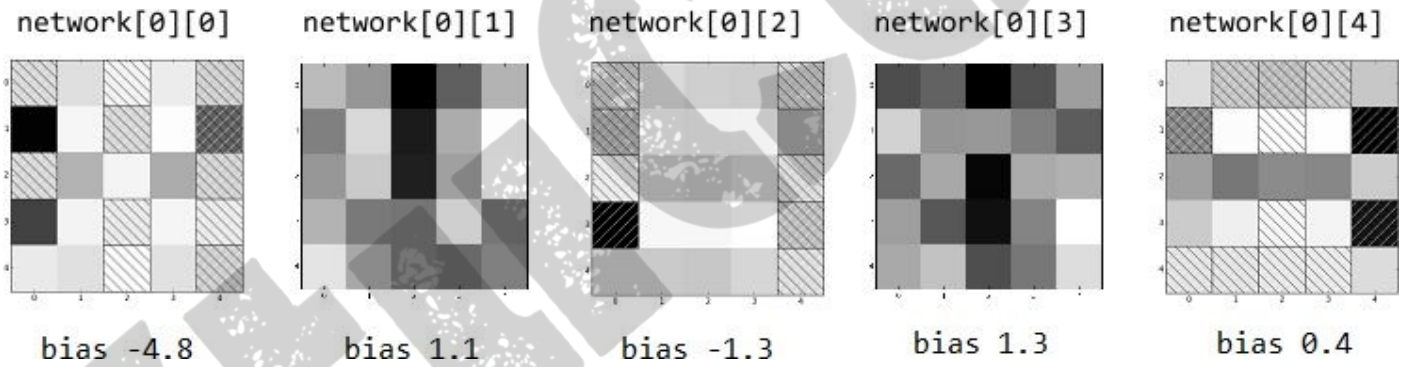

Figure 18-4. Weights for the hidden layer

In Figure 18-4 we see that the first hidden neuron has large positive weights in the left column and in the center of the middle row, while it has large negative weights in the right column. (And you can see that it has a pretty large negative bias, which means that it won't fire strongly unless it gets precisely the positive inputs it's "looking for.")

Indeed, on those inputs, it does what you'd expect:

```
left_column_only = [1, 0, 0, 0, 0] * 5
print feed_forward(network, left_column_only)[0][0]  # 1.0

center_middle_row = [0, 0, 0, 0, 0] * 2 + [0, 1, 1, 1, 0] + [0, 0, 0, 0, 0] * 2
print feed_forward(network, center_middle_row)[0][0]  # 0.95

right_column_only = [0, 0, 0, 0, 1] * 5
print feed_forward(network, right_column_only)[0][0]  # 0.0
```

Similarly, the middle hidden neuron seems to "like" horizontal lines but not side vertical

lines, and the last hidden neuron seems to “like” the center row but not the right column. (The other two neurons are harder to interpret.)

What happens when we run my stylized 3 through the network?

```
my_three =  [0,1,1,1,0,  # .@@@.
             0,0,0,1,1,  # ...@@
             0,0,1,1,0,  # ..@@.
             0,0,0,1,1,  # ...@@
             0,1,1,1,0]  # .@@@.

hidden, output = feed_forward(network, my_three)
```

The `hidden` outputs are:

```
0.121080  # from network[0][0], probably dinged by (1, 4)
0.999979  # from network[0][1], big contributions from (0, 2) and (2, 2)
0.999999  # from network[0][2], positive everywhere except (3, 4)
0.999992  # from network[0][3], again big contributions from (0, 2) and (2, 2)
0.000000  # from network[0][4], negative or zero everywhere except center row
```

which enter into the “three” output neuron with weights `network[-1][3]`:

```
-11.61  # weight for hidden[0]
 -2.17  # weight for hidden[1]
  9.31  # weight for hidden[2]
 -1.38  # weight for hidden[3]
-11.47  # weight for hidden[4]
- 1.92  # weight for bias input
```

So that the neuron computes:

```
sigmoid(.121 * -11.61 + 1 * -2.17 + 1 * 9.31 - 1.38 * 1 - 0 * 11.47 - 1.92)
```

which is 0.92, as we saw. In essence, the hidden layer is computing five different partitions of 25-dimensional space, mapping each 25-dimensional input down to five numbers. And then each output neuron looks only at the results of those five partitions.

As we saw, `my_three` falls slightly on the “low” side of partition 0 (i.e., only slightly activates hidden neuron 0), far on the “high” side of partitions 1, 2, and 3, (i.e., strongly activates those hidden neurons), and far on the low side of partition 4 (i.e., doesn’t active that neuron at all).

And then each of the 10 output neurons uses only those five activations to decide whether `my_three` is their digit or not.

Clustering

> Where we such clusters had
>
> As made us nobly wild, not mad
>
> Robert Herrick

Most of the algorithms in this book are what's known as supervised learning, in that they start with a set of *labeled* data and use that as the basis for making predictions about new, unlabeled data. Clustering, however, is an example of unsupervised learning, in which we work with completely unlabeled data (or in which our data has labels but we ignore them).

The Idea

Whenever you look at some source of data, it's likely that the data will somehow form *clusters*. A data set showing where millionaires live probably has clusters in places like Beverly Hills and Manhattan. A data set showing how many hours people work each week probably has a cluster around 40 (and if it's taken from a state with laws mandating special benefits for people who work at least 20 hours a week, it probably has another cluster right around 19). A data set of demographics of registered voters likely forms a variety of clusters (e.g., "soccer moms," "bored retirees," "unemployed millennials") that pollsters and political consultants likely consider relevant.

Unlike some of the problems we've looked at, there is generally no "correct" clustering. An alternative clustering scheme might group some of the "unemployed millenials" with "grad students," others with "parents' basement dwellers." Neither scheme is necessarily more correct — instead, each is likely more optimal with respect to its own "how good are the clusters?" metric.

Furthermore, the clusters won't label themselves. You'll have to do that by looking at the data underlying each one.

The Model

For us, each `input` will be a vector in *d*-dimensional space (which, as usual, we will represent as a list of numbers). Our goal will be to identify clusters of similar inputs and (sometimes) to find a representative value for each cluster.

For example, each input could be (a numeric vector that somehow represents) the title of a blog post, in which case the goal might be to find clusters of similar posts, perhaps in order to understand what our users are blogging about. Or imagine that we have a picture containing thousands of `(red, green, blue)` colors and that we need to screen-print a 10-color version of it. Clustering can help us choose 10 colors that will minimize the total "color error."

One of the simplest clustering methods is *k-means*, in which the number of clusters *k* is chosen in advance, after which the goal is to partition the inputs into sets $S_1, \ldots, S_k$ in a way that minimizes the total sum of squared distances from each point to the mean of its assigned cluster.

There are a lot of ways to assign *n* points to *k* clusters, which means that finding an optimal clustering is a very hard problem. We'll settle for an iterative algorithm that usually finds a good clustering:

1. Start with a set of *k-means*, which are points in *d*-dimensional space.
2. Assign each point to the mean to which it is closest.
3. If no point's assignment has changed, stop and keep the clusters.
4. If some point's assignment has changed, recompute the means and return to step 2.

Using the `vector_mean` function from Chapter 4, it's pretty simple to create a class that does this:

```
class KMeans:
    """performs k-means clustering"""

    def __init__(self, k):
        self.k = k          # number of clusters
        self.means = None   # means of clusters

    def classify(self, input):
        """return the index of the cluster closest to the input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs):
        # choose k random points as the initial means
        self.means = random.sample(inputs, self.k)
        assignments = None

        while True:
            # Find new assignments
            new_assignments = map(self.classify, inputs)

            # If no assignments have changed, we're done.
            if assignments == new_assignments:
```

```
            return

        # Otherwise keep the new assignments,
        assignments = new_assignments

        # And compute new means based on the new assignments
        for i in range(self.k):
            # find all the points assigned to cluster i
            i_points = [p for p, a in zip(inputs, assignments) if a == i]

            # make sure i_points is not empty so don't divide by 0
            if i_points:
                self.means[i] = vector_mean(i_points)
```

Let’s take a look at how this works.

Example: Meetups

To celebrate DataSciencester's growth, your VP of User Rewards wants to organize several in-person meetups for your hometown users, complete with beer, pizza, and DataSciencester t-shirts. You know the locations of all your local users (Figure 19-1), and she'd like you to choose meetup locations that make it convenient for everyone to attend.

Depending on how you look at it, you probably see two or three clusters. (It's easy to do visually because the data is only two-dimensional. With more dimensions, it would be a lot harder to eyeball.)

Imagine first that she has enough budget for three meetups. You go to your computer and try this:

```
random.seed(0)          # so you get the same results as me
clusterer = KMeans(3)
clusterer.train(inputs)
print clusterer.means
```

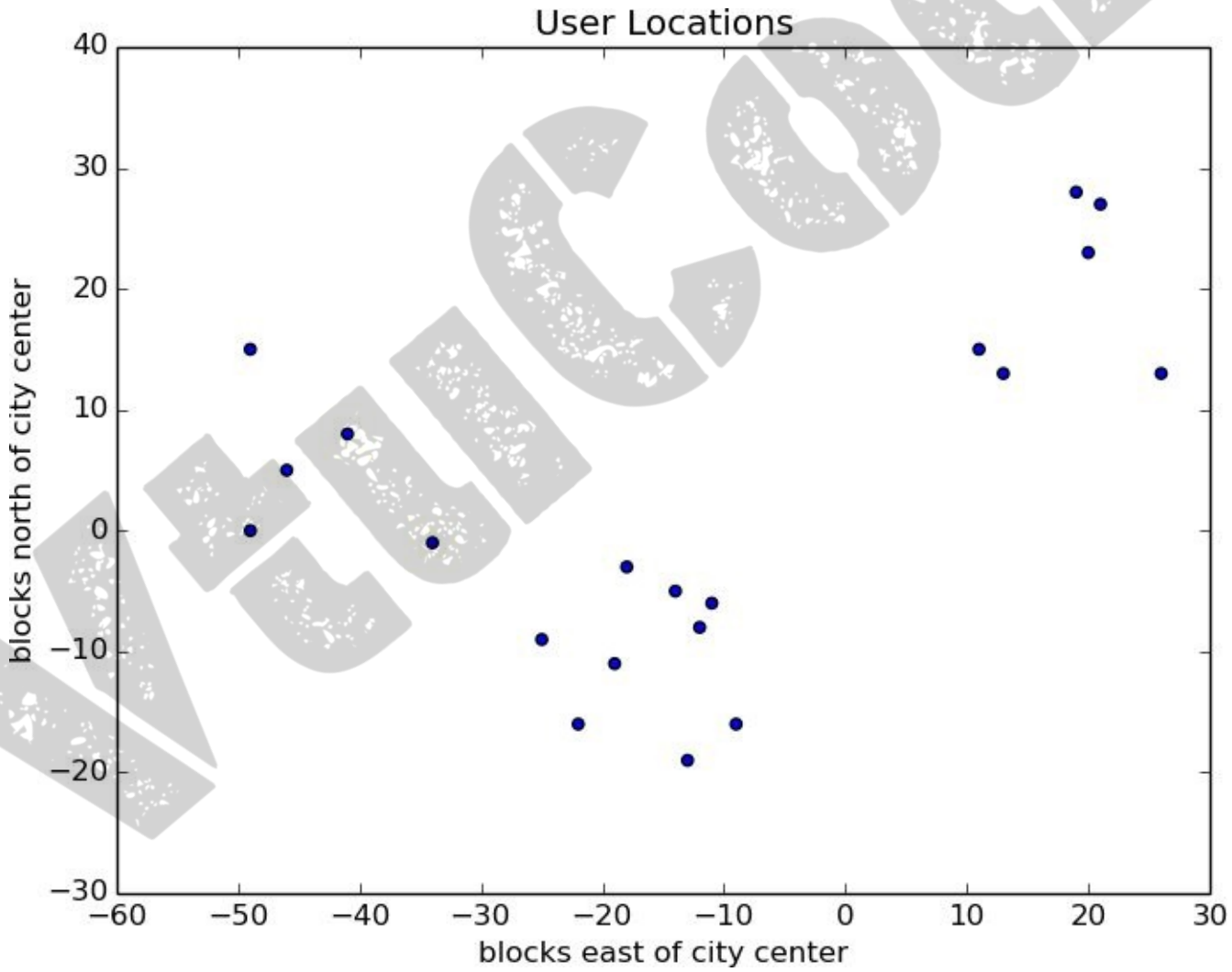

Figure 19-1. The locations of your hometown users

You find three clusters centered at [-44,5], [-16,-10], and [18, 20], and you look for meetup venues near those locations (Figure 19-2).

You show it to the VP, who informs you that now she only has enough budget for *two*

meetups.

“No problem,” you say:

```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

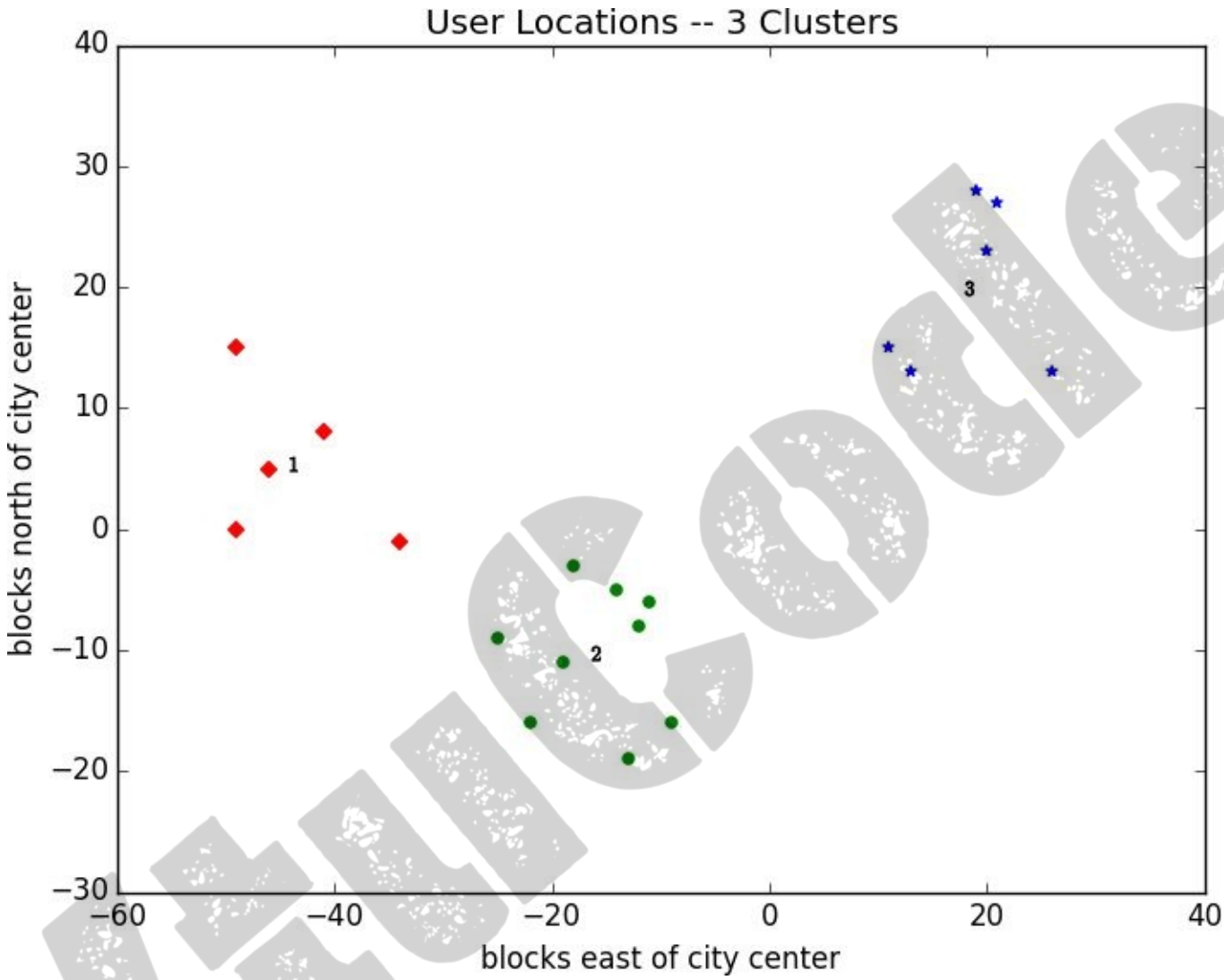

Figure 19-2. User locations grouped into three clusters

As shown in Figure 19-3, one meetup should still be near [18, 20], but now the other should be near [-26, -5].

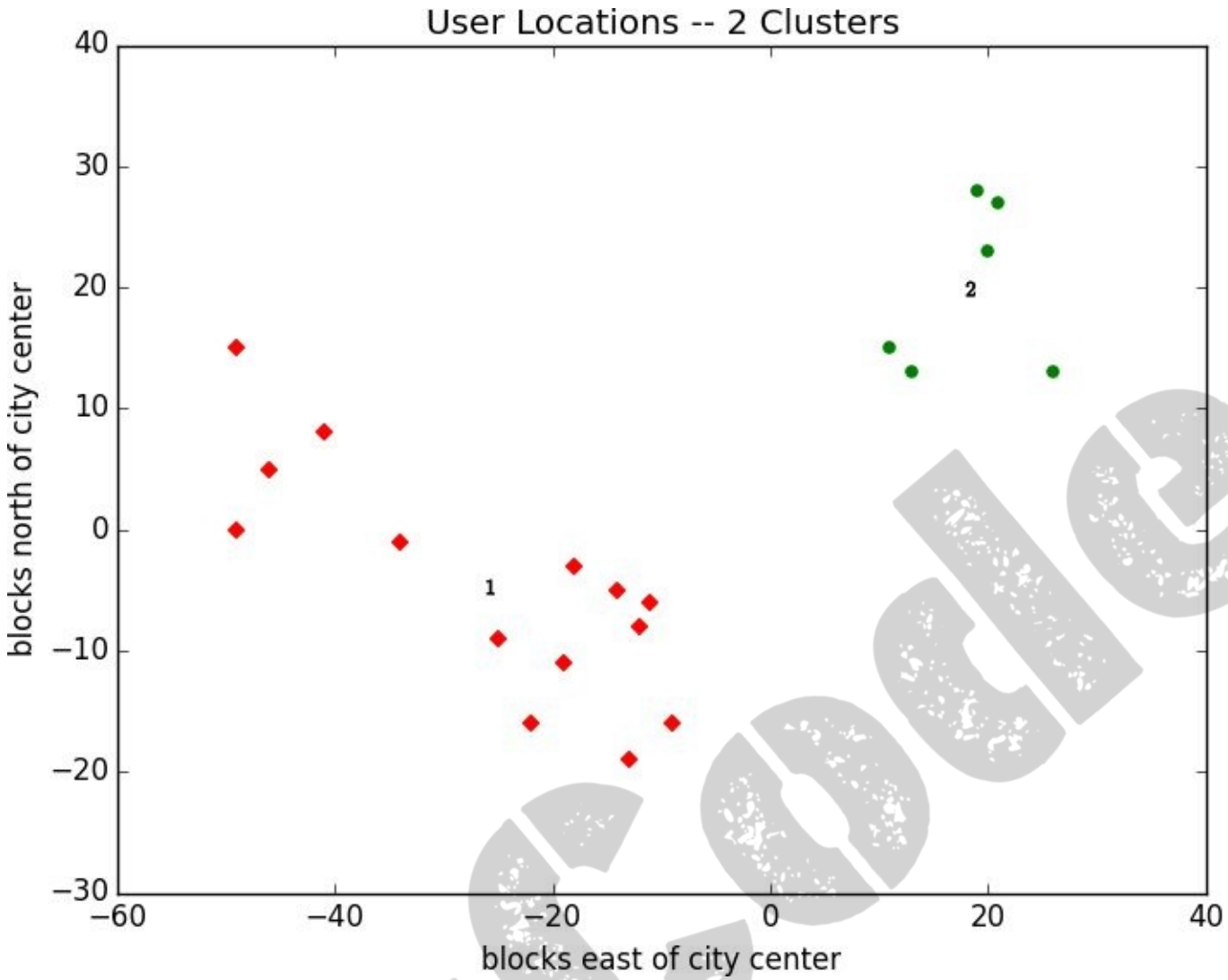

Figure 19-3. User locations grouped into two clusters

Choosing k

In the previous example, the choice of *k* was driven by factors outside of our control. In general, this won't be the case. There is a wide variety of ways to choose a *k*. One that's reasonably easy to understand involves plotting the sum of squared errors (between each point and the mean of its cluster) as a function of *k* and looking at where the graph "bends":

```
def squared_clustering_errors(inputs, k):
    """finds the total squared error from k-means clustering the inputs"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))

# now plot from 1 up to len(inputs) clusters

ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("total squared error")
plt.title("Total Error vs. # of Clusters")
plt.show()
```

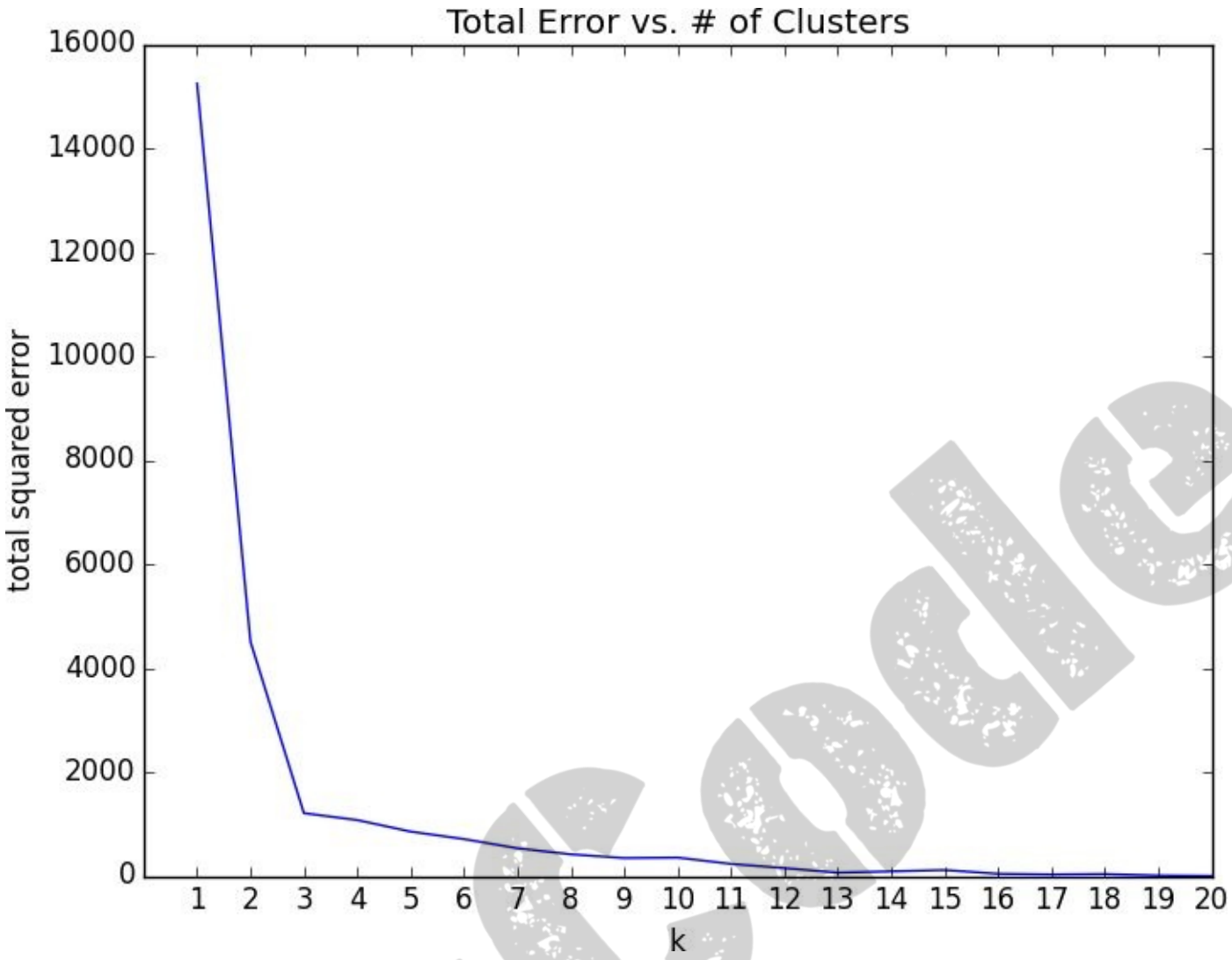

Figure 19-4. Choosing a k

Looking at Figure 19-4, this method agrees with our original eyeballing that 3 is the "right" number of clusters.

Example: Clustering Colors

The VP of Swag has designed attractive DataSciencester stickers that he'd like you to hand out at meetups. Unfortunately, your sticker printer can print at most five colors per sticker. And since the VP of Art is on sabbatical, the VP of Swag asks if there's some way you can take his design and modify it so that it only contains five colors.

Computer images can be represented as two-dimensional array of pixels, where each pixel is itself a three-dimensional vector `(red, green, blue)` indicating its color.

Creating a five-color version of the image then entails:

1. Choosing five colors

2. Assigning one of those colors to each pixel

It turns out this is a great task for k-means clustering, which can partition the pixels into five clusters in red-green-blue space. If we then recolor the pixels in each cluster to the mean color, we're done.

To start with, we'll need a way to load an image into Python. It turns out we can do this with `matplotlib`:

```
path_to_png_file = r"C:\images\image.png"   # wherever your image is
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

Behind the scenes `img` is a NumPy array, but for our purposes, we can treat it as a list of lists.

`img[i][j]` is the pixel in the *i*th row and *j*th column, and each pixel is a list `[red, green, blue]` of numbers between 0 and 1 indicating the color of that pixel:

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

In particular, we can get a flattened list of all the pixels as:

```
pixels = [pixel for row in img for pixel in row]
```

and then feed them to our clusterer:

```
clusterer = KMeans(5)
clusterer.train(pixels)   # this might take a while
```

Once it finishes, we just construct a new image with the same format:

```
def recolor(pixel):
    cluster = clusterer.classify(pixel)        # index of the closest cluster
    return clusterer.means[cluster]            # mean of the closest cluster
```

```
new_img = [[recolor(pixel) for pixel in row]   # recolor this row of pixels
           for row in img]                     # for each row in the image
```

and display it, using `plt.imshow()`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

It is difficult to show color results in a black-and-white book, but Figure 19-5 shows grayscale versions of a full-color picture and the output of using this process to reduce it to five colors:

Figure 19-5. Original picture and its 5-means decoloring

Bottom-up Hierarchical Clustering

An alternative approach to clustering is to "grow" clusters from the bottom up. We can do this in the following way:

1. Make each input its own cluster of one.

2. As long as there are multiple clusters remaining, find the two closest clusters and merge them.

At the end, we'll have one giant cluster containing all the inputs. If we keep track of the merge order, we can recreate any number of clusters by unmerging. For example, if we want three clusters, we can just undo the last two merges.

We'll use a really simple representation of clusters. Our values will live in *leaf* clusters, which we will represent as 1-tuples:

```
leaf1 = ([10, 20],)   # to make a 1-tuple you need the trailing comma
leaf2 = ([30, -15],)  # otherwise Python treats the parentheses as parentheses
```

We'll use these to grow *merged* clusters, which we will represent as 2-tuples (merge order, children):

```
merged = (1, [leaf1, leaf2])
```

We'll talk about merge order in a bit, but first let's create a few helper functions:

```
def is_leaf(cluster):
    """a cluster is a leaf if it has length 1"""
    return len(cluster) == 1

def get_children(cluster):
    """returns the two children of this cluster if it's a merged cluster;
    raises an exception if this is a leaf cluster"""
    if is_leaf(cluster):
        raise TypeError("a leaf cluster has no children")
    else:
        return cluster[1]

def get_values(cluster):
    """returns the value in this cluster (if it's a leaf cluster)
    or all the values in the leaf clusters below it (if it's not)"""
    if is_leaf(cluster):
        return cluster      # is already a 1-tuple containing value
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

In order to merge the closest clusters, we need some notion of the distance between clusters. We'll use the *minimum* distance between elements of the two clusters, which merges the two clusters that are closest to touching (but will sometimes produce large chain-like clusters that aren't very tight). If we wanted tight spherical clusters, we might use the *maximum* distance instead, as it merges the two clusters that fit in the smallest ball. Both are common choices, as is the *average* distance:

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """compute all the pairwise distances between cluster1 and cluster2
    and apply _distance_agg_ to the resulting list"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in get_values(cluster2)])
```

We'll use the merge order slot to track the order in which we did the merging. Smaller numbers will represent *later* merges. This means when we want to unmerge clusters, we do so from lowest merge order to highest. Since leaf clusters were never merged (which means we never want to unmerge them), we'll assign them infinity:

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0]  # merge_order is first element of 2-tuple
```

Now we're ready to create the clustering algorithm:

```
def bottom_up_cluster(inputs, distance_agg=min):
    # start with every input a leaf cluster / 1-tuple
    clusters = [(input,) for input in inputs]

    # as long as we have more than one cluster left…
    while len(clusters) > 1:
        # find the two closest clusters
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                     key=lambda (x, y): cluster_distance(x, y, distance_agg))

        # remove them from the list of clusters
        clusters = [c for c in clusters if c != c1 and c != c2]

        # merge them, using merge_order = # of clusters left
        merged_cluster = (len(clusters), [c1, c2])

        # and add their merge
        clusters.append(merged_cluster)

    # when there's only one cluster left, return it
    return clusters[0]
```

Its use is very simple:

```
base_cluster = bottom_up_cluster(inputs)
```

This produces a cluster whose ugly representation is:

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                     ([20, 23],)]),
               ([26, 13],)]),
          (16, [([11, 15],),
                ([13, 13],)])]),
     (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                               ([-46, 5],)]),
                          ([-41, 8],)]),
                    ([-49, 15],)]),
               ([-34, -1],)]),
          (6, [(7, [(8, [(10, [([-22, -16],),
                               ([-19, -11],)]),
                          ([-25, -9],)]),
```

```
                (13, [(15, [(17, [([-11, -6],),
                                  ([-12, -8],)]),
                            ([-14, -5],)]),
                      ([-18, -3],)])]),
          (12, [([-13, -19],),
                ([-9, -16],)])])])
```

For every merged cluster, I lined up its children vertically. If we say “cluster 0” for the cluster with merge order 0, you can interpret this as:

- Cluster 0 is the merger of cluster 1 and cluster 2.
- Cluster 1 is the merger of cluster 3 and cluster 16.
- Cluster 16 is the merger of the leaf `[11, 15]` and the leaf `[13, 13]`.
- And so on…

Since we had 20 inputs, it took 19 merges to get to this one cluster. The first merge created cluster 18 by combining the leaves `[19, 28]` and `[21, 27]`. And the last merge created cluster 0.

Generally, though, we don’t want to be squinting at nasty text representations like this. (Although it could be an interesting exercise to create a user-friendlier visualization of the cluster hierarchy.) Instead let’s write a function that generates any number of clusters by performing the appropriate number of unmerges:

```
def generate_clusters(base_cluster, num_clusters):
    # start with a list with just the base cluster
    clusters = [base_cluster]

    # as long as we don't have enough clusters yet…
    while len(clusters) < num_clusters:
        # choose the last-merged of our clusters
        next_cluster = min(clusters, key=get_merge_order)
        # remove it from the list
        clusters = [c for c in clusters if c != next_cluster]
        # and add its children to the list (i.e., unmerge it)
        clusters.extend(get_children(next_cluster))

    # once we have enough clusters…
    return clusters
```

So, for example, if we want to generate three clusters, we can just do:

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

which we can easily plot:

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # magic unzipping trick
    plt.scatter(xs, ys, color=color, marker=marker)

    # put a number at the mean of the cluster
    x, y = vector_mean(cluster)
```

```
        plt.plot(x, y, marker='$' + str(i) + '$', color='black')

    plt.title("User Locations—3 Bottom-Up Clusters, Min")
    plt.xlabel("blocks east of city center")
    plt.ylabel("blocks north of city center")
    plt.show()
```

This gives very different results than k-means did, as shown in Figure 19-6.

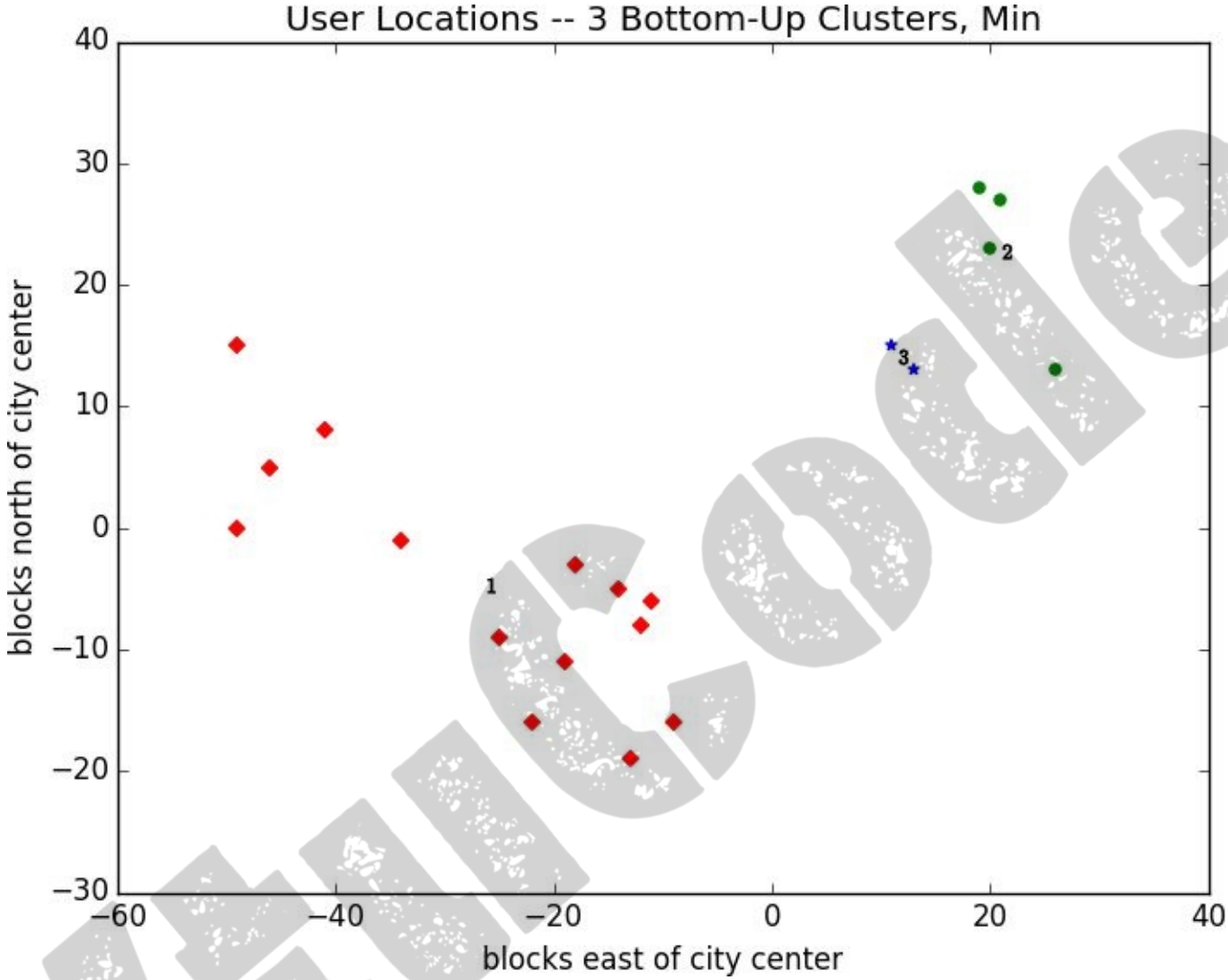

Figure 19-6. Three bottom-up clusters using min distance

As we mentioned above, this is because using `min` in `cluster_distance` tends to give chain-like clusters. If we instead use `max` (which gives tight clusters) it looks the same as the 3-means result (Figure 19-7).

NOTE

The `bottom_up_clustering` implementation above is relatively simple, but it's also shockingly inefficient. In particular, it recomputes the distance between each pair of inputs at every step. A more efficient implementation might precompute the distances between each pair of inputs and then perform a lookup inside cluster_distance. A *really* efficient implementation would likely also remember the `cluster_distances` from the previous step.

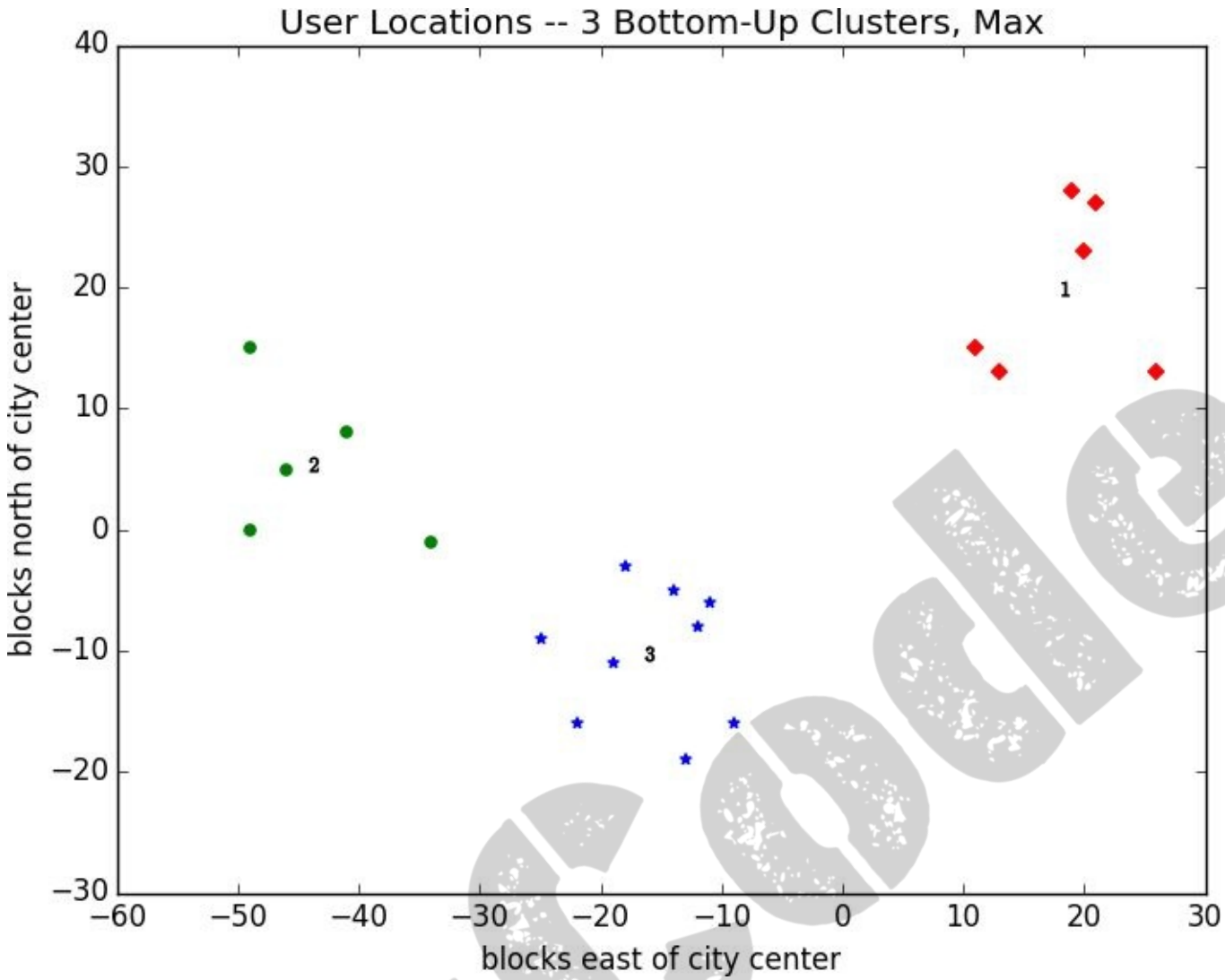

Figure 19-7. Three bottom-up clusters using max distance