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.

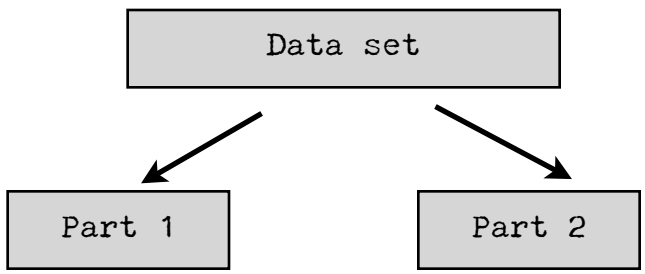
At the end of the previous chapter we worked with three different datasets: the women athlete dataset, the iris dataset, and the auto miles-per-gallon one. We divided each of these datasets in turn into two subsets. One subset we used to construct the classifier. This data set is called the *training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

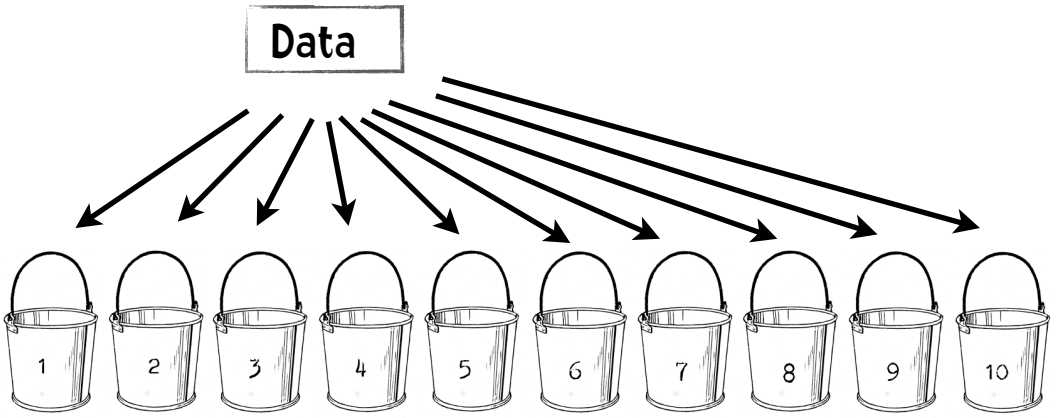
10-fold Cross Validation

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:

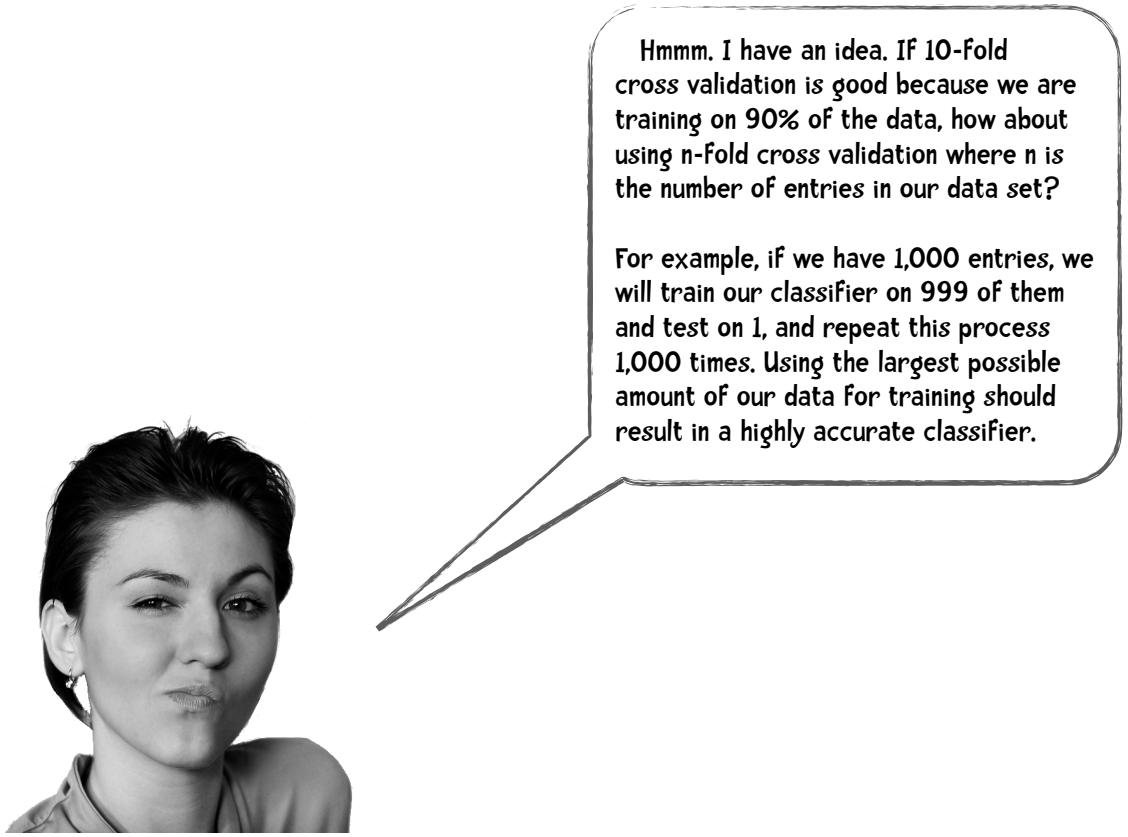
35 of the basketball players were classified correctly
29 of the non basketball players were classified correctly

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.

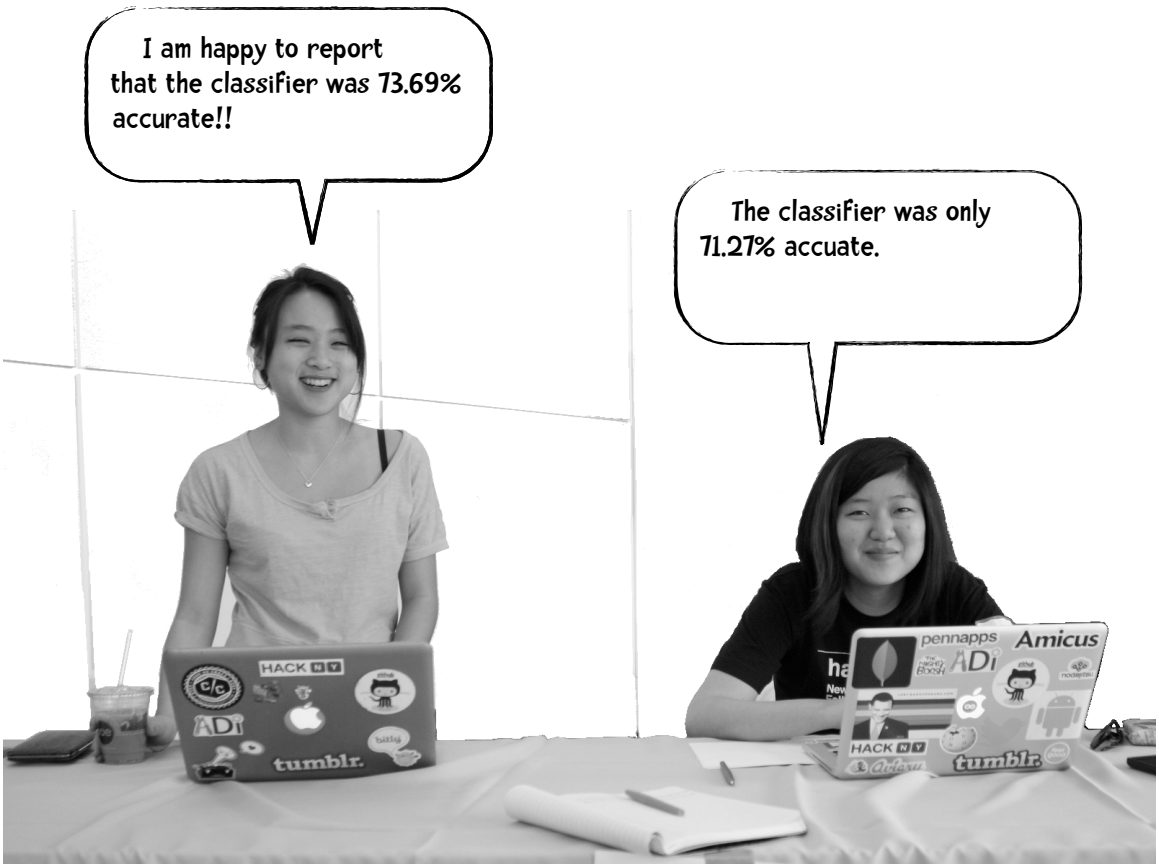


Leave-One-Out

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}}
```

The key of this dictionary represents the true class of the instances. For example, the first line represents results for instances whose true classification is 35 mpg. The value for each key is another dictionary that represents how our classifier classified the instances. For example, the line

```
    '15':   {'20': 3, '15': 4, '10': 1},
```

represents a test where 3 of the instances that were really 15mpg were misclassified as 20mpg, 4 were classified correctly as 15mpg, and 1 was classified incorrectly as 10mpg.

procedure to perform 10-fold cross-validation.

Finally, we need to write a procedure that will perform 10-fold cross-validation. That is, it builds 10 classifiers. Each classifier is trained on 9 of the buckets and tested on data from the remaining bucket.

```
def tenfold(bucketPrefix, dataFormat):
    results = {}
    for i in range(1, 11):
        c = Classifier(bucketPrefix, i, dataFormat)
        t = c.testBucket(bucketPrefix, i)
        for (key, value) in t.items():
            results.setdefault(key, {})
            for (ckey, cvalue) in value.items():
                results[key].setdefault(ckey, 0)
                results[key][ckey] += cvalue

    # now print results
    categories = list(results.keys())
    categories.sort()
    print(   "\n       Classified as: ")
    header =    "        "
    subheader = "      +"
    for category in categories:
        header += category + "   "
        subheader += "----+"
    print (header)
    print (subheader)
    total = 0.0
    correct = 0.0
```

```
    for category in categories:
        row = category + "    |"
        for c2 in categories:
            if c2 in results[category]:
                count = results[category][c2]
            else:
                count = 0
            row += " %2i |" % count
            total += count
            if c2 == category:
                correct += count
        print(row)
    print(subheader)
    print("\n%5.3f percent correct" %((correct * 100) / total))
    print("total of %i instances" % total)

tenfold("mpgData", "class     num    num    num    num    num    comment")
```

Running the program yields the following results:

```
       Classified as:
        10   15   20   25   30   35   40   45
      +----+----+----+----+----+----+----+----+
10    |  5 |  8 |  0 |  0 |  0 |  0 |  0 |  0 |
15    |  8 | 63 | 14 |  1 |  0 |  0 |  0 |  0 |
20    |  0 | 14 | 67 |  8 |  5 |  1 |  1 |  0 |
25    |  0 |  1 | 13 | 35 | 22 |  6 |  1 |  1 |
30    |  0 |  1 |  3 | 17 | 21 | 14 |  5 |  2 |
35    |  0 |  0 |  2 |  7 | 10 | 13 |  5 |  1 |
40    |  0 |  0 |  1 |  0 |  5 |  5 |  0 |  0 |
45    |  0 |  0 |  0 |  2 |  1 |  1 |  0 |  2 |
      +----+----+----+----+----+----+----+----+

52.551 percent correct
total of 392 instances
```

Kappa Statistic!

At the start of this chapter we mentioned some of the questions we might be interested in answering about a classifier including *How good is this classifier*. We also have been refining our evaluation methods and looked at 10-fold cross-validation and confusion matrices. In the example on the previous pages we determined that our classifier for predicted miles per gallon of selected car models was 53.316% accurate. But does 53.316% mean our classifier is good or not so good? To answer that question we are going to look at one more statistics, the Kappa Statistic.

The Kappa Statistic compares the performance of a classifier to that of a classifier that makes predictions based solely on chance. To show you how this works I will start with a simpler example than the mpg one and again return to the women athlete domain. Here are the results of a classifier in that domain:

	gymnast	basketball player	marathoner	TOTALS
gymnast	35	5	20	60
basketball player	0	88	12	100
marathoner	5	7	28	40
TOTALS	40	100	60	200

I also show the totals for the rows and columns. To determine the accuracy we sum the numbers on the diagonal (35 + 88 + 28 = 151) and divide by the total number of instances (200) to get 151 / 200 = .755

Now I am going to generate another confusion matrix that will represent the results of a random classifier (a classifier that makes random predictions). First, we are going to make a copy of the above table only containing the totals:

	gymnast	basketball player	marathoner	TOTALS
gymnast				60
basketball player				100
marathoner				40
TOTALS	40	100	60	200

Looking at the bottom row, we see that 50% of the time (100 instances out of 200) our classifier classifies an instance as “Basketball Player”, 20% of the time (40 instances out of 200) it classifies an instance as “gymnast” and 30% as “marathoner.”

Classifier:

gymnast: 20%
basketball player: 50%
marathoner: 30%

We are going to use these percentages to fill in the rest of the table. There were 60 total real gymnasts. Our random classifier will classify 20% of those as gymnasts. 20% of 60 is 12 so we put a 12 in the table. It will classify 50% as basketball players (or 30 of them) and 30% as marathoners.

	gymnast	basketball player	marathoner	TOTALS
gymnast	12	30	18	60
basketball player				100
marathoner				40
TOTALS	40	100	60	200

And we will continue in this way. There are 100 real basketball players. The random classifier will classify 20% of them (or 20) as gymnasts, 50% as basketball players and 30% as marathoners. And so on:

	gymnast	basketball player	marathoner	TOTALS
gymnast	12	30	18	60
basketball player	20	50	30	100
marathoner	8	20	12	40
TOTALS	40	100	60	200

To determine the accuracy of the random method we sum the numbers on the diagonal and divide by the total number of instances:

$$P(r) = \frac{12 + 50 + 12}{200} = \frac{74}{200} = .37$$

The Kappa Statistic will tell us how much better the real classifier is compared to this random one. The formula is

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)}$$

where P(c) is the accuracy of the real classifier and P(r) is the accuracy of the random one. In this case the accuracy of the real classifier was .755 and that of the random one was .37 so

$$\kappa = \frac{.755 - .37}{1 - .37} = \frac{.385}{.63} = .61$$

How do we interpret that .61? Does that mean our classifier is poor, good, or great? Here is a chart that will help us interpret that number:

A commonly cited* scale on how to interpret Kappa

< 0:	less than chance performance
0.01-0.20	slightly good
0.21-0.40	fair performance
0.41-0.60	moderate performance
0.61-0.80	substantially good performance
0.81-1.00	near perfect performance

* Landis, JR, Koch, GG. 1977. The measurement of observer agreement for categorical data. Biometrics 33:159-74

sharpen your pencil

Suppose we developed a somewhat silly classifier that predicts the major of current university students based on how well they liked 10 movies. We have a data set of 600 students consisting of computer science (cs) majors, education majors (ed), English majors (eng) and psychology majors (psych). The confusion matrix is shown below. Can you compute the Kappa Statistic and interpret what that statistic means?

	predicted major				
	cs	ed	eng	psych	Total
cs	50	8	15	7	
ed	0	75	12	33	
eng	5	12	123	30	
psych	5	25	30	170	

accuracy = 0.697

solution

How good is our classifier? Can you compute the Kappa Statistic and interpret what that statistic means?

First, we sum all the columns:

	cs	**ed**	**eng**	**psych**	**TOTAL**
SUM	60	120	180	240	600
%	10%	20%	30%	40%	100%

Next, we construct the confusion matrix for the random classifier

predicted major

	cs	ed	eng	psych	Total
cs	8	16	24	32	80
ed	12	24	36	48	120
eng	17	34	51	68	170
psych	23	46	69	92	230
Total	60	120	180	240	600

The accuracy of this random classifier is:
(8 + 24 + 51 + 92) / 600 = (175 / 600) = 0.292

solution continued

So the accuracy of our classifier P(c) is 0.697
and that of the random classifier P(r) is 292

The Kappa Statistic is

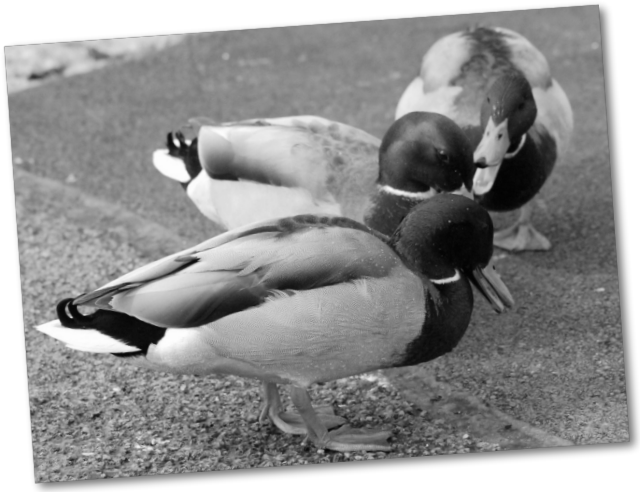
$$\kappa = \frac{P(c) - P(r)}{1 - P(r)}$$

$$\kappa = \frac{0.697 - 0.292}{1 - 0.292} = \frac{0.405}{0.708} = 0.572$$

This suggests our algorithm performs moderately well.

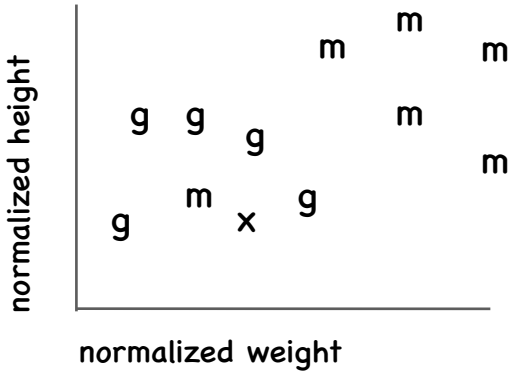
Improvements to the Nearest Neighbor Algorithm!

One trivial example of a classifier is the **Rote Classifier**, which just memorizes the entire training set and only classifies an instance if that instance exactly matches one in the training set. If we only evaluated classifiers on instances in the training data, the Rote Classifier would always be 100% accurate. In real life, the rote classifier is not a good choice because there will be instances we want to classify that are not in the training set. You can view the nearest neighbor algorithm we have been working with as an extension of the rote classifier. Instead of requiring exact matches we are looking at instances that are close matches. Pang-Ning Tan, Michael Steinbach, and Vipin Kumar in their data mining textbook [1] call this the *If it walks like a duck, quacks like a duck, and looks like a duck, then it's probably a duck* approach.



One problem with the nearest neighbor algorithm occurs when we have outliers. Let me explain what I mean by that. And let us return, yet again, to the women athlete domain; this time only looking at gymnasts and marathoners. Suppose we have a particularly short and lightweight marathoner. In diagram form, this data might be represented as on the next page, where *m* indicates 'marathoner' and *g*, 'gymnast.

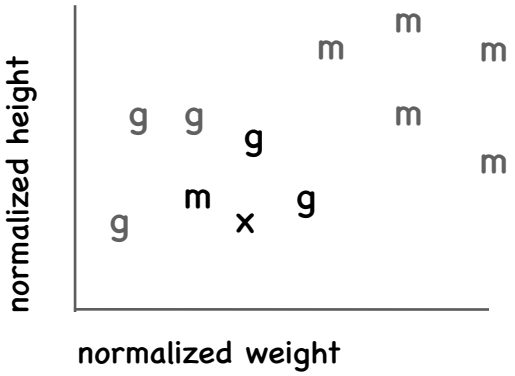
[1] Introduction to Data Mining. 2005. Addison-Wesley



We can see that short lightweight marathoner as the sole *m* in the group of *g*'s. Suppose *x* is an instance we would like to classify. Its nearest neighbor is that outlier *m,* so it would get classified as a marathoner. If we just eyeballed the diagram we would say that *x* is most likely a gymnast since it appears to be in the group of gymnasts.

kNN

One way to improve our current nearest neighbor approach is instead of looking at one nearest neighbor we look at a number of nearest neighbors—k nearest neighbors or kNN. Each neighbor will get a vote and the algorithm will predict that the instance will belong to the class with the highest number of votes. For example, suppose we are using three nearest neighbors ($k = 3$). In that case we have 2 votes for *gymnast* and one for *marathoner,* so we would predict *x* is a gymnast:



So when we are trying to predict a **discrete class** (marathoners, gymnasts, or basketball players, for example) we can use this voting method. The class with the most votes will be the one assigned to the instance. If there is a tie the predicted class will be selected randomly from the classes that are tied. When we are trying to predict a **numeric value** like how many stars a person will give the band *Funky Meters* we can apportion influence from the nearest neighbors to compute a distance-weighted value. Let me parse that out a bit more. Suppose we are trying to predict how well Ben will like *Funky Meters* and Ben's three closest neighbors are Sally, Tara, and Jade. Here are their distances from Ben and their ratings for *Funky Meters*.

User	Distance	Rating
Sally	5	4
Tara	10	5
Jade	15	5

So Sally was closest to Ben and she gave *Funky Meters* a 4. Because I want the rating of the closest person to be weighed more heavily in the final value than the other neighbors, the first step we will do is to convert the distance measure to make it so that the larger the number the closer that person is. We can do this by computing the inverse of the distance (that is, 1 over the distance). So the inverse of Sally's distance of 5 is

$$\frac{1}{5} = 0.2$$

User	Inverse Distance	Rating
Sally	0.2	4
Tara	0.1	5
Jade	0.067	5

Now I am going to divide each of those inverse distances by the sum of all the inverse distances. The sum of the inverse distances is 0.2 + 0.1 + 0.067 = 0.367.

User	Influence	Rating
Sally	0.545	4
Tara	0.272	5
Jade	0.183	5

We should notice two things. First, that the sum of the influence values totals 1. The second thing to notice is that with the original distance numbers Sally was twice as close to Ben as Tara was, and that is preserved in the final numbers were Sally has twice the influence as

Tara does. Finally we are going to multiple each person's influence and rating and sum the results:

predicted Score for Ben

$$= 0.545 \times 4 + 0.272 \times 5 + 0.183 \times 5$$

$$= 2.18 + 1.36 + 0.915 = 4.455$$

sharpen your pencil

I am wondering how well Sofia will like the jazz pianist Hiromi. What is the predicted value given the following data using the k nearest neighbor algorithm with k = 3.?

person	distance from Sofia	rating for Hiromi
Gabriela	4	3
Ethan	8	3
Jayden	10	5

sharpen your pencil - solution

the first thing to do is to compute the inverse (1 over the distance) of each distance:

Person	Inverse Distance	Rating
Gabriela	1/4 = 0.25	3
Ethan	1/8 = 0.125	3
Jayden	1/10 = 0.1	5

The sum of the inverse distances is 0.475. Next I am going to compute the influence of each person by dividing the inverse distance by the sum of each distance

Person	Influence	Rating
Gabriela	0.526	3
Ethan	0.263	3
Jayden	0.211	5

Finally, I multiply the influence by the rating and sum the results:

$$= (0.526 \times 3) + (0.263 \times 3) + (0.211 \times 5)$$

$$= 1.578 + 0.789 + 1.055 = 3.422$$

A new dataset and a challenge!

It is time to look at a new dataset, the Pima Indians Diabetes Data Set developed by the United States National Institute of Diabetes and Digestive and Kidney Diseases.

Astonishingly, over 30% of Pima people develop diabetes. In contrast, the diabetes rate in the United States is 8.3% and in China it is 4.2%.

Each instance in the dataset represents information about a Pima woman over the age of 21 and belonged to one of two classes: a person who developed diabetes within five years, or a person that did not. There are eight attributes:

attributes:

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

Here is an example of the data (the last column represents the class—0=no diabetes; 1=diabetes):

2	99	52	15	94	24.6	0.637	21	0
3	83	58	31	18	34.3	0.336	25	0
5	139	80	35	160	31.6	0.361	25	1
3	170	64	37	225	34.5	0.356	30	1

So, for example, the first woman has had 2 children, has a plasma glucose concentration of 99, a diastolic blood pressure of 52 and so on.

code it - part 1

There are two files on our website. pimaSmall.zip is a zip file containing 100 instances of the data divided into 10 files (buckets). pima.zip is a zip file containing 393 instances. When I used the pimaSmall data with the nearest neighbor classifier we built in the previous chapter using 10-fold cross-validation I got these results:

```
       Classified as:
         0   1
       +----+----+
0      | 45 | 14 |
1      | 27 | 14 |
       +----+----+

59.000 percent correct
total of 100 instances
```

Hint: The python function heapq.nsmallest(n, list) will return a list with the n smallest items.

Here is your task:

Download the classifier code from our website and implement the kNN algorithm. Let us change the initializer method of the class to add another argument, k:

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

The method signature should look like `def knn(self, itemVector):` It should make use of self.k (remember to set that value in the init method) and return the class (in this Pima Cancer dataset case '0' or '1'). You should also modify the procedure tenfold to pass k to the initializer.

code it - answer

My modification to __init___ was simply:

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k):
        self.k = k
        ...
```

My knn method was

```
def knn(self, itemVector):
    """returns the predicted class of itemVector using k
    Nearest Neighbors"""
    # changed from min to heapq.nsmallest to get the
    # k closest neighbors
    neighbors = heapq.nsmallest(self.k,
                            [(self.manhattan(itemVector, item[1]), item)
                            for item in self.data])
    # each neighbor gets a vote
    results = {}
    for neighbor in neighbors:
        theClass = neighbor[1][0]
        results.setdefault(theClass, 0)
        results[theClass] += 1
    resultList = sorted([(i[1], i[0]) for i in results.items()],
                        reverse=True)
    #get all the classes that have the maximum votes
    maxVotes = resultList[0][0]
    possibleAnswers = [i[1] for i in resultList if i[0] == maxVotes]
    # randomly select one of the classes that received the max votes
    answer = random.choice(possibleAnswers)
    return( answer)
```

My slight modification to tenfold was:

```
def tenfold(bucketPrefix, dataFormat, k):
    results = {}
    for i in range(1, 11):
        c = Classifier(bucketPrefix, i, dataFormat, k)
```

...

You can download this code at guidetodatamining.com. Remember, this is just one way to implement this method, and it is not necessarily the best way.

code it - part 2

Which makes the most difference? Having more data (comparing the results from pimaSmall and pima) or having a better algorithm (comparing k=1 to k=3)?

code it - results!

Here are my accuracy results (k=1 is the nearest neighbor algorithm from the last chapter):

	pimaSmall	pima
k=1	59.00%	71.247%
k=3	61.00%	72.519%

So it seems that roughly tripling the amount of data increases the accuracy much more than improving the algorithm does.

sharpen your pencil

Hmm. 72.519% seems like pretty good accuracy but is it? Compute the Kappa Statistic to find out:

	no diabetes	diabetes
no diabetes	219	44
diabetes	64	66

Performance:

- ☐ slightly good
- ☐ fair
- ☐ moderate
- ☐ substantially good
- ☐ near perfect

sharpen your pencil – answer

	no diabetes	diabetes	TOTAL
no diabetes	219	44	263
diabetes	64	66	130
TOTAL	283	110	393
ratio	0.7201	0.2799	

random (r) classifier:

	no diabetes	diabetes
no diabetes	189.39	73.61
diabetes	93.61	36.39

accuracy

$$p(r) = \frac{189.39 + 36.61}{393}$$
$$= .5745$$

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.72519 - .5745}{1 - .5745} = \frac{.15069}{.4255} = .35415$$

Only fair performance

More data, better algorithms & a broken bus

Several years ago I was at a conference in Mexico City. This conference was a bit unusual in that it alternated between a day of presentations and a day of touring (the Monarch Butterflies, Inca ruins, etc). The days of touring involved riding long distances on a bus and the bus had a tendency to break down. As a result, a bunch of us PhD types spend a good deal of time standing at the side of road talking to one another as the bus was being attended to. These roadside exchanges were the highpoint of the conference for me. One of the people I talked to was a person named Eric Brill. Eric Brill is famous for developing what is called the Brill tagger, which does part-of-speech tagging. Similar to what we have been doing in the last few chapters, the Brill tagger classifies data—in this case, it classifies words by their part of speech (noun, verb, etc.). The algorithm Brill came up with was significantly better than its predecessors (and as a result Brill became famous in natural language processing circles). At the side of that Mexican road, I got to talking with Eric Brill about improving the performance of algorithms. His view is that you get more of an improvement by getting more data for the training set, than you would by improving the algorithm. In fact, he felt that if he kept the original part-of-speech tagging algorithm and just increased the size of the training data, the improvement would exceed that of his famous algorithm. Although, he said, you cannot get a PhD for just collecting more data, but you can for developing an algorithm with marginally improved performance!

Here's another example. In various machine translation competitions, Google always places at the top. Granted that Google has a large number of very bright people developing great algorithms, much of Google's dominance is due to its enormous training sets it acquired from web.

更多数据 ⇨ Més dades ⇨ More data

This isn't to say that you shouldn't pick the best algorithm for the job. As we have already seen picking a good algorithm makes a significant difference. However, if you are trying to solve a practical problem (rather than publish a research paper) it might not be worth your while to spend a lot of time researching and tweaking algorithms. You will perhaps get more bang for your buck—or a better return on your time—if you concentrate on getting more data.

With that nod toward the importance of data, I will continue my path of introducing new algorithms.

People have used kNN classifiers for

recommending items at Amazon
assessing consumer credit risk
classifying land cover using image analysis
recognizing faces
classifying the gender of people in images
recommending web pages
recommending vacation packages