Bill = {0, 0, 0, 1, 1, 1, 1, 0, 1, 0 ... }

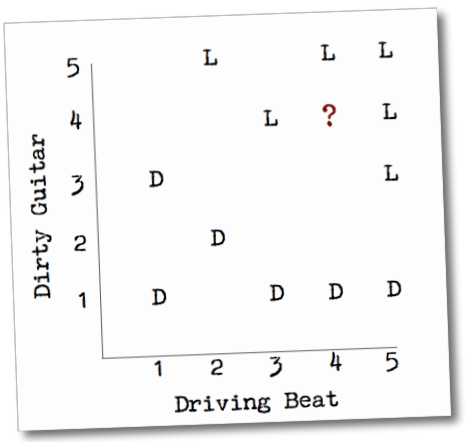
Obviously there is no need to normalize this data. What about the Pandora case: all variables lie on a scale from 1 to 5 inclusive. Should we normalize or not? It probably wouldn't hurt the accuracy of the algorithm if we normalized, but keep in mind that there is a computational cost involved with normalizing. In this case, we might empirically compare results between using the regular and normalized data and select the best performing approach. Later in this chapter we will see a case where normalization reduces accuracy.

Back to Pandora

In the Pandora inspired example, we had each song represented by a number of attributes. If a user creates a radio station for Green Day we decide what to play based on a nearest neighbor approach. Pandora allows a user to give a particular tune a thumbs up or thumbs down rating. How do we use the information that a user gives a thumbs up for a particular song.?

Suppose I use 2 attributes for songs: the amount of dirty guitar and the presence of a driving beat both rated on a 1-5 scale. A user has given the thumbs up to 5 songs indicating he liked the song (and indicated on the following chart with a 'L'); and a thumbs down to 5 songs indicating he disliked the song (indicated by a 'D').

Do you think the user will like or dislike the song indicated by the '?' in this chart?



I am guessing you said he would like the song. We base this on the fact that the '?' is closer to the Ls in the chart than the Ds. We will spend the rest of this chapter and the next describing computational approaches to this idea. The most obvious approach is to find the nearest neighbor of the "?" and predict that it will share the class of the nearest neighbor. The question mark's nearest neighbor is an L so we would predict that the '? tune' is something the user would like.

The Python nearest neighbor classifier code

Let's use the example dataset I used earlier—ten tunes rated on 7 attributes (amount of piano, vocals, driving beat, blues influence, dirty electric guitar, backup vocals, rap influence).

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1
Todd Snider/ Don't Tempt Me	4	5	4	4	1	5	1
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1
Black Eyed Peas/ Rock that Body	2	5	5	1	2	2	4
La Roux/ Bulletproof	5	5	4	2	1	1	1
Mike Posner/ Cooler than me	2.5	4	4	1	1	1	1
Lady Gaga/ Alejandro	1	5	3	2	1	2	1

Earlier in this chapter we developed a Python representation of this data:

```
music = {"Dr Dog/Fate": {"piano": 2.5, "vocals": 4, "beat": 3.5,
                         "blues": 3, "guitar": 5, "backup vocals": 4,
                         "rap": 1},
         "Phoenix/Lisztomania": {"piano": 2, "vocals": 5, "beat": 5,
                                 "blues": 3, "guitar": 2,
                                 "backup vocals": 1, "rap": 1},
         "Heartless Bastards/Out at Sea": {"piano": 1, "vocals": 5,
                                           "beat": 4, "blues": 2,
                                           "guitar": 4,
                                           "backup vocals": 1,
                                           "rap": 1},
         "Todd Snider/Don't Tempt Me": {"piano": 4, "vocals": 5,
                                        "beat": 4, "blues": 4,
                                        "guitar": 1,
                                        "backup vocals": 5, "rap": 1},
```

Here the strings *piano, vocals, beat, blues, guitar, backup vocals*, and *rap* occur multiple times; if I have a 100,000 tunes those strings are repeated 100,000 times. I'm going to remove those strings from the representation of our data and simply use vectors:

```
#
#  the item vector represents the attributes: piano, vocals,
#  beat, blues, guitar, backup vocals, rap
#
items = {"Dr Dog/Fate": [2.5, 4, 3.5, 3, 5, 4, 1],
         "Phoenix/Lisztomania": [2, 5, 5, 3, 2, 1, 1],
         "Heartless Bastards/Out at Sea": [1, 5, 4, 2, 4, 1, 1],
         "Todd Snider/Don't Tempt Me": [4, 5, 4, 4, 1, 5, 1],
         "The Black Keys/Magic Potion": [1, 4, 5, 3.5, 5, 1, 1],
         "Glee Cast/Jessie's Girl": [1, 5, 3.5, 3, 4, 5, 1],
         "La Roux/Bulletproof": [5, 5, 4, 2, 1, 1, 1],
         "Mike Posner": [2.5, 4, 4, 1, 1, 1, 1],
         "Black Eyed Peas/Rock That Body": [2, 5, 5, 1, 2, 2, 4],
         "Lady Gaga/Alejandro": [1, 5, 3, 2, 1, 2, 1]}
```

In linear algebra, a vector is a quantity that has magnitude and direction.
Various well defined operators can be performed on vectors including adding and subtracting vectors and scalar multiplication.

In data mining, a vector is simply a list of numbers that represent the attributes of an object. The example on the previous page represented attributes of a song as a list of numbers. Another example, would be representing a text document as a vector--each position of the vector would represent a particular word and the number at that position would represent how many times that word occurred in the text.

Plus, using the word "vector" instead of "list of attributes" is cool!

Once we define attributes this way, we can perform vector operations (from linear algebra) on them.

In addition to representing the attributes of a song as a vector, I need to represent the thumbs up/ thumbs down ratings that users gives to songs. Because each user doesn't rate all songs (sparse data) I will go with the dictionary of dictionaries approach:

```
users = {"Angelica": {"Dr Dog/Fate": "L", "Phoenix/Lisztomania": "L",
                      "Heartless Bastards/Out at Sea": "D",
                      "Todd Snider/Don't Tempt Me": "D",
                      "The Black Keys/Magic Potion": "D",
                      "Glee Cast/Jessie's Girl": "L",
                      "La Roux/Bulletproof": "D",
                      "Mike Posner": "D",
                      "Black Eyed Peas/Rock That Body": "D",
                      "Lady Gaga/Alejandro": "L"},
         "Bill":  {"Dr Dog/Fate": "L", "Phoenix/Lisztomania": "L",
                   "Heartless Bastards/Out at Sea": "L",
                   "Todd Snider/Don't Tempt Me": "D",
                   "The Black Keys/Magic Potion": "L",
                   "Glee Cast/Jessie's Girl": "D",
                   "La Roux/Bulletproof": "D", "Mike Posner": "D",
                   "Black Eyed Peas/Rock That Body": "D",
                   "Lady Gaga/Alejandro": "D"}             }
```

My way of representing 'thumbs up' as *L* for *like* and 'thumbs down' as *D* is arbitrary. You could use 0 and 1, *like* and *dislike*.

In order to use the new vector format for songs I need to revise the Manhattan Distance and the computeNearestNeighbor functions.

```
def manhattan(vector1, vector2):
   """Computes the Manhattan distance."""
   distance = 0
   total = 0
   n = len(vector1)
   for i in range(n):
      distance += abs(vector1[i] - vector2[i])
   return distance
```

```
def computeNearestNeighbor(itemName, itemVector, items):
   """creates a sorted list of items based on their distance to item"""
   distances = []
   for otherItem in items:
      if otherItem != itemName:
         distance = manhattan(itemVector, items[otherItem])
         distances.append((distance, otherItem))
   # sort based on distance -- closest first
   distances.sort()
   return distances
```

Finally, I need to create a classify function. I want to predict how a particular user would rate an item represented by itemName and itemVector. For example:

```
"Chris Cagle/ I Breathe In. I Breathe Out"  [1, 5, 2.5, 1, 1, 5, 1]
```

(NOTE: To better format the Python example below, I will use the string *Cagle* to represent that singer and song pair.)

The first thing the function needs to do is find the nearest neighbor of this Chris Cagle tune. Then it needs to see how the user rated that nearest neighbor and predict that the user will rate Chris Cagle the same. Here's my rudimentary classify function:

```
def classify(user, itemName, itemVector):
   """Classify the itemName based on user ratings
   Should really have items and users as parameters"""
   # first find nearest neighbor
   nearest = computeNearestNeighbor(itemName, itemVector, items)[0][1]
   rating = users[user][nearest]
   return rating
```

Ok. Let's give this a try. I wonder if Angelica will like Chris Cagle's I Breathe In, I Breathe Out?

```
classify('Angelica', 'Cagle', [1, 5, 2.5, 1, 1, 5, 1])
"L"
```

We are predicting she will like it! Why are we predicting that?

```
computeNearestNeighbor('Angelica', 'Cagle', [1, 5, 2.5, 1, 1, 5, 1])

[(4.5, 'Lady Gaga/Alejandro'), (6.0, "Glee Cast/Jessie's Girl"), (7.5,
"Todd Snider/Don't Tempt Me"), (8.0, 'Mike Posner'), (9.5, 'Heartless
Bastards/Out at Sea'), (10.5, 'Black Eyed Peas/Rock That Body'), (10.5,
'Dr Dog/Fate'), (10.5, 'La Roux/Bulletproof'), (10.5, 'Phoenix/
Lisztomania'), (14.0, 'The Black Keys/Magic Potion')]
```

We are predicting that Angelica will like Chris Cagle's *I Breathe In, I Breathe Out* because that tune's nearest neighbor is Lady Gaga's *Alejandro* and Angelica liked that tune.

What we have done here is build a classifier—in this case, our task was to classify tunes as belonging to one of two groups—the like group and the dislike group.

A classifier is a program that uses an object's attributes to determine what group or class it belongs to!

A classifier uses a set of objects that are already labeled with the class they belong to. It uses that set to classify new, unlabeled objects. So in our example, we knew about songs that Angelica liked (labeled 'liked') and songs she did not like. We wanted to predict whether Angelica would like a Chris Cagle tune.

First we found a song Angelica rated that was most similar to the Chris Cagle tune.

It was Lady Gaga's *Alejandro*

Next, we checked whether Angelica liked or disliked the *Alejandro*—she liked it. So we predict that Angelica will also like the Chris Cagle tune, *I Breathe In, I Breathe Out*.

Classifiers can be used in a wide range of applications. The following page lists just a few.

Twitter Sentiment Classification

A number of people are working on classifying the sentiment (a positive or negative opinion) in tweets. This can be used in a variety of ways. For example, if Axe releases a new underarm deodorant, they can check whether people like it or not. The attributes are the words in the tweet.

Classification for Targeted Political Ads

This is called microtargeting. People are classified into such groups as "Barn Raisers", "Inner Compass", and "Hearth Keepers." Hearth Keepers, for example, focus on their family and keep to themselves.

Health and the Quantified Self

It's the start of the quanitifed self explosion. We can now buy simple devices like the Fitbit, and the Nike Fuelband. Intel and other companies are working on intelligent homes that have floors that can weigh us, keep track of our movements and alert someone if we deviate from normal. Experts are predicting that in a few years we will be wearing tiny compu-patches that can monitor dozens of factors in real time and make instant classifications.

Automatic identification of people in photos.

There are apps now that can identify and tag your friends in photographs. (And the same techniques apply to identifying people walking down the street using public video cams.) Techniques vary but some of them use attributes like the relative position and size of a person's eyes, nose, jaw, etc.

Targeted Marketing

Similar to political microtargeting. Instead of a broad advertising campaign to sell my expensive Vegas time share luxury condos, can I identify likely buyers and market just to them? Even better if I can identify subgroups of likely buyers and I can really tailor my ads to specific groups.

The list is endless

- classifying people as terrorist or nonterrorist
- automatic classification of email (hey, this email looks pretty important; this is regular email; this looks like spam)
- predicting medical clinical outcomes
- identifying financial fraud (for ex., credit card fraud)

What sport?

To give you a preview of what we will be working on in the next few chapters let us work with an easier example than those given on the previous page—classifying what sport various world-class women athletes play based solely on their height and weight. In the following table I have a small sample dataset drawn from a variety of web sources.

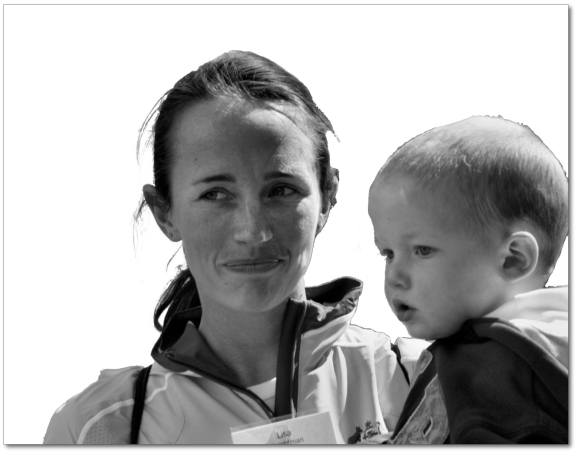
Name	Sport	Age	Height	Weight
Asuka Teramoto	Gymnastics	16	54	66
Brittainey Raven	Basketball	22	72	162
Chen Nan	Basketball	30	78	204
Gabby Douglas	Gymnastics	16	49	90
Helalia Johannes	Track	32	65	99
Irina Miketenko	Track	40	63	106
Jennifer Lacy	Basketball	27	75	175
Kara Goucher	Track	34	67	123
Linlin Deng	Gymnastics	16	54	68
Nakia Sanford	Basketball	34	76	200
Nikki Blue	Basketball	26	68	163
Qiushuang Huang	Gymnastics	20	61	95
Rebecca Tunney	Gymnastics	16	58	77
Rene Kalmer	Track	32	70	108
Shanna Crossley	Basketball	26	70	155
Shavonte Zellous	Basketball	24	70	155
Tatyana Petrova	Track	29	63	108
Tiki Gelana	Track	25	65	106
Valeria Straneo	Track	36	66	97
Viktoria Komova	Gymnastics	17	61	76

The gymnastic data lists some of the top participants in the 2012 and 2008 Olympics. The basketball players play for teams in the WNBA. The women track stars were finishers in the 2012 Olympic marathon . Granted this is a trivial example but it will allow us to apply some of the techniques we have learned.

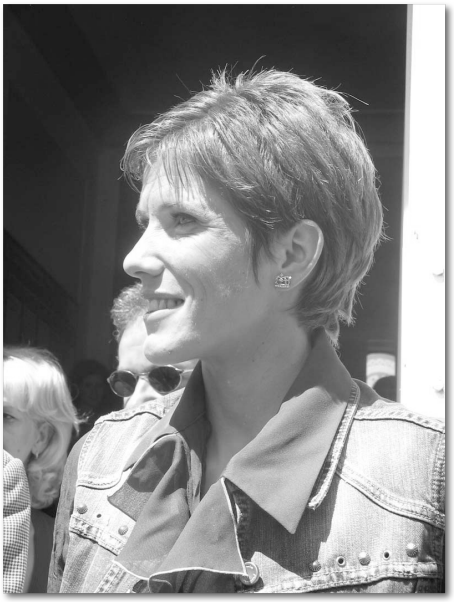
As you can see, I've included age in the table. Just scanning the data you can see that age alone is a moderately good predictor. Try to guess the sports of these athletes.

Candace Parker; Age 26

McKayla Maroney; Age 16



Lisa Jane Weightman; Age 34



Olivera Jevtić: Age 35

The answers

Candace Parker plays basketball for the WNBA's Los Angeles Sparks and Russia's UMMC Ekaterinburg. McKayla Maroney was a member of the U.S. Women's Gymnastic Team and won a Gold and a Silver. Olivera Jevtić is a Serbian long-distance runner who competed in the 2008 and 2012 Olympics. Lisa Jane Weightman is an Australian long-distance runner who also competed in the 2008 and 2012 Olympics.

You just performed classification—you predicted the class of objects based on their attributes. (In this case, predicting the sport of athletes based on a single attribute, age.)

brain calisthenics

Suppose I want to guess what sport a person plays based on their height and weight. My database is small—only two people. Nakia Sanford, the center for the Women's National Basketball Association team Phoenix Mercury, is 6'4" and weighs 200 pounds. Sarah Beale, a forward on England's National Rugby Team, is 5'10" and weighs 190. Based on that database, I want to classify Catherine Spencer as either a basketball player or rugby player. She is 5'10" and weighs 200 pounds. What sport do you think she plays?

brain calisthenics - cont'd

If you said rugby, you would be correct. Catherine Spencer is a forward on England's national team. However, if we based our guess on a distance formula like Manhattan Distance we would be wrong. The Manhattan Distance between Catherine and Basketball player Nakia is 6 (they weigh the same and have a six inch difference in height). The distance between Catherine and Rugby player Sarah is 10 (their height is the same and they differ in weight by 10 pounds). So we would pick the closest person, Nakia, and predict Catherine plays the same sport.

Is there anything we learned that could help us make more accurate classifications?

brain calisthenics - cont'd

We can use the Modified Standard Score!!!

$$\frac{\text{(each value) - (median)}}{\text{(absolute standard deviation)}}$$

Test Data.

Let us remove age from the picture. Here is a group of individuals I would like to classify:

Name	Sport	Height	Weight
Crystal Langhorne		74	190
Li Shanshan		64	101
Kerri Strug		57	87
Jaycie Phelps	`	60	97
Kelly Miller		70	140
Zhu Xiaolin		67	123
Lindsay Whalen		69	169
Koko Tsurumi		55	75
Paula Radcliffe		68	120
Erin Thorn		69	144

Let's build a classifier!

Python Coding

Instead of hard-coding the data in the Python code, I decided to put the data for this example into two files: athletesTrainingSet.txt and athletesTestSet.txt.

I am going to use the data in the athletesTrainingSet.txt file to build the classifier. The data in the athletesTestSet.txt file will be used to evaluate this classifier. In other words, each entry in the test set will be classified by using all the entries in the training set.

The data files and the Python code are on the book's website, guidetodatamining.com.

The format of these files looks like this:

Asuka Teramoto	Gymnastics	54	66
Brittainey Raven	Basketball	72	162
Chen Nan	Basketball	78	204
Gabby Douglas	Gymnastics	49	90

Each line of the text represents an object described as a tab-separated list of values. I want my classifier to use a person's height and weight to predict what sport that person plays. So the last two columns are the numerical attributes I will use in the classifier and the second column represents the class that object is in. The athlete's name is not used by the classifier. I don't try to predict what sport a person plays based on their name and I am not trying to predict the name from some attributes.

However, keeping the name might be useful as a means of explaining the classifier's decision to users: "We think Amelia Pond is a gymnast because she is closest in height and weight to Gabby Douglas who is a gymnast."

As I said, I am going to write my Python code to not be so hard coded to a particular example (for example, to only work for the athlete example). To help meet this goal I am going to add an initial header line to the athlete training set file that will indicate the function of each column. Here are the first few lines of that file:

comment	class	num	num
Asuka Teramoto	Gymnastics	54	66
Brittainey Raven	Basketball	72	162

Any column labeled *comment* will be ignored by the classifier; a column labeled *class* represents the class of the object, and columns labeled *num* indicate numerical attributes of that object.

brain calisthenics -

How do you think we should represent this data in Python? Here are some possibilities (or come up with your own representation).

a dictionary of the form:

```
{'Asuka Termoto': ('Gymnastics', [54, 66]),
 'Brittainey Raven': ('Basketball', [72, 162]), ...
```

a list of lists of the form:

```
[['Asuka Termoto', 'Gymnastics', 54, 66],
 ['Brittainey Raven', 'Basketball', 72, 162], ...
```

a list of tuples of the form:

```
[('Gymnastics', [54, 66], ['Asuka Termoto']),
 ('Basketball', [72, 162], ['Brittainey Raven'],...
```

brain calisthenics - answer

a dictionary of the form:

```
{'Asuka Termoto': ('Gymnastics', [54, 66]),
 'Brittainey Raven': ('Basketball', [72, 162]), ...
```

This is not a very good representation of our data. The key for the dictionary is the athlete's name, which we do not even use in the calculations.

a list of lists of the form:

```
[['Asuka Termoto', 'Gymnastics', 54, 66],
 ['Brittainey Raven', 'Basketball', 72, 162], ...
```

This is not a bad representation. It mirrors the input file and since the nearest neighbor algorithm requires us to iterate through the list of objects, a list makes sense.

a list of tuples of the form:

```
[('Gymnastics', [54, 66], ['Asuka Termoto']),
 ('Basketball', [72, 162], ['Brittainey Raven'],...
```

I like this representation better than the above since it separates the attributes into their own list and makes the division between class, attributes, and comments precise. I made the comment (the name in this case) a list since there could be multiple columns that are comments.

My python code that reads in a file and converts it to the format

```
[('Gymnastics', [54, 66], ['Asuka Termoto']),
 ('Basketball', [72, 162], ['Brittainey Raven'],...
```

looks like this:

```
class Classifier:

    def __init__(self, filename):

        self.medianAndDeviation = []

        # reading the data in from the file
        f = open(filename)
        lines = f.readlines()
        f.close()
        self.format = lines[0].strip().split('\t')
        self.data = []
        for line in lines[1:]:
            fields = line.strip().split('\t')
            ignore = []
            vector = []
            for i in range(len(fields)):
                if self.format[i] == 'num':
                    vector.append(int(fields[i]))
                elif self.format[i] == 'comment':
                    ignore.append(fields[i])
                elif self.format[i] == 'class':
                    classification = fields[i]
            self.data.append((classification, vector, ignore))
```

code it

Before we can standardize the data using the Modified Standard Score we need methods that will compute the median and absolute standard deviation of numbers in a list:


```
>>> heights = [54, 72, 78, 49, 65, 63, 75, 67, 54]
>>> median = classifier.getMedian(heights)
>>> median
65
>>> asd = classifier.getAbsoluteStandardDeviation(heights, median)
>>> asd
8.0
```

Can you write these methods?

AssertionError?

See next page

Download the template testMedianAndASD.py to write and test these methods at guidetodatamining.com

Assertion Errors and the Assert statement.

It is important that each component of a solution to a problem be turned into a piece of code that implements it and a piece of code that tests it. In fact, it is good practice to write the test code before you write the implementation. The code template I have provided contains a test function called `unitTest.` A simplified version of that function, showing only one test, is shown here:

```
def unitTest():
    list1 = [54, 72, 78, 49, 65, 63, 75, 67, 54]
    classifier = Classifier('athletesTrainingSet.txt')
    m1 = classifier.getMedian(list1)
    assert(round(m1, 3) == 65)
    print("getMedian and getAbsoluteStandardDeviation work correctly")
```

The getMedian function you are to complete initially looks like this:

```
def getMedian(self, alist):
        """return median of alist"""

        """TO BE DONE"""
        return 0
```

So initially, `getMedian` returns 0 as the median for any list. You are to complete getMedian so it returns the correct value. In the `unitTest` procedure, I call getMedian with the list

```
[54, 72, 78, 49, 65, 63, 75, 67, 54]
```

The assert statement in `unitTest` says the value returned by `getMedian` should equal 65. If it does, execution continues to the next line and

```
getMedian and getAbsoluteStandardDeviation work correctly
```

is printed. If they are not equal the program terminates with an error:

```
File "testMedianAndASD.py", line 78, in unitTest
    assert(round(m1, 3) == 65)
AssertionError
```

If you download the code from the book's website and run it without making any changes, you will get this error. Once you have correctly implemented `getMedian` and `getAbsoluteStandardDeviation` this error will disappear.

This use of assert as a means of testing software components is a common technique among software developers.

"it is important that each part of the specification be turned into a piece of code that implements it and a test that tests it. If you don't have tests like these then you don't know when you are done, you don't know if you got it right, and you don't know that any future changes might be breaking something." - Peter Norvig

Solution

Here is one way of writing these algorithms:

```
  def getMedian(self, alist):
        """return median of alist"""
        if alist == []:
            return []
        blist = sorted(alist)
        length = len(alist)
        if length % 2 == 1:
            # length of list is odd so return middle element
            return blist[int(((length + 1) / 2) -  1)]
        else:
            # length of list is even so compute midpoint
            v1 = blist[int(length / 2)]
            v2 =blist[(int(length / 2) - 1)]
            return (v1 + v2) / 2.0


    def getAbsoluteStandardDeviation(self, alist, median):
        """given alist and median return absolute standard deviation"""
        sum = 0
        for item in alist:
            sum += abs(item - median)
        return sum / len(alist)
```

As you can see my getMedian method first sorts the list before finding the median. Because I am not working with huge data sets I think this is a fine solution. If I wanted to optimize my code, I might replace this with a selection algorithm.

Right now, the data is read from the file athletesTrainingSet.txt and stored in the list `data` in the classifier with the following format:

```
[('Gymnastics', [54, 66], ['Asuka Teramoto']),
 ('Basketball', [72, 162], ['Brittainey Raven']),
 ('Basketball', [78, 204], ['Chen Nan']),
 ('Gymnastics', [49, 90], ['Gabby Douglas']), ...
```

Now I would like to normalize the vector so the list `data` in the classifier contains normalized values. For example,

```
[('Gymnastics', [-1.93277, -1.21842], ['Asuka Teramoto']),
 ('Basketball', [1.09243, 1.63447], ['Brittainey Raven']),
 ('Basketball', [2.10084, 2.88261], ['Chen Nan']),
 ('Gymnastics', [-2.77311, -0.50520], ['Gabby Douglas']),
 ('Track', [-0.08403, -0.23774], ['Helalia Johannes']),
 ('Track', [-0.42017, -0.02972], ['Irina Miketenko']),
```

To do this I am going to add the following lines to my init method:

```
        # get length of instance vector
        self.vlen = len(self.data[0][1])
        # now normalize the data
        for i in range(self.vlen):
            self.normalizeColumn(i)
```

In the for loop we want to normalize the data, column by column. So the first time through the loop we will normalize the height column, and the next time through, the weight column.

code it

Can you write the normalizeColumn method?

Download the template normalizeColumnTemplate.py to write and test this method at guidetodatamining.com

Solution

Here is an implementation of the normalizeColumn method:

```
def normalizeColumn(self, columnNumber):
  """given a column number, normalize that column in self.data"""
  # first extract values to list
  col = [v[1][columnNumber] for v in self.data]
  median = self.getMedian(col)
  asd = self.getAbsoluteStandardDeviation(col, median)
  #print("Median: %f   ASD = %f" % (median, asd))
  self.medianAndDeviation.append((median, asd))
  for v in self.data:
     v[1][columnNumber] = (v[1][columnNumber] - median) / asd
```

You can see I also store the median and absolute standard deviation of each column in the list `medianAndDeviation`. I use this information when I want to use the classifier to predict the class of a new instance. For example, suppose I want to predict what sport is played by Kelly Miller, who is 5 feet 10 inches and weighs 170. The first step is to convert her height and weight to Modified Standard Scores. That is, her original attribute vector is [70, 140].

After processing the training data, the value of meanAndDeviation is

```
[(65.5, 5.95), (107.0, 33.65)]
```

meaning the data in the first column of the vector has a median of 65.5 and an absolute standard deviation of 5.95; the second column has a median of 107 and a deviation of 33.65.

I use this info to convert the original vector [70,140] to one containing Modified Standard Scores. This computation for the first attribute is

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{70 - 65.5}{5.95} = \frac{4.5}{5.95} = 0.7563$$

and the second:

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{140 - 107}{33.65} = \frac{33}{33.65} = 0.98068$$

The python method that does this is:

```
    def normalizeVector(self, v):
        """We have stored the median and asd for each column.
        We now use them to normalize vector v"""
        vector = list(v)
        for i in range(len(vector)):
            (median, asd) = self.medianAndDeviation[i]
            vector[i] = (vector[i] - median) / asd
        return vector
```

The final bit of code to write is the part that predicts the class of a new instance—in our current example, the sport a person plays. To determine the sport played by Kelly Miller, who is 5 feet 10 inches (70 inches) and weighs 170 we would call

```
classifier.classify([70, 170])
```

In my code, `classify` is just a wrapper method for `nearestNeighbor`:

```
def classify(self, itemVector):
  """Return class we think item Vector is in"""
  return(self.nearestNeighbor(self.normalizeVector(itemVector))[1][0])
```

code it

Can you write the nearestNeighbor method? (For my solution, I wrote an additional method, manhattanDistance.)

Yet again, download the template classifyTemplate.py to write and test this method at guidetodatamining.com.

Solution

The implementation of the `nearestNeighbor` methods turns out to be very short.

```
def manhattan(self, vector1, vector2):
   """Computes the Manhattan distance."""
   return sum(map(lambda v1, v2: abs(v1 - v2), vector1, vector2))


def nearestNeighbor(self, itemVector):
   """return nearest neighbor to itemVector"""
   return min([ (self.manhattan(itemVector, item[1]), item)
                for item in self.data])
```

That's it!!!

We have written a nearest neighbor classifier in roughly 200 lines of Python.

In the complete code which you can download from our website, I have included a function, `test`, which takes as arguments a training set file and a test set file and prints out how well the classifier performed. Here is how well the classifier did on our athlete data:

```
>>> test("athletesTrainingSet.txt", "athletesTestSet.txt")
-         Track  Aly Raisman       Gymnastics  62    115
+    Basketball  Crystal Langhorne Basketball  74    190
+    Basketball  Diana Taurasi     Basketball  72    163
<snip>
-         Track  Hannah Whelan     Gymnastics  63    117
+    Gymnastics  Jaycie Phelps     Gymnastics  60    97
80.00% correct
```

As you can see, the classifier was 80% accurate. It performed perfectly on predicting basketball players but made four errors between track and gymnastics.

Irises Data Set

I also tested our simple classifier on the Iris Data Set, arguably the most famous data set used in data mining. It was used by Sir Ronald Fisher back in the 1930s. The Iris Data Set consists of 50 samples for each of three species of Irises (Iris Setosa, Iris Virginica, and Iris Versicolor). The data set includes measurements for two parts of the Iris's flower: the sepal (the green covering of the flower bud) and the petals.

Sir Fisher was a remarkable person. He revolutionized statistics and Richard Dawkins called him "the greatest biologist since Darwin."

All the data sets described in the book are available on the book's website: guidetodatamining.com. This allows you to download the data and experiment with the algorithm. Does normalizing the data improve or worsen the accuracy? Does having more data in the training set improve results? What effect does switching to Euclidean Distance have?

REMEMBER: Any learning that takes place happens in your brain, not mine. The more you interact with the material in the book, the more you will learn.

The Iris data set looks like this (species is what the classifier is trying to predict):

Sepal length	Sepal width	Petal Length	Petal Width	Species
5.1	3.5	1.4	0.2	I.setosa
4.9	3.0	1.4	0.2	I setosa

There were 120 instances in the training set and 30 in the test set (none of the test set instances were in the training set).

How well did our classifier do on the Iris Data Set?

```
>>> test('irisTrainingSet.data', 'irisTestSet.data')

93.33% correct
```

Again, a fairly impressive result considering how simple our classifier is. Interestingly, without normalizing the data the classifier is 100% accurate. We will explore this normalization problem in more detail in a later chapter.

miles per gallon.

Finally, I tested our classifier on a modified version of another widely used data set, the Auto Miles Per Gallon data set from Carnegie Mellon University. It was initially used in the 1983 American Statistical Association Exposition. The format of the data looks like this

mpg	cylinders	c.i.	HP	weight	secs. 0-60	make/model
30	4	68	49	1867	19.5	fiat 128
45	4	90	48	2085	21.7	vw rabbit (diesel)
20	8	307	130	3504	12	chevrolet chevelle malibu

In the modified version of the data, we are trying to predict mpg, which is a discrete category (with values 10, 15, 20, 25, 30, 35, 40, and 45) using the attributes cylinders, displacement, horsepower, weight, and acceleration.

There are 342 instances of cars in the training set and 50 in the test set. If we just predicted the miles per gallon randomly, our accuracy would be 12.5%.

```
>>> test('mpgTrainingSet.txt', 'mpgTestSet.txt')

56.00% correct
```

Without normalization the accuracy is 32%.

How can we improve the accuracy of our predictions?

Will improving the classification algorithm help?

How about increasing the size of our training set?

How about having more attributes.

Tune in to the next chapter to find out!

odds and ends

Heads Up on Normalization

In this chapter we talked the importance of normalizing data. This is critical when attributes have drastically different scales (for example, income and age). In order to get accurate distance measurements, we should rescale the attributes so they all have the same scale.

While most data miners use the term 'normalization' to refer to this rescaling, others make a distinction between 'normaliza-tion' and 'standardization.' For them, normalization means scaling values so they lie on a scale from 0 to 1. Standardization, on the other hand, refers to scaling an attribute so the average (mean or median) is 0, and other values are deviations from this average (standard deviation or absolute standard deviation). So for these data miners, Standard Score and Modified Standard Score are examples of standardization.

Recall that one way to normalize an attribute on a scale between 0 and 1 is to find the minimum (min) and maximum (max) values of that attribute. The normalized value of a value is then

$$\frac{value - \min}{\max - \min}$$

Let's compare the accuracy of a classifer that uses this formula over one that uses the Modified Standard

You say normalize and I say standardize You say tomato and I say tomato

code it

Can you modify our classifier code so that it normalizes the attributes using the formula on our previous page?

You can test its accuracy with our three data sets:

	classifier built		
data set	using no normalization	using the formula on previous page	using Modified Standard Score
Athletes	80.00%	?	80.00%
Iris	100.00%	?	93.33%
MPG	32.00%	?	56.00%

my results

Here are my results:

	classifier built		
data set	using no normalization	using the formula on previous page	using Modified Standard Score
Athletes	80.00%	60.00%	80.00%
Iris	100.00%	83.33%	93.33%
MPG	32.00%	36.00%	56.00%

Hmm. These are disappointing results compared with using Modified Standard Score.

It is fun playing with data sets and trying different methods. I obtained the Iris and MPG data sets from the UCI Machine Learning Repository (archive.ics.uci.edu/ml). I encourage you to go there, download a data set or two, convert the data to match data format, and see how well our classifier does.

Evaluating algorithms and kNN

Let us return to the athlete example from the previous chapter.

In that example we built a classifier which took the height and weight of an athlete as input and classified that input by sport—gymnastics, track, or basketball.

So Marissa Coleman, pictured on the left, is 6 foot 1 and weighs 160 pounds. Our classifier correctly predicts she plays basketball:

```
>>> cl = Classifier('athletesTrainingSet.txt')
>>> cl.classify([73, 160])
'Basketball'
```

and predicts that someone 4 foot 9 and 90 pounds is likely to be a gymnast:

```
>>> cl.classify([59, 90])
'Gymnastics'
```

Once we build a classifier, we might be interested in answering some questions about it such as:

How can we answer these questions?

Training set and test set.

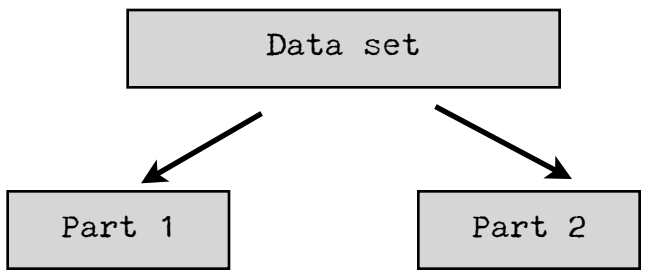
At the end of the previous chapter we worked with three different datasets: the women athlete dataset, the iris dataset, and the auto miles-per-gallon one. We divided each of these datasets in turn into two subsets. One subset we used to construct the classifier. This data set is called the *training set*. The other set was used to evaluate the classifier. That data is called the *test set*. *Training set* and *test set* are common terms in data mining.

People in data mining never test with the data they used to train the system.

You can see why we don't use the training data for testing if we consider the nearest neighbor algorithm. If Marissa Coleman the basketball player from the above example, was in our training data, she at 6 foot 1 and 160 pounds would be the nearest neighbor of herself. So when evaluating a nearest neighbor algorithm, if our test set is a subset of our training data we would always be close to 100% accurate. More generally, in evaluating any data mining algorithm, if our test set is a subset of our training data the results will be optimistic and often overly optimistic. So that doesn't seem like a great idea.

How about the idea we used in the last chapter? We divide our data into two parts. The larger part we use for training and the smaller part we use for evaluation. As it turns out that has its problems too. We could be extremely unlucky in how we divide up our data. For example, all the basketball players in our test set might be short (like Debbie Black who is only 5 foot 3 and weighs 124 pounds) and get classified as marathoners. And all the track people in the test set might be short and lightweight for that sport like Tatyana Petrova (5 foot 3 and 108 pounds) and get classified as gymnasts. With a test set like this, our accuracy will be poor. On the other hand, we could be very lucky in our selection of a test set. Every person in the test set is the prototypical height and weight for their respective sports and our accuracy is near 100%. In either case, the accuracy based on a single test set may not reflect the true accuracy when our classifier is used with new data.

A solution to this problem might be to repeat the process a number of times and average the results. For example, we might divide the data in half. Let's call the parts Part 1 and Part 2:



We can use the data in Part 1 to train our classifier and the data in Part 2 to test it. Then we will repeat the process, this time training with Part 2 and testing with Part 1. Finally we average the results. One problem with this though, is that we are only using 1/2 the data for training during each iteration. But we can fix this by increasing the number of parts. For example, we can have three parts and for each iteration we will train on 2/3 of the data and test on 1/3. So it might look like this

iteration 1	train with parts 1 and 2	test with part 3
iteration 2	train with parts 1 and 3	test with part 2
iteration 3	train with parts 2 and 3	test with part 1

Average the results.

In data mining, the most common number of parts is 10, and this method is called ...

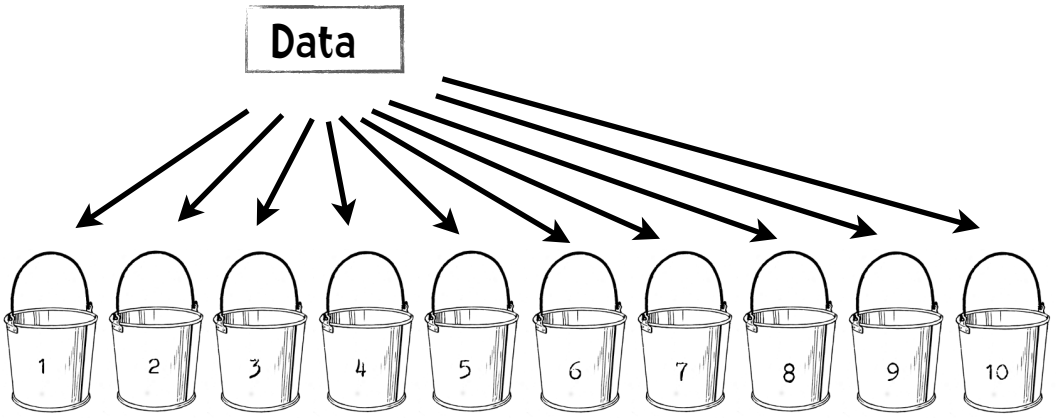
10-fold Cross Validation

With this method we have one data set which we divide randomly into 10 parts. We use 9 of those parts for training and reserve one tenth for testing. We repeat this procedure 10 times each time reserving a different tenth for testing.

Let's look at an example. Suppose I want to build a classifier that just answers yes or no to the question *Is this person a professional basketball player*? My data consists of information about 500 basketball players and 500 non-basketball players.

ten-fold cross validation example:

Step 1, we equally divide the data into 10 buckets:



So we will put 50 basketball players in each bucket and 50 non-players. Each bucket holds information on 100 individuals.

Step 2, we iterate through the following steps ten times:

2.1. During each iteration hold back one of the buckets. For iteration 1, we will hold back bucket 1, iteration 2, bucket 2, and so on.

2.2 We will train the classifier with data from the other buckets. (during the first iteration we will train with the data in buckets 2 through 10).

2.3 We will test the classifier we just built using data from the bucket we held back and save the results. In our case these results might be:

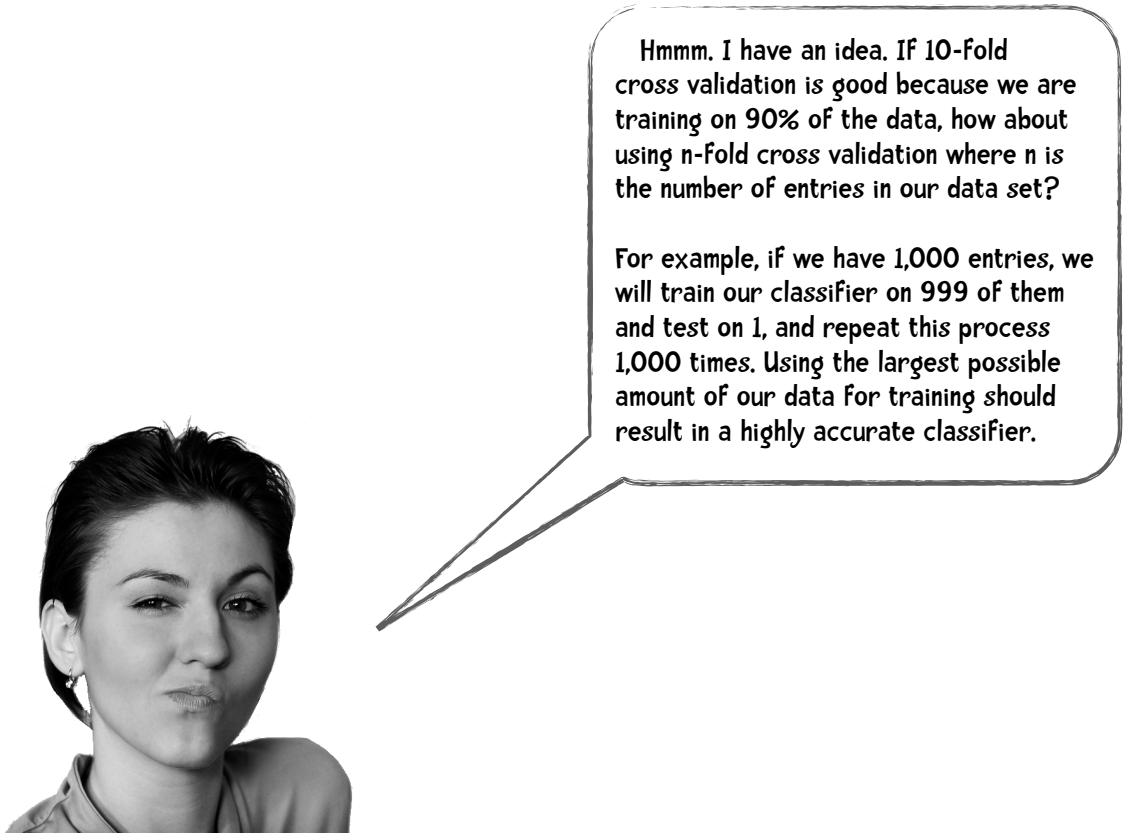
35 of the basketball players were classified correctly
29 of the non basketball players were classified correctly

Step 3, we sum up the results.

Often we will put the final results in a table that looks like this:

	classified as a basketball player	classified as not a basketball player
really a basketball player	372	128
really not a basketball player	220	280

So of the 500 basketball players 372 of them were classified correctly. One thing we could do is add things up and say that of the 1,000 people we classified 652 (372 + 280) of them correctly. So our accuracy is 65.2%. The measures we obtain using ten-fold cross-validation are more likely to be truly representative of the classifiers performance compared with two-fold, or three-fold cross-validation. This is so, because each time we train the classifier we are using 90% of our data compared with using only 50% for two-fold cross-validation.

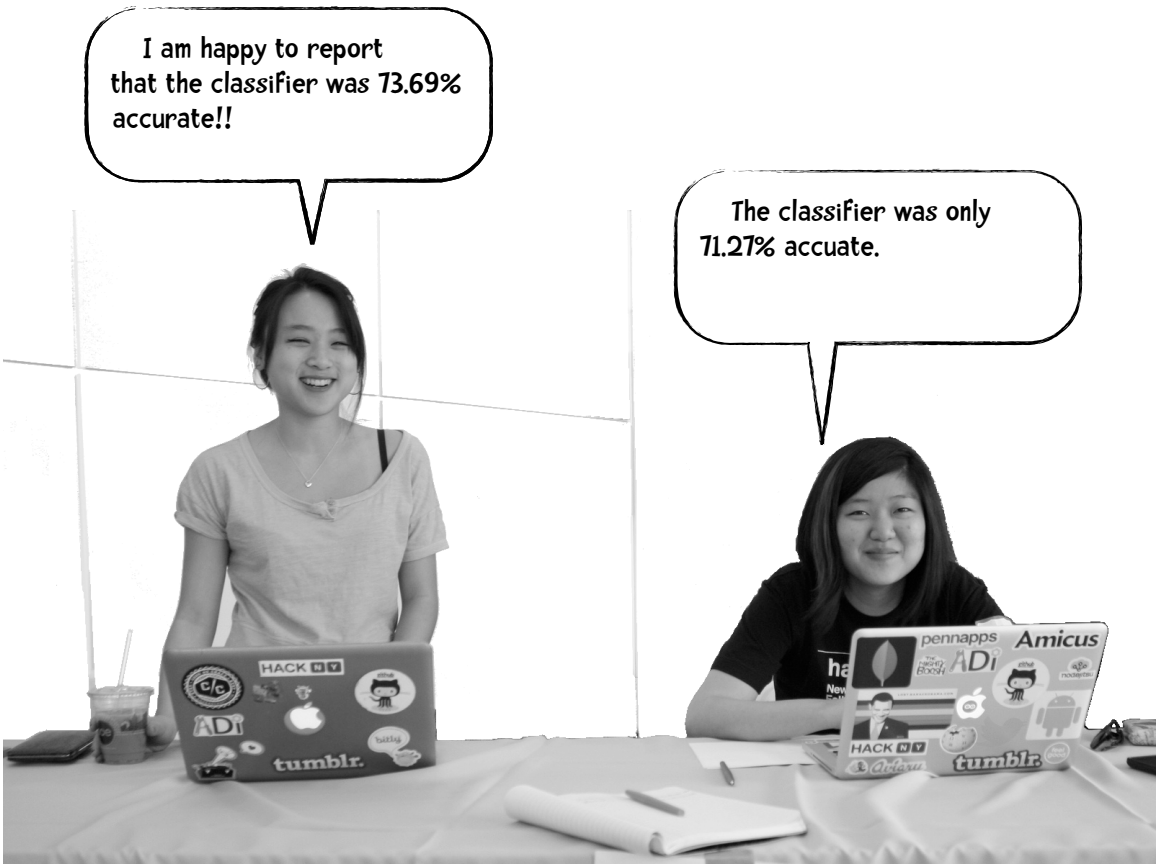


Leave-One-Out

In the machine learning literature, *n*-fold cross validation (where *n* is the number of samples in our data set) is called leave-one-out. We already mentioned one benefit of leave-one-out—at every iteration we are using the largest possible amount of our data for training. The other benefit is that it is deterministic.

What do we mean by 'deterministic'?

Suppose Lucy spends an intense 80 hour week creating and coding a new classifier. It is Friday and she is exhausted so she asks two of her colleagues (Emily and Li) to evaluate the classifier over the weekend. She gives each of them the classifier and the same dataset and asks them to use 10-fold cross validation. On Monday she asks for the results ...



Hmm. They did not get the same results. Did Emily or Li make a mistake? Not necessarily. In 10-fold cross validation we place the data randomly into 10 buckets. Since there is this random element, it is likely that Emily and Li did not divide the data into buckets in exactly the same way. In fact, it is highly unlikely that they did. So when they train the classifier, they are not using exactly the same data and when they test this classifier they are using different test sets. So it is quite logical that they would get different results. This result has nothing to do with the fact that two different people were performing the evaluation. If Lucy herself ran 10-fold cross validation twice, she too would get slightly different results. The reason we get different results is that there is a random component to placing the data into buckets. So 10-fold cross validation is called non-deterministic because when we run the test again we are not guaranteed to get the same result. In contrast, the leave-one-out method is deterministic. Every time we use leave-one-out on the same classifier and the same data we will get the same result. That is a good thing!

The disadvantages of leave-one-out

The main disadvantage of leave-one-out is the computational expense of the method. Consider a modest-sized dataset of 1,000 instances and that it takes one minute to train a classifier. For 10-fold cross validation we will spend 10 minutes in training. In leave-one-out we will spend 16 hours in training. If our dataset contains a million entries the total time spent in training would nearly be two years. Eeeks!

The other disadvantage of leave-one-out is related to stratification.

Stratification.

Let us return to an example from the previous chapter—building a classifier that predicts what sport a woman plays (basketball, gymnastics, or track). When training the classifier we want the training data to be representative and contain data from all three classes. Suppose we assign data to the training set in a completely random way. It is possible that no basketball players would be included in the training set and because of this, the resulting classifier would not be very good at classifying basketball players. Or consider creating a data set of 100 athletes. First we go to the Women's NBA website and write down the info on 33 basketball players; next we go to Wikipedia and get 33 women who competed in gymnastics, at the 2012 Olympics and write that down; finally, we go again to Wikipedia to get information on women who competed in track at the Olympics and record data for 34 people. So our dataset looks like this:

comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76
comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76
comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76

33 women baskball players

33 women gymnasts

34 women marathoners

Let's say we are doing 10-fold cross validation. We start at the beginning of the list and put every ten people in a different bucket. In this case we have 10 basketball players in both the first and second buckets. The third bucket has both basketball players and gymnasts. The fourth and fifth buckets solely contain gymnasts and so on. None of our buckets are representative of the dataset as a whole and you would be correct in thinking this would skew our results. The preferred method of assigning instances to buckets is to make sure that the classes (basketball players, gymnasts, marathoners) are representing in the same proportions as they are in the complete dataset. Since one-third of the complete dataset consists of basketball players, one-third of the entries in each bucket should also be basketball players. And one-third the entries should be gymnasts and one-third marathoners. This is called **stratification** and this is a good thing. The problem with the leave-one-out evaluation method is that necessarily all the test sets are non-stratified since they contain only one instance. In sum, while leave-one-out may be appropriate for very small datasets, 10-fold cross validation is by far the most popular choice.

Confusion Matrices

So far, we have been evaluating our classifier by computing the percent accuracy. That is,

$$\frac{\text{number of test cases correctly classified}}{\text{Total number of test cases}}$$

sometimes we may want a more detailed picture of the performance of our classification algorithm and one such detailed visualization is a table called t*he confusion matrix*. The rows of the confusion matrix represent the actual class of the test cases, the columns represent what our classifier predicted.

The name *confusion matrix* comes from the observation that it is easy for us to see where our algorithm gets confused. Let's look at an example using our women athlete domain. Suppose we have a dataset that consists of attributes for 100 women gymnasts, 100 players in the Women's National Basketball Association, and 100 women marathoners. We evaluate the classifier using 10-fold cross-validation. In 10-fold cross-validation we use each instance of our dataset exactly once for testing. The results of this test might be the following confusion matrix:

	gymnast	basketball player	marathoner
gymnast	83	0	17
basketball player	0	92	8
marathoner	9	16	75

Again, the real class of each instance is represented by the rows; the class predicted by our classifier is represented by the columns. So, for example, 83 instances of gymnasts were classified correctly as gymnasts but 17 were misclassified as marathoners. 92 basketball players were classified correctly as basketball players but 8 were misclassified as marathoners. 75 marathoners were classified correctly but 9 were misclassified as gymnasts and 16 misclassified as basketball players.

The diagonal of the confusion matrix represents instances that were classified correctly.

	gymnast	basketball player	marathoner
gymnast	83	0	17
basketball player	0	92	8
marathoner	9	16	85

In this case the accuracy of the algorithm is:

$$\frac{83+92+75}{300} = \frac{250}{300} = 83.33\%$$

It is easy to inspect the matrix to get an idea of what type of errors our classifier is making. It this example, it seems our algorithm is pretty good at distinguishing between gymnasts and basketball players. Sometimes gymnasts and basketball players get misclassified as marathoners and marathoners occasionally get misclassified as gymnasts or basketball players.

Confusion matrices are not that confusing!

A programming example

Let us go back to a dataset we used in the last chapter, the Auto Miles Per Gallon data set from Carnegie Mellon University. The format of the data looked like:

mpg	cylinders	c.i.	HP	weight	secs. 0-60	make/model
30	4	68	49	1867	19.5	fiat 128
45	4	90	48	2085	21.7	vw rabbit (diesel)
20	8	307	130	3504	12	chevrolet chevelle malibu

I am trying to predict the miles per gallon of a vehicle based on number of cylinders, displacement (cubic inches), horsepower, weight, and acceleration. I put all 392 instances in a file named mpgData.txt and wrote the following short Python program that divided the data into ten buckets using a stratified method. (Both the data file and Python code are available on the website guidetodatamining.com.)

```
import random
def buckets(filename, bucketName, separator, classColumn):
    """the original data is in the file named filename
    bucketName is the prefix for all the bucket names
    separator is the character that divides the columns
    (for ex., a tab or comma) and classColumn is the column
    that indicates the class"""

    # put the data in 10 buckets
    numberOfBuckets = 10
    data = {}
    # first read in the data and divide by category
    with open(filename) as f:
        lines = f.readlines()
    for line in lines:
        if separator != '\t':
            line = line.replace(separator, '\t')
        # first get the category
        category = line.split()[classColumn]
        data.setdefault(category, [])
        data[category].append(line)
    # initialize the buckets
    buckets = []
    for i in range(numberOfBuckets):
        buckets.append([])
    # now for each category put the data into the buckets
    for k in data.keys():
        #randomize order of instances for each class
        random.shuffle(data[k])
        bNum = 0
        # divide into buckets
        for item in data[k]:
            buckets[bNum].append(item)
            bNum = (bNum + 1) % numberOfBuckets
    # write to file
    for bNum in range(numberOfBuckets):
        f = open("%s-%02i" % (bucketName, bNum + 1), 'w')
        for item in buckets[bNum]:
            f.write(item)
        f.close()

buckets("mpgData.txt", 'mpgData','\t',0)
```

Executing this code will produce ten files labelled *mpgData01*, *mpgData02*, etc.

code it

Can you revise the nearest neighbor code from the last chapter so the function test performs 10-fold cross validation on the 10 data files we just created (you can download them at guidetodatamining.com)?

Your program should output a confusion matrix that looks something like:

		predicted MPG							
		10	15	20	25	30	35	40	45
actual MPG	10	3	10	0	0	0	0	0	0
	15	3	68	14	1	0	0	0	0
	20	0	14	66	9	5	1	1	0
	25	0	1	14	35	21	6	1	1
	30	0	1	3	17	21	14	5	2
	35	0	0	2	8	9	14	4	1
	40	0	0	1	0	5	5	0	0
	45	0	0	0	2	1	1	0	2

```
53.316% accurate
total of 392 instances
```

code it - one solution

One solution involves only

- Changing the initializer method to read in data from 9 buckets.
- Adding a new method to test from data in one bucket
- Adding a new procedure that performs 10-fold cross-validation

Let us look at these in turn.

initializer method ___init___

The signature of the init method looks like:

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
```

The filenames of the buckets will be something like mpgData-01, mpgData-02, etc. In this case, `bucketPrefix` will be "mpgData". `testBucketNumber` is the bucket containing the test data. If `testBucketNumber` is 3, the classifier will be trained on buckets 1, 2, 4, 5, 6, 7, 8, 9, and 10. `dataFormat` is a string specifying how to interpret the columns in the data. For example,

```
"class     num    num    num    num    num    comment"
```

specifies that the first column represents the class of the instance. The next 5 columns represent numerical attributes of the instance and the final column should be interpreted as a comment.

The complete, new initializer method is as follows:

```
import copy

class Classifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):

        """ a classifier will be built from files with the bucketPrefix
        excluding the file with textBucketNumber. dataFormat is a
        string that describes how to interpret each line of the data
        files. For example, for the mpg data the format is:
        "class    num   num   num   num   num   comment"
        """
        self.medianAndDeviation = []

        # reading the data in from the file
        self.format = dataFormat.strip().split('\t')
        self.data = []
        # for each of the buckets numbered 1 through 10:
        for i in range(1, 11):
            # if it is not the bucket we should ignore, read the data
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            vector.append(float(fields[i]))
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            classification = fields[i]
                    self.data.append((classification, vector, ignore))
        self.rawData = copy.deepcopy(self.data)
        # get length of instance vector
        self.vlen = len(self.data[0][1])
        # now normalize the data
        for i in range(self.vlen):
            self.normalizeColumn(i)
```

testBucket method

Next, we write a new method that will test the data in one bucket:

```
    def testBucket(self, bucketPrefix, bucketNumber):
        """Evaluate the classifier with data from the file
        bucketPrefix-bucketNumber"""

        filename = "%s-%02i" % (bucketPrefix, bucketNumber)
        f = open(filename)
        lines = f.readlines()
        totals = {}
        f.close()
        for line in lines:
            data = line.strip().split('\t')
            vector = []
            classInColumn = -1
            for i in range(len(self.format)):
                  if self.format[i] == 'num':
                      vector.append(float(data[i]))
                  elif self.format[i] == 'class':
                      classInColumn = i
            theRealClass = data[classInColumn]
            classifiedAs = self.classify(vector)
            totals.setdefault(theRealClass, {})
            totals[theRealClass].setdefault(classifiedAs, 0)
            totals[theRealClass][classifiedAs] += 1
        return totals
```

This takes as input a bucketPrefix and a bucketNumber. If the prefix is "mpgData " and the number is 3, the test data will be read from the file `mpgData-03`. `testBucket` will return a dictionary in the following format:

```
{'35':   {'35': 1, '20': 1, '30': 1},
 '40':   {'30': 1},
 '30':   {'35': 3, '30': 1, '45': 1, '25': 1},
 '15':   {'20': 3, '15': 4, '10': 1},
 '10':   {'15': 1},
 '20':   {'15': 2, '20': 4, '30': 2, '25': 1},
 '25':   {'30': 5, '25': 3}}
```