PREV
Chapter 4. A gentle introduction to classification

NEXT
Chapter 6. Hidden Markov models

Chapter 5. Automatically clustering data

This chapter covers

- Basic clustering with k-means
- Representing audio
- Audio segmentation
- Clustering with a self-organizing map

Suppose you have a collection of not-pirated, totally legal MP3s on your hard drive. All your songs are crowded in one massive folder. Perhaps automatically grouping similar songs into categories such as Country, Rap, and Rock would help organize them. This act of assigning an item to a group (such as an MP3 to a playlist) in an unsupervised fashion is called *clustering*.

The previous chapter on classification assumes you're given a training dataset of correctly labeled data. Unfortunately, you don't always have that luxury when you collect data in the real world. For example, suppose you want to divide a large amount of music into interesting playlists. How could you possibly group songs if you don't have direct access to their metadata?

Spotify, SoundCloud, Google Music, Pandora, and many other music-streaming services try to solve this problem in order to recommend similar songs to customers. Their approach includes a mixture of various machine-learning techniques, but clustering is often at the heart of the solution.

Clustering is the process of intelligently categorizing the items in your dataset. The overall idea is that two items in the same cluster are "closer" to each other than items that belong to separate clusters. That's the general definition, leaving the interpretation of *closeness* open. For example, perhaps cheetahs and leopards belong in the same cluster, whereas elephants belong to another, when closeness is measured by the similarity of two species in the hierarchy of biological classification (family, genus, and species).

You can imagine that many clustering algorithms are out there. This chapter focuses on two types: *k-means* and *self-organizing map*. These approaches are completely *unsupervised*, meaning they fit a model without ground-truth examples.

First, you'll learn how to load audio files into TensorFlow and represent them as feature vectors. Then, you'll implement various clustering techniques to solve real-world problems.

5.1. TRAVERSING FILES IN TENSORFLOW

Some common input types in machine-learning algorithms are audio and image files. This shouldn't come as a surprise, because sound recordings and photographs are raw, redundant, and often noisy representations of semantic concepts. Machine learning is a tool to help handle these complications.

These data files have various implementations: for example, an image can be encoded as a PNG or JPEG file, and an audio file can be an MP3 or a WAV. In this chapter, you'll investigate how to read audio files as input to your clustering algorithm so you automatically group music that sounds similar.

Exercise 5.1

What are the pros and cons of MP3 and WAV? How about PNG versus JPEG?

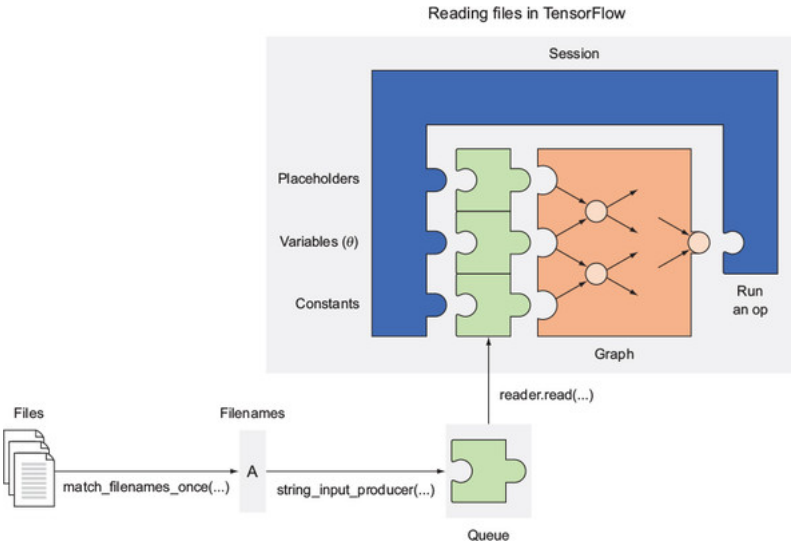
ANSWER

MP3 and JPEG significantly compress the data, so such files are easy to store or transmit. But because these are lossy, WAV and PNG are closer to the original content.

Reading files from disk isn't exactly a machine-learning-specific ability. You can use a variety of Python libraries to load files into memory, such as NumPy or SciPy. Some developers like to treat the data-preprocessing step separately from the machine-learning step. There's no absolute right or wrong way to manage the pipeline, but we'll try TensorFlow for both data preprocessing and learning.

TensorFlow provides an operator called `tf.train.match_filenames_once(...)` to list files in a directory. You can then pass this information along to the queue operator `tf.train.string_input_producer(...)`. That way, you can access filenames one at a time, without loading everything at once. Given a filename, you can decode the file to retrieve usable data. Figure 5.1 outlines the whole process of using the queue.

Figure 5.1. You can use a queue in TensorFlow to read files. The queue is built into the TensorFlow framework, and you can use the `reader.read(...)` function to access (and dequeue) it.



The following listing shows an implementation of reading files from disk in TensorFlow.

Listing 5.1. Traversing a directory for data

```
import tensorflow as tf

filenames = tf.train.match_filenames_once('./audio_dataset/*.wav')   1
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)           2
reader = tf.WholeFileReader()                                        3
filename, file_contents = reader.read(filename_queue)                4

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_files = sess.run(count_num_files)                            5

    coord = tf.train.Coordinator()                                   6
    threads = tf.train.start_queue_runners(coord=coord)              6

    for i in range(num_files):                                       7
        audio_file = sess.run(filename)                              7
        print(audio_file)                                            7
```

- ***1*** **Stores filenames that match a pattern**
- ***2*** **Sets up a pipeline for retrieving filenames randomly**
- ***3*** **Natively reads a file in TensorFlow**
- ***4*** **Runs the reader to extract file data**
- ***5*** **Counts the number of files**
- ***6*** **Initializes threads for the filename queue**
- ***7*** **Loops through the data one by one**

Tip

If you couldn't get listing 5.1 to work, you may want to try the advice posted on this book's official forum: http://mng.bz/Q9aD (http://mng.bz/Q9aD).

5.2. EXTRACTING FEATURES FROM AUDIO

Machine-learning algorithms are typically designed to use feature vectors as input; but sound files use a different format. You need a way to extract features from sound files to create feature vectors.

It helps to understand how these files are represented. If you've ever seen a vinyl record, you've probably noticed the representation of audio as grooves indented in the disk. Our ears interpret audio from a series of vibrations through air. By recording the vibration properties, an algorithm can store sound in a data format.

The real world is continuous, but computers store data in discrete values. The sound is digitalized into a discrete representation through an analog-to-digital converter (ADC). You can think about sound as fluctuation of a wave over time. But that data is too noisy and difficult to comprehend.

An equivalent way to represent a wave is by examining its frequencies at each time interval. This perspective is called the *frequency domain*. It's easy to convert between time domains and frequency domains by using a mathematical operation called a *discrete Fourier transform* (commonly implemented using an algorithm known as the *fast Fourier transform*). You'll use this technique to extract a feature vector out of a sound.

A handy Python library can help you view audio in this frequency domain. Download it from https://github.com/BinRoot/BregmanToolkit/archive/master.zip. Extract it, and then run the following command to set it up:

```
$ python setup.py install
```

Python 2 required

The BregmanToolkit is officially supported on Python 2. If you're using Jupyter Notebook, you can have access to both versions of Python by following the directions outlined in the official Jupyter docs: http://mng.bz/ebvw (http://mng.bz/ebvw).

In particular, you can include Python 2 with the following commands:

```
$ python2 -m pip install ipykernel
$ python2 -m -ipykernel install --user
```

A sound may produce 12 kinds of pitches. In music terminology, the 12 pitches are C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. Listing 5.2 shows how to retrieve the contribution of each pitch in a 0.1-second interval, resulting in a matrix with 12 rows.

The number of columns grows as the length of the audio file increases. Specifically, there will be 10 × t columns for a t-second audio. This matrix is also called a *chromagram* of the audio.

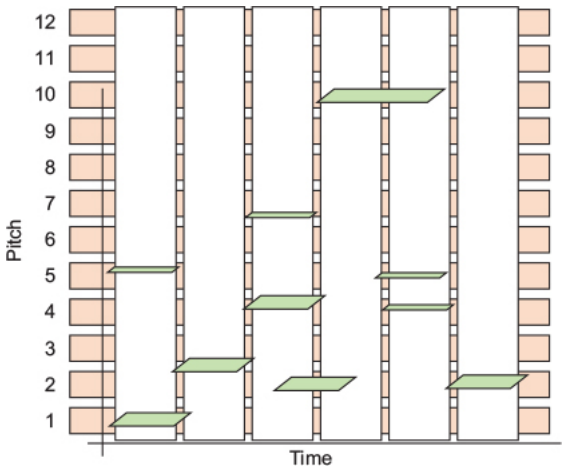
Listing 5.2. Representing audio in Python

```
from bregman.suite import *

def get_chromagram(audio_file):                                          1
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)         2
    return F.X                                                           3
```

- ***1* Passes in the filename**
- ***2* Uses these parameters to describe 12 pitches every 0.1 second**
- ***3* Represents the values of a 12-dimensional vector 10 times per second**

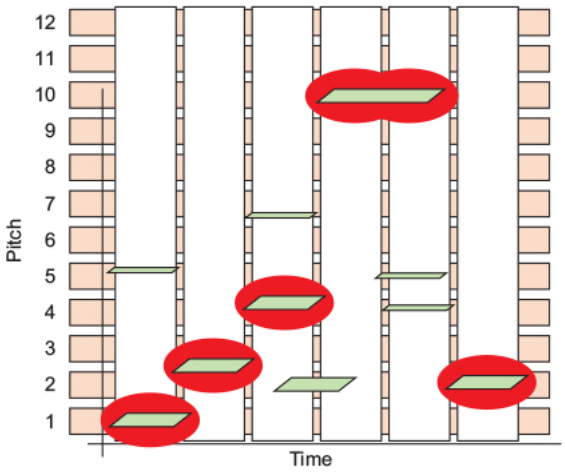
The chromagram output is a matrix, shown in figure 5.2. A sound clip can be read as a chromagram, and a chromagram is a recipe for generating a sound clip. Now you have a way to convert between audio and matrices. And as you've learned, most machine-learning algorithms accept feature vectors as a valid form of data. That said, the first machine-learning algorithm you'll look at is k-means clustering.

Figure 5.2. The chromagram matrix, where the x-axis represents time, and the y-axis represents pitch class. The green parallelograms indicate the presence of that pitch at that time.



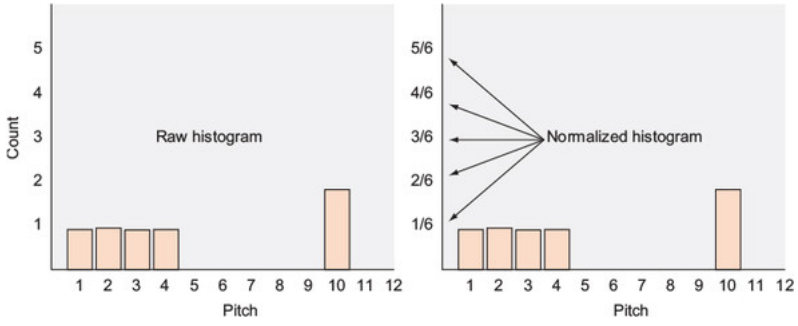
To run machine-learning algorithms on your chromagram, you first need to decide how you're going to represent a feature vector. One idea is to simplify the audio by looking only at the most significant pitch class per time interval, as shown in figure 5.3.

Figure 5.3. The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval.



Then you count the number of times each pitch shows up in the audio file. Figure 5.4 shows this data as a histogram, forming a 12-dimensional vector. If you normalize the vector so that all the counts add up to 1, you can easily compare audio of different lengths.

Figure 5.4. You count the frequency of loudest pitches heard at each interval to generate this histogram, which acts as your feature vector.



Exercise 5.2

What are some other ways to represent an audio clip as a feature vector?

Answer

You can visualize the audio clip as an image (such as a spectrogram), and use image-analysis techniques to extract image features.

Take a look at the following listing to generate the histogram from figure 5.4, which is your feature vector.

Listing 5.3. Obtaining a dataset for k-means

```
import tensorflow as tf
import numpy as np
from bregman.suite import *


filenames = tf.train.match_filenames_once('./audio_dataset/*.wav')
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)
reader = tf.WholeFileReader()
filename, file_contents = reader.read(filename_queue)
```

```
chroma = tf.placeholder(tf.float32)                                    1
max_freqs = tf.argmax(chroma, 0)                                       1

def get_next_chromagram(sess):
    audio_file = sess.run(filename)
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)
    return F.X

def extract_feature_vector(sess, chroma_data):                         2
    num_features, num_samples = np.shape(chroma_data)
    freq_vals = sess.run(max_freqs, feed_dict={chroma: chroma_data})
    hist, bins = np.histogram(freq_vals, bins=range(num_features + 1))
    return hist.astype(float) / num_samples

def get_dataset(sess):                                                 3
    num_files = sess.run(count_num_files)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    xs = []
    for _ in range(num_files):
        chroma_data = get_next_chromagram(sess)
        x = [extract_feature_vector(sess, chroma_data)]
        x = np.matrix(x)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

- ***1*** **Creates an op to identify the pitch with the biggest contribution**
- ***2*** **Converts a chromagram into a feature vector**
- ***3*** **Constructs a matrix where each row is a data item**

Note

All code listings are available from this book's website at www.manning.com/books/machine-learning-with-tensorflow (http://www.manning.com/books/machine-learning-with-tensorflow) and on GitHub at https://github.com/BinRoot/TensorFlow-Book/tree/master/ch05_clustering.

5.3. K-MEANS CLUSTERING

The *k-means algorithm* is one of the oldest yet most robust ways to cluster data. The *k* in k-means is a variable representing a natural number. So, you can imagine there's 3-means clustering, or 4-means clustering, or any other value for k. Thus, the first step of k-means clustering is to choose a value for *k*. Just to be more concrete, let's pick $k = 3$. With that in mind, the goal of 3-means clustering is to divide the dataset into three categories (also called *clusters*).

Choosing the number of clusters

Choosing the right number of clusters often depends on the task. For example, suppose you're planning an event for hundreds of people, both young and old. If you have the

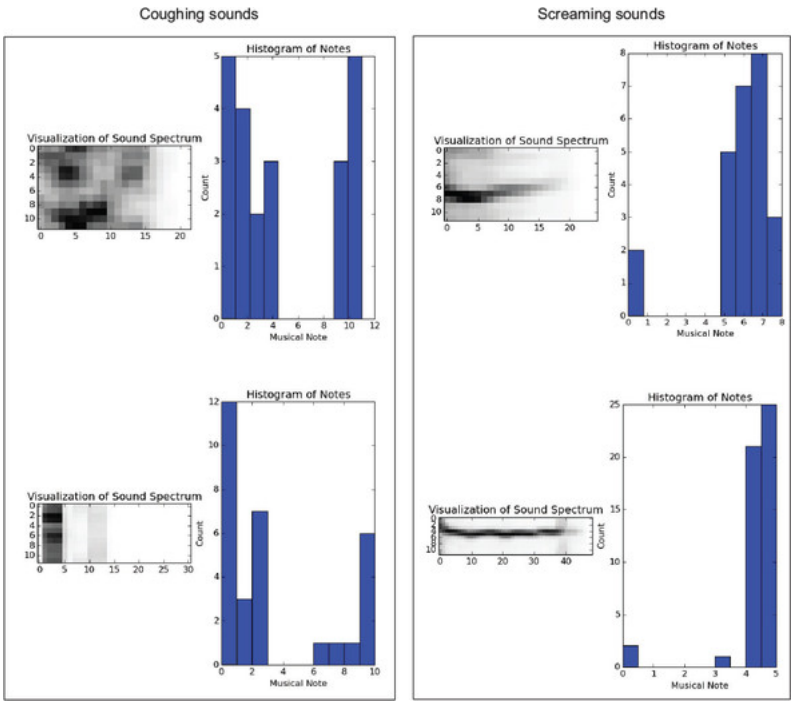
budget for only two entertainment options, you can use k-means clustering with $k = 2$ to separate the guests into two age groups. Other times, determining the value of k isn't as obvious. Automatically figuring out the value of k is a bit more complicated, so we won't touch on that much in this section. In simplified terms, a straightforward way of determining the best value of k is to iterate over a range of k-means simulations and apply a cost function to determine which value of k caused the best differentiation between clusters at the lowest value of k.

The k-means algorithm treats data points as points in space. If your dataset is a collection of guests at an event, you can represent each one by their age. Thus, your dataset is a collection of feature vectors. In this case, each feature vector is only one-dimensional, because you're considering only the age of the person.

For clustering music by the audio data, the data points are feature vectors from the audio files. If two points are close together, their audio features are similar. You want to discover which audio files belong in the same "neighborhood," because those clusters will probably be a good way to organize your music files.

The midpoint of all the points in a cluster is called its *centroid*. Depending on the audio features you choose to extract, a centroid could capture concepts such as loud sound, high-pitched sound, or saxophone-like sound. It's important to note that the k-means algorithm assigns nondescript labels, such as cluster 1, cluster 2, and cluster 3. Figure 5.5 shows examples of the sound data.

Figure 5.5. Four examples of audio files. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar histograms. Your clustering algorithms will be able to group these sounds together.



The k-means algorithm assigns a feature vector to one of the k clusters by choosing the cluster whose centroid is closest to it. The k-means algorithm starts by guessing the cluster location. It iteratively improves its guess over time. The algorithm either

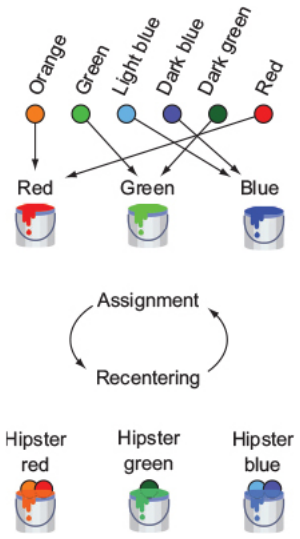
converges when it no longer improves the guesses, or stops after a maximum number of attempts.

The heart of the algorithm consists of two tasks, assignment and recentering:

1. In the assignment step, you assign each data item (feature vector) to a category of the closest centroid.
2. In the recentering step, you calculate the midpoints of the newly updated clusters.

These two steps repeat to provide increasingly better clustering results, and the algorithm stops either when it has repeated a desired number of times or when the assignments no longer change. Figure 5.6 illustrates the algorithm.

Figure 5.6. One iteration of the k-means algorithm. Let's say you're clustering colors into three buckets (an informal way to say *category*). You can start with an initial guess of red, green, and blue and begin the assignment step. Then you update the bucket colors by averaging the colors that belong to each bucket. You keep repeating until the buckets no longer substantially change color, arriving at the color representing the centroid of each cluster.



Listing 5.4 shows how to implement the k-means algorithm using the dataset generated by listing 5.3. For simplicity, you'll choose $k = 2$, so you can easily verify that your algorithm partitions the audio files into two dissimilar categories. You'll use the first k vectors as initial guesses for centroids.

Listing 5.4. Implementing k-means

```
k = 2                                                                    1
max_iterations = 100                                                     2

def initial_cluster_centroids(X, k):                                     3
    return X[0:k, :]

def assign_cluster(X, centroids):                                        4
    expanded_vectors = tf.expand_dims(X, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.subtract(expanded_vectors,
      expanded_centroids)), 2)
    mins = tf.argmin(distances, 0)
    return mins

def recompute_centroids(X, Y):                                           5
    sums = tf.unsorted_segment_sum(X, Y, k)
    counts = tf.unsorted_segment_sum(tf.ones_like(X), Y, k)
    return sums / counts
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    X = get_dataset(sess)
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:                    6
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
    print(centroids)
```

- ***1*** **Decides the number of clusters**
- ***2*** **Declares the maximum number of iterations to run k-means**
- ***3*** **Chooses the initial guesses of cluster centroids**
- ***4*** **Assigns each data item to its nearest cluster**
- ***5*** **Updates the cluster centroids to their midpoint**
- ***6*** **Iterates to find the best cluster locations**

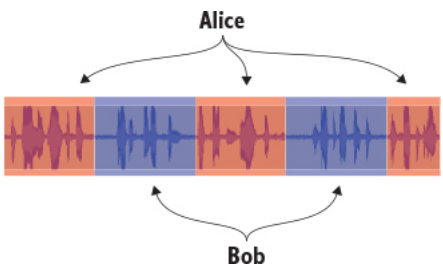
And that's it! If you know the number of clusters and the feature vector representation, you can use listing 5.4 to cluster anything! In the next section, you'll apply clustering to audio snippets within an audio file.

5.4. AUDIO SEGMENTATION

In the preceding section, you clustered various audio files to automatically group them. This section is about using clustering algorithms within just one audio file. Whereas the former is called *clustering*, the latter is referred to as *segmentation*. Segmentation is another word for clustering, but we often say *segment* instead of *cluster* when dividing a single image or audio file into separate components. It's similar to the way dividing a sentence into words is different from dividing a word into letters. Though they both share the general idea of breaking bigger pieces into smaller components, words are different from letters.

Let's say you have a long audio file, maybe of a podcast or talk show. Imagine writing a machine-learning algorithm to identify which of two people is speaking in an audio interview. The goal of segmenting an audio file is to associate which parts of the audio clip belong to the same category. In this case, you'd have a category for each person, and the utterances made by each person should converge to their appropriate categories, as shown in figure 5.7.

Figure 5.7. Audio segmentation is the process of automatically labeling segments.



Open a new source file, and follow along with listing 5.5, which will get you started by organizing the audio data for segmentation. It splits an audio file into multiple segments of size `segment_size`. A long audio file would contain hundreds, if not thousands, of segments.

Listing 5.5. Organizing data for segmentation

```
import tensorflow as tf
import numpy as np
from bregman.suite import *

k = 2                                                          1
segment_size = 50                                              2
max_iterations = 100                                           3

chroma = tf.placeholder(tf.float32)
max_freqs = tf.argmax(chroma, 0)

def get_chromagram(audio_file):
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)
    return F.X

def get_dataset(sess, audio_file):                             4
    chroma_data = get_chromagram(audio_file)
    print('chroma_data', np.shape(chroma_data))
    chroma_length = np.shape(chroma_data)[1]
    xs = []
    for i in range(chroma_length / segment_size):
        chroma_segment = chroma_data[:, i*segment_size:(i+1)*segment_size]
        x = extract_feature_vector(sess, chroma_segment)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

- ***1*** **Decides the number of clusters**
- ***2*** **The smaller the segment size, the better the results (but slower performance).**
- ***3*** **Decides when to stop the iterations**
- ***4*** **Obtains a dataset by extracting segments of the audio as separate data items**

Now run k-means clustering on this dataset to identify when segments are similar. The intention is that k-means will categorize similar-sounding segments with the same label. If two people have significantly different-sounding voices, their sound snippets will belong to different labels.

Listing 5.6. Segmenting an audio clip

```
with tf.Session() as sess:
    X = get_dataset(sess, 'TalkingMachinesPodcast.wav')
    print(np.shape(X))
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:            1
        i += 1
        Y = assign_cluster(X, centroids)
```

```
        centroids = sess.run(recompute_centroids(X, Y))
        if i % 50 == 0:
            print('iteration', i)
    segments = sess.run(Y)
    for i in range(len(segments)):                          2
        seconds = (i * segment_size) / float(10)
        min, sec = divmod(seconds, 60)
        time_str = '{}m {}s'.format(min, sec)
        print(time_str, segments[i])
```

- ***1* Runs the k-means algorithm**
- ***2* Prints the labels for each time interval**

The output of running listing 5.6 is a list of timestamps and cluster IDs that correspond to who is talking during the podcast:

```
('0.0m 0.0s', 0)
('0.0m 2.5s', 1)
('0.0m 5.0s', 0)
('0.0m 7.5s', 1)
('0.0m 10.0s', 1)
('0.0m 12.5s', 1)
('0.0m 15.0s', 1)
('0.0m 17.5s', 0)
('0.0m 20.0s', 1)
('0.0m 22.5s', 1)
('0.0m 25.0s', 0)
('0.0m 27.5s', 0)
```

Exercise 5.3

How can you detect whether the clustering algorithm has converged (so that you can stop the algorithm early)?

ANSWER

One way is to monitor how the cluster centroids change, and declare convergence once no more updates are necessary (for example, when the difference in the size of the error isn't changing significantly between iterations). To do this, you'd need to calculate the size of the error and decide what constitutes "significantly."

5.5. CLUSTERING USING A SELF-ORGANIZING MAP

A *self-organizing map* (SOM) is a model for representing data into a lower-dimensional space. In doing so, it automatically shifts similar data items closer together. For example, suppose you're ordering pizza for a large gathering of people. You don't want to order the same type of pizza for every single person—because one might happen to fancy pineapple with mushrooms and peppers for their toppings, and you may prefer anchovies with arugula and onions.

Each person's preference of toppings can be represented as a three-dimensional vector. An SOM lets you embed these three-dimensional vectors in two dimensions (as long as

you define a distance metric between pizzas). Then, a visualization of the two-dimensional plot reveals good candidates for the number of clusters.

Although it may take longer to converge than the k-means algorithm, the SOM approach has no assumptions about the number of clusters. In the real world, it's hard to select a value for the number of clusters. Consider a gathering of people, as shown in figure 5.8, in which the clusters change over time.

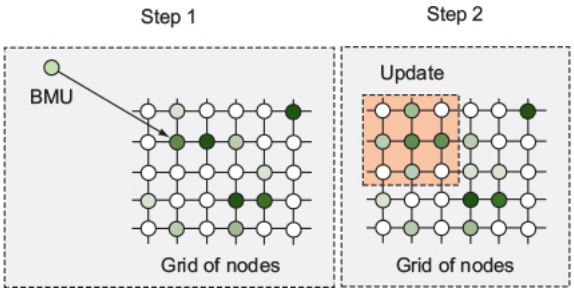
Figure 5.8. In the real world, we see groups of people in clusters all the time. Applying k-means requires knowing the number of clusters ahead of time. A more flexible tool is a self-organizing map, which has no preconceptions about the number of clusters.

The SOM merely reinterprets the data into a structure conducive to clustering. The algorithm works as follows. First, you design a grid of nodes; each node holds a weight vector of the same dimension as a data item. The weights of each node are initialized to random numbers, typically from a standard normal distribution.

Next, you show data items to the network one by one. For each data item, the network identifies the node whose weight vector most closely matches it. This node is called the *best matching unit* (BMU).

After the network identifies the BMU, all neighbors of the BMU are updated so their weight vectors move closer to the BMU's value. The closer nodes are affected more strongly than nodes farther away. Moreover, the number of neighbors around a BMU shrinks over time at a rate determined usually by trial and error. Figure 5.9 illustrates the algorithm.

Figure 5.9. One iteration of the SOM algorithm. The first step is to identify the best matching unit (BMU), and the second step is to update the neighboring nodes. You keep iterating these two steps with training data until certain convergence criteria are reached.



The following listing shows how to start implementing a SOM in TensorFlow. Follow along by opening a new source file.

Listing 5.7. Setting up the SOM algorithm

```
import tensorflow as tf
import numpy as np

class SOM:
    def __init__(self, width, height, dim):
        self.num_iters = 100
        self.width = width
        self.height = height
        self.dim = dim
        self.node_locs = self.get_locs()

        nodes = tf.Variable(tf.random_normal([width*height, dim]))     1
        self.nodes = nodes

        x = tf.placeholder(tf.float32, [dim])                          2
        iter = tf.placeholder(tf.float32)                              2

        self.x = x                                                     3
        self.iter = iter                                               3

        bmu_loc = self.get_bmu_loc(x)                                  4

        self.propagate_nodes = self.get_propagation(bmu_loc, x, iter)  5
```

- ***1*** **Each node is a vector of dimension dim. For a 2D grid, there are width × height nodes; get_locs is defined in listing 5.10.**
- ***2*** **These two ops are inputs at each iteration.**
- ***3*** **You'll need to access them from another method.**
- ***4*** **Finds the node that most closely matches the input (in listing 5.9)**
- ***5*** **Updates the values of the neighbors (in listing 5.8)**

In the next listing, you define how to update neighboring weights, given the current time interval and BMU location. As time goes by, the BMU's neighboring weights are less influenced to change. That way, over time the weights gradually settle.

Listing 5.8. Defining how to update the values of neighbors

```
    def get_propagation(self, bmu_loc, x, iter):
        num_nodes = self.width * self.height
        rate = 1.0 - tf.div(iter, self.num_iters)                          1
        alpha = rate * 0.5
        sigma = rate * tf.to_float(tf.maximum(self.width, self.height)) / 2.
        expanded_bmu_loc = tf.expand_dims(tf.to_float(bmu_loc), 0)         2
        sqr_dists_from_bmu = tf.reduce_sum(
          tf.square(tf.subtract(expanded_bmu_loc, self.node_locs)), 1)
        neigh_factor =                                                     3
          tf.exp(-tf.div(sqr_dists_from_bmu, 2 * tf.square(sigma)))
        rate = tf.multiply(alpha, neigh_factor)
        rate_factor =
          tf.stack([tf.tile(tf.slice(rate, [i], [1]),
                  [self.dim]) for i in range(num_nodes)])
        nodes_diff = tf.multiply(
          rate_factor,
          tf.subtract(tf.stack([x for i in range(num_nodes)]), self.nodes))
        update_nodes = tf.add(self.nodes, nodes_diff)                      4
        return tf.assign(self.nodes, update_nodes)                         5
```

- ***1*** **The rate decreases as iter increases. This value influences the alpha and sigma parameters.**
- ***2*** **Expands bmu_loc, so you can efficiently compare it pairwise with each element of node_locs**
- ***3*** **Ensures that nodes closer to the BMU change more dramatically**
- ***4*** **Defines the updates**
- ***5*** **Returns an op to perform the updates**

The following listing shows how to find the BMU location, given an input data item. It searches through the grid of nodes to find the one with the closest match. This is similar to the assignment step in k-means clustering, where each node in the grid is a potential cluster centroid.

Listing 5.9. Getting the node location of the closest match

```
def get_bmu_loc(self, x):
    expanded_x = tf.expand_dims(x, 0)
    sqr_diff = tf.square(tf.subtract(expanded_x, self.nodes))
    dists = tf.reduce_sum(sqr_diff, 1)
    bmu_idx = tf.argmin(dists, 0)
    bmu_loc = tf.stack([tf.mod(bmu_idx, self.width), tf.div(bmu_idx,
    ➥ self.width)])
    return bmu_loc
```

In the next listing, you create a helper method to generate a list of (x, y) locations on all the nodes in the grid.

Listing 5.10. Generating a matrix of points

```
def get_locs(self):
    locs = [[x, y]
             for y in range(self.height)
             for x in range(self.width)]
    return tf.to_float(locs)
```

Finally, let's define a method called `train` to run the algorithm, as shown in listing 5.11. First, you must set up the session and run the `global_variables_initializer` op. Next, you loop `num_iters` a certain number of times to update weights using the input data one by one. After the loop ends, you record the final node weights and their locations.

Listing 5.11. Running the SOM algorithm

```
def train(self, data):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for i in range(self.num_iters):
            for data_x in data:
                sess.run(self.propagate_nodes, feed_dict={self.x: data_x,
                ➥ self.iter: i})
        centroid_grid = [[] for i in range(self.width)]
        self.nodes_val = list(sess.run(self.nodes))
        self.locs_val = list(sess.run(self.node_locs))
        for i, l in enumerate(self.locs_val):
```

```
                centroid_grid[int(l[0])].append(self.nodes_val[i])
            self.centroid_grid = centroid_grid
```

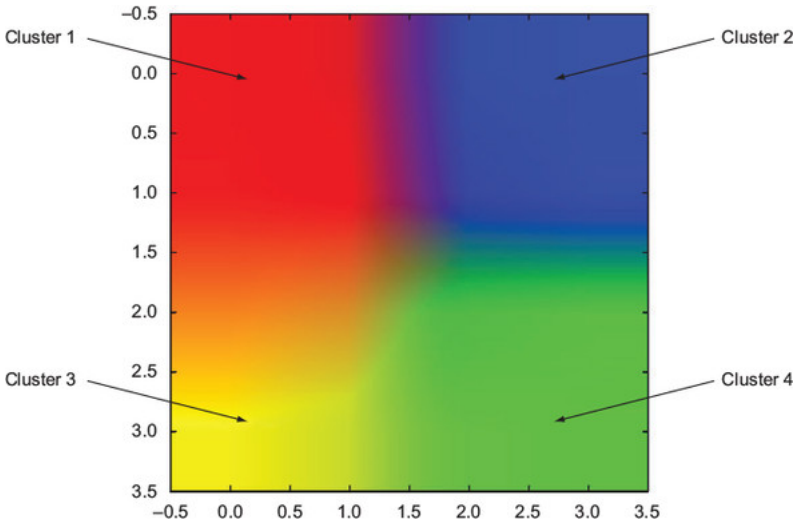
That's it! Now let's see it in action. Test the implementation by showing the SOM some input. In listing 5.12, the input is a list of three-dimensional feature vectors. By training the SOM, you'll discover clusters within the data. You'll use a 4 × 4 grid, but it's best to try various values to cross-validate the best grid size. Figure 5.10 shows the output of running the code.

Listing 5.12. Testing the implementation and visualizing the results

```
from matplotlib import pyplot as plt
import numpy as np
from som import SOM

colors = np.array(
     [[0., 0., 1.],
      [0., 0., 0.95],
      [0., 0.05, 1.],
      [0., 1., 0.],
      [0., 0.95, 0.],
      [0., 1, 0.05],
      [1., 0., 0.],
      [1., 0.05, 0.],
      [1., 0., 0.05],
      [1., 1., 0.]])

som = SOM(4, 4, 3)             1
som.train(colors)

plt.imshow(som.centroid_grid)
plt.show()
```

- *1* **The grid size is 4 × 4, and the input dimension is 3.**

Figure 5.10. The SOM places all three-dimensional data points into a two-dimensional grid. From it, you can pick the cluster centroids (automatically or manually) and achieve clustering in an intuitive lower-dimensional space.



The SOM embeds higher-dimensional data into 2D to make clustering easy. This acts as a handy preprocessing step. You can manually go in and indicate the cluster centroids by observing the SOM's output, but it's also possible to automatically find good centroid candidates by observing the gradient of the weights. For the

adventurous, we suggest reading the famous paper "Clustering of the Self-Organizing Map," by Juha Vesanto and Esa Alhoniemi: http://mng.bz/XzyS (http://mng.bz/XzyS).

5.6. APPLICATION OF CLUSTERING

You've already seen two practical applications of clustering: organizing music and segmenting an audio clip to label similar sounds. Clustering is especially helpful when the training dataset doesn't contain corresponding labels. As you know, such a situation characterizes unsupervised learning. Sometimes, data is just too inconvenient to annotate.

For example, suppose you want to understand sensor data from the accelerometer of a phone or smartwatch. At each time step, the accelerometer provides a three-dimensional vector, but you have no idea whether the human is walking, standing, sitting, dancing, jogging, or so on. You can obtain such a dataset at http://mng.bz/rTMe (http://mng.bz/rTMe).

To cluster the time-series data, you'll need to summarize the list of accelerometer vectors into a concise feature vector. One way is to generate a histogram of differences between consecutive magnitudes of the acceleration. The derivative of acceleration is called *jerk*, and you can apply the same operation to obtain a histogram outlining differences in jerk magnitudes.

This process of generating a histogram out of data is exactly like the preprocessing steps on audio data explained in this chapter. After you've transformed the histograms into feature vectors, you can use the same code listings taught earlier (such as k-means in TensorFlow).

Note

Whereas previous chapters discussed supervised learning, this chapter focused on unsupervised learning. In the next chapter, you'll see a machine-learning algorithm that is neither of the two. It's a modeling framework that doesn't get much attention by programmers nowadays but is the essential tool for statisticians for unveiling hidden factors in data.

5.7. SUMMARY

- Clustering is an unsupervised machine-learning algorithm for discovering structure in data.
- K-means clustering is one of the easiest algorithms to implement and understand, and it also performs well in terms of speed and accuracy.
- If the number of clusters isn't specified, you can use the self-organizing map (SOM) algorithm to view the data in a simplified perspective.