PREV
Chapter 3. Splitting datasets one feature at a time: decision trees

NEXT
Chapter 5. Logistic regression

Chapter 4. Classifying with probability theory: naïve Bayes

This chapter covers

- Using probability distributions for classification
- Learning the naïve Bayes classifier
- Parsing data from RSS feeds
- Using naïve Bayes to reveal regional attitudes

In the first two chapters we asked our classifier to make hard decisions. We asked for a definite answer for the question "Which class does this data instance belong to?" Sometimes the classifier got the answer wrong. We could instead ask the classifier to give us a best guess about the class and assign a probability estimate to that best guess.

Probability theory forms the basis for many machine-learning algorithms, so it's important that you get a good grasp on this topic. We touched on probability a bit in chapter 3 when we were calculating the probability of a feature taking a given value. We calculated the probability by counting the number of times the feature equals that value divided by the total number of instances in the dataset. We're going to expand a little from there in this chapter.

We'll look at some ways probability theory can help us classify things. We start out with the simplest probabilistic classifier and then make a few assumptions and learn the naïve Bayes classifier. It's called naïve because the formulation makes some naïve assumptions. Don't worry; you'll see these in detail in a bit. We'll take full advantage of Python's text-processing abilities to split up a document into a word vector. This will be used to classify text. We'll build another classifier and see how it does on a real-world spam email dataset. We'll review conditional probability in case you need a refresher. Finally, we'll show how you can put what the classifier has learned into human-readable terms from a bunch of personal ad postings.

4.1. Classifying with Bayesian decision theory

Naïve Bayes

Pros: Works with a small amount of data, handles multiple classes

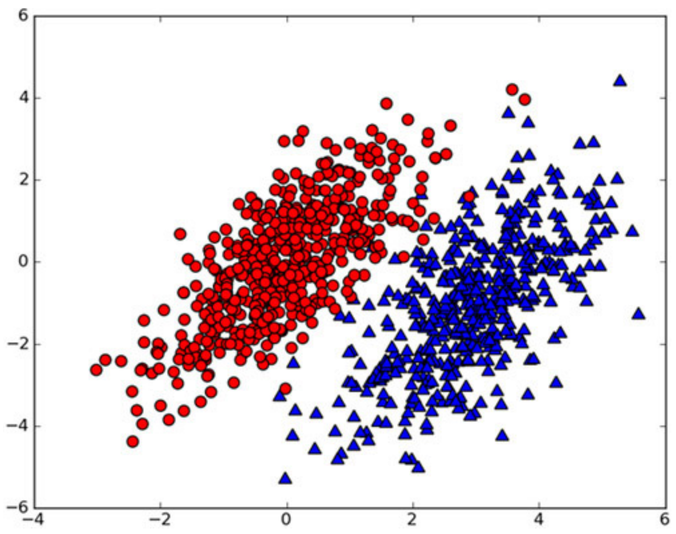
Cons: Sensitive to how the input data is prepared

Works with: Nominal values

Naïve Bayes is a subset of Bayesian decision theory, so we need to talk about Bayesian decision theory quickly before we get to naïve Bayes.

Assume for a moment that we have a dataset with two classes of data inside. A plot of this data is shown in figure 4.1.

Figure 4.1. Two probability distribu-tions with known parameters describing the distribution



We have the data shown in figure 4.1 and we have a friend who read this book; she found the statistical parameters of the two classes of data. (Don't worry about how to find the statistical parameters for this type of data now; we'll get to that in chapter 10.) We have an equation for the probability of a piece of data belonging to Class 1 (the circles): `p1(x, y)`, and we have an equation for the class belonging to Class 2 (the triangles): `p2(x, y)`. To classify a new measurement with features (x, y), we use the following rules:

If `p1(x, y) > p2(x, y)`, then the class is 1.

If `p2(x, y) > p1(x, y)`, then the class is 2.

Put simply, we choose the class with the higher probability. That's Bayesian decision theory in a nutshell: choosing the decision with the highest probability. Let's get back to the data in figure 4.1. If you can represent the data in six floating-point numbers, and the code to calculate the probability is two lines in Python, which would you rather do?

1. Use kNN from chapter 1, and do 1,000 distance calculations.
2. Use decision trees from chapter 2, and make a split of the data once along the x-axis and once along the y-axis.
3. Compute the probability of each class, and compare them

The decision tree wouldn't be very successful, and kNN would require a lot of calculations compared to the simple probability calculation. Given this problem, the best choice would be the probability comparison we just discussed.

We're going to have to expand on the p1 and p1 probability measures I provided here. In order to be able to calculate p1 and p2, we need to discuss conditional probability. If you feel that you have a good handle on conditional probability, you can skip the next section.

Bayes?

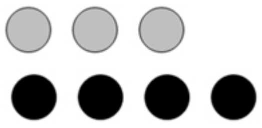
This interpretation of probability that we use belongs to the category called Bayesian probability; it's popular and it works well. Bayesian probability is named after Thomas Bayes, who was an eighteenth-century theologian. Bayesian probability allows prior knowledge and logic to be applied to uncertain statements. There's another interpretation called *frequency probability*, which only draws conclusions from data and doesn't allow for logic and prior knowledge.

4.2. Conditional probability

Let's spend a few minutes talking about probability and conditional probability. If you're comfortable with the `p(x,y|c`$_1$`)` symbol, you may want to skip this section.

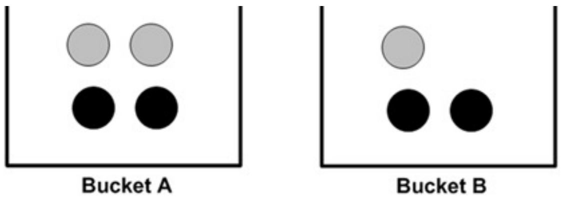
Let's assume for a moment that we have a jar containing seven stones. Three of these stones are gray and four are black, as shown in figure 4.2. If we stick a hand into this jar and randomly pull out a stone, what are the chances that the stone will be gray? There are seven possible stones and three are gray, so the probability is 3/7. What is the probability of grabbing a black stone? It's 4/7. We write the probability of gray as `P(gray)`. We calculated the probability of drawing a gray stone `P(gray)` by counting the number of gray stones and dividing this by the total number of stones.

Figure 4.2. A collection has seven stones that are gray or black. If we randomly select a stone from this set, the probability it will be a gray stone is 3/7. Similarly, the probability of selecting a black stone is 4/7.



What if the seven stones were in two buckets? This is shown in figure 4.3.

Figure 4.3. Seven stones sitting in two buckets



If you want to calculate the `P(gray)` or `P(black)`, would knowing the bucket change the answer? If you wanted to calculate the probability of drawing a gray stone from bucket B, you could probably figure out how do to that. This is known as *conditional probability*. We're calculating the probability of a gray stone, given that the unknown stone comes from bucket B. We can write this as `P(gray|bucketB)`, and this would be read as "the probability of gray given bucket B." It's not hard to see that `P(gray|bucketA)` is 2/4 and `P(gray|bucketB)` is 1/3.

To formalize how to calculate the conditional probability, we can say

```
P(gray|bucketB) = P(gray and bucketB)/P(bucketB)
```

Let's see if that makes sense: `P(gray and bucketB) = 1/7`. This was calculated by taking the number of gray stones in bucket B and dividing by the total number of stones. Now, `P(bucketB)` is 3/7 because there are three stones in bucket B of the total seven stones. Finally, `P(gray|bucketB) = P(gray and bucketB)/P(bucketB) = (1/7) / (3/7) = 1/3`. This formal definition may seem like too much work for this simple example, but it will be useful when we have more features. It's also useful to have this formal definition if we ever need to algebraically manipulate the conditional probability.

Another useful way to manipulate conditional probabilities is known as Bayes' rule. Bayes' rule tells us how to swap the symbols in a conditional probability statement. If we have `P(x|c)` but want to have `P(c|x)`, we can find it with the following:

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}$$

Now that we've discussed conditional probability, we need to see how to apply this to our classifier. The next section will discuss how to use conditional probabilities with Bayesian decision theory.

4.3. Classifying with conditional probabilities

In section 4.1, I said that Bayesian decision theory told us to find the two probabilities:

If `p1(x, y) > p2(x, y)`, then the class is 1.

If `p2(x, y) > p1(x, y)`, then the class is 2.

These two rules don't tell the whole story. I just left them as `p1()` and `p2()` to keep it as simple as possible. What we really need to compare are $p(c_1|x,y)$ and $p(c_2|x,y)$. Let's read these out to emphasize what they mean. Given a point identified as x,y, what is the probability it came from class c_1? What is the probability it came from class c_2?. The problem is that the equation from our friend is $p(x,y|c_1)$, which is not the same. We can use Bayes' rule to switch things around. Bayes' rule is applied to these statements as follows:

$$p(c_i|x,y) = \frac{p(x,y|c_i)p(c_i)}{p(x,y)}$$

With these definitions, we can define the Bayesian classification rule:

If $P(C_1|x, y) > P(C_2|x, y)$, the class is C_1.

If $P(C_1|x, y) < P(C_2|x, y)$, the class is C_2.

Using Bayes' rule, we can calculate this unknown from three known quantities. We'll soon write some code to calculate these probabilities and classify items using Bayes' rule. Now that we've introduced a bit of probability theory, and you've seen how you can build a classifier with it, we're going to put this in action. The next section will introduce a simple yet powerful application of the Bayesian classifier.

4.4. Document classification with naïve Bayes

One important application of machine learning is automatic document classification. In document classification, the whole document such as an individual email is our instance and the features are things in that email. Email is an example that keeps coming up, but you could classify news stories, message board discussions, filings with the government, or any type of text. You can look at the documents by the words used in them and treat the presence or absence of each word as a feature. This would give you as many features as there are words in your vocabulary. Naïve Bayes—an extension of the Bayesian classifier introduced in the last section—is a popular algorithm for the document-classification problem.

Earlier I mentioned that we're going to use individual words as features and look for the presence or absence of each word. How many features is that? Which (human) language are we assuming? It may be more than one language. The estimated total number of words in the English language is over 500,000.[1] To be able to read in English, it's estimated that you need to understand thousands of words.

1 http://hypertextbook.com/facts/2001/JohnnyLing.shtml (http://hypertextbook.com/facts/2001/JohnnyLing.shtml) retrieved October 20, 2010.

General approach to naïve Bayes

1. Collect: Any method. We'll use RSS feeds in this chapter.
2. Prepare: Numeric or Boolean values are needed.
3. Analyze: With many features, plotting features isn't helpful. Looking at histograms is a better idea.
4. Train: Calculate the conditional probabilities of the independent features.
5. Test: Calculate the error rate.
6. Use: One common application of naïve Bayes is document classification. You can use naïve Bayes in any classification setting. It doesn't have to be text.

Let's assume that our vocabulary is 1,000 words long. In order to generate good probability distributions, we need enough data samples. Let's call this N samples. In previous examples in this book, we had 1,000 examples for the dating site, 200 examples per digit in the handwriting recognition, and 24 examples for our decision tree. Having 24 examples was a little bit low, 200 samples was better, and 1,000 samples was great. In the dating example we had three features. Statistics tells us that if we need N samples for one feature, we need N^{10} for 10 features and N^{1000} for our 1,000-feature vocabulary. The number will get very large very quickly.

If we assume independence among the features, then our N^{1000} data points get reduced to 1000*N. By *independence* I mean statistical independence; one feature or word is just as likely by itself as it is next to other words. We're assuming that the word *bacon* is as likely to appear next to *unhealthy* as it is next to *delicious*. We know this assumption isn't true; *bacon* almost always appears near *delicious* but very seldom near *unhealthy*. This is what is meant by *naïve* in the naïve Bayes classifier. The other assumption we make is that every feature is equally important. We know that isn't true either. If we were trying to classify a message board posting as inappropriate, we probably don't need to look at 1,000 words; maybe 10 or 20 will do. Despite the minor flaws of these assumptions, naïve Bayes works well in practice.

At this point you know enough about this topic to get started with some code. If everything doesn't make sense right now, it might help to see this in action. In the next section, we'll start to implement the naïve Bayes classifier in Python. We'll go through everything that's needed to classify text with Python.

4.5. Classifying text with Python

In order to get features from our text, we need to split up the text. But how do we do that? Our features are going to be tokens we get from the text. A *token* is any combination of characters. You can think of tokens as words, but we may use things that aren't words such as URLs, IP addresses, or any string of characters. We'll reduce every piece of text to a vector of tokens where 1 represents the token existing in the document and 0 represents that it isn't present.

To see this in action, let's make a quick filter for an online message board that flags a message as inappropriate if the author uses negative or abusive language. Filtering out this sort of thing is common because abusive postings make people not come back and can hurt an online community. We'll have two categories: abusive and not. We'll use 1 to represent abusive and 0 to represent not abusive.

First, we're going to show how to transform lists of text into a vector of numbers. Next, we'll show how to calculate conditional probabilities from these vectors. Then, we'll create a classifier, and finally, we'll look at some practical considerations for implementing naïve Bayes in Python.

4.5.1. Prepare: making word vectors from text

We're going to start looking at text in the form of word vectors or token vectors, that is, transform a sentence into a vector. We consider all the words in all of our documents and decide what we'll use for a vocabulary or set of words we'll consider. Next, we need to transform each individual document into a vector from our vocabulary. To get started, open your text editor, create a new file called bayes.py, and add the code from the following listing.

Listing 4.1. Word list to vector function

```
def loadDataSet():
    postingList=[['my', 'dog', 'has', 'flea', \
                  'problems', 'help', 'please'],
                 ['maybe', 'not', 'take', 'him', \
                  'to', 'dog', 'park', 'stupid'],
                 ['my', 'dalmation', 'is', 'so', 'cute', \
                   'I', 'love', 'him'],
                 ['stop', 'posting', 'stupid', 'worthless', 'garbage'],
                 ['mr', 'licks', 'ate', 'my', 'steak', 'how',\
                   'to', 'stop', 'him'],
                 ['quit', 'buying', 'worthless', 'dog', 'food', 'stupid']]
    classVec = [0,1,0,1,0,1]    #1 is abusive, 0 not
    return postingList,classVec

def createVocabList(dataSet):                            ❶ Create an
    vocabSet = set([])                                      empty set
    for document in dataSet:
        vocabSet = vocabSet | set(document)                 Create the union
    return list(vocabSet)                                ❷ of two sets

def setOfWords2Vec(vocabList, inputSet):
    returnVec = [0]*len(vocabList)                          Create a vector
    for word in inputSet:                               ❸ of all 0s
        if word in vocabList:
            returnVec[vocabList.index(word)] = 1
        else: print "the word: %s is not in my Vocabulary!" % word
    return returnVec
```

The first function creates some example data to experiment with. The first variable returned from `loadDatSet()` is a tokenized set of documents from a Dalmatian (spotted breed of dog) lovers message board. The text has been broken up into a set of tokens. Punctuation has been removed from this text as well. We'll return to text processing later. The second variable of `loadDatSet()` returns a set of class labels. Here you have two classes, `abusive` and `not abusive`. The text has been labeled by a human and will be used to train a program to automatically detect abusive posts.

Next, the function `createVocabList()` will create a list of all the unique words in all of our documents. To create this unique list you use the Python set data type. You can give a list of items to the set constructor, and it will only return a unique list. First, you create an empty set. ❶ Next, you append the set with a new set from each document. ❷ The | operator is used for union of two sets; recall that this is the bitwise OR operator from C. Bitwise OR and set union also use the same symbols in mathematical notation.

Finally, after you have our vocabulary list, you can use the function `setOfWords2Vec()`, which takes the vocabulary list and a document and outputs a vector of 1s and 0s to represent whether a word from our vocabulary is present or not in the given document. You then create a vector the same length as the vocabulary list and fill it up with 0s. ❸ Next, you go through the words in the document, and if the word is in the vocabulary list, you set its value to 1 in the output vector. If everything goes well, you shouldn't need to test if a word is in `vocabList`, but you may use this later.

Now let's look at these functions in action. Save bayes.py, and enter the following into your Python shell:

```
>>> import bayes
>>> listOPosts,listClasses = bayes.loadDataSet()
>>> myVocabList = bayes.createVocabList(listOPosts)
>>> myVocabList
['cute', 'love', 'help', 'garbage', 'quit', 'I', 'prol
'stop', 'flea', 'dalmation', 'licks', 'food', 'not',
'posting', 'has', 'worthless', 'ate', 'to', 'maybe',
'how', 'stupid', 'so', 'take', 'mr', 'steak', 'my']
```

If you examine this list, you'll see that there are no repeated words. The list is unsorted, and if you want to sort it, you can do that later.

Let's look at the next function `setOfWords2Vec()`:

```
>>> bayes.setOfWords2Vec(myVocabList, listOPosts[0])
[0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 1]
>>> bayes.setOfWords2Vec(myVocabList, listOPosts[3])
[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1
0, 1, 0, 0, 0, 0, 0]
```

This has taken our vocabulary list or list of all the words you'd like to examine and created a feature for each of them. Now when you apply a given document (a posting to the Dalmatian site), it will be transformed into a word vector. Check to see if this makes sense. What's the word at index 2 in `myVocabList`? It should be *help*. This word should be in our first document. Now check to see that it isn't in our fourth document.

4.5.2. Train: calculating probabilities from word vectors

Now that you've seen how to convert from words to numbers, let's see how to calculate the probabilities with these numbers. You know whether a word occurs in a document, and you know what class the document belongs to. Do you remember Bayes' rule from section 3.2? It's rewritten here, but I've changed the x,y to **w**. The bold type means that it's a vector; that is, we have many values, in our case as many values as words in our vocabulary.

$$p(c_i|\mathbf{w}) = \frac{p(\mathbf{w}|c_i)p(c_i)}{p(\mathbf{w})}$$

We're going to use the right side of the formula to get the value on the left. We'll do this for each class and compare the two probabilities. How do we get the stuff on the right? We can calculate $p(c_i)$ by adding up how many times we see class `i` (abusive posts or non-abusive posts) and then dividing by the total number of posts. How can we get $p(\mathbf{w}|c_i)$? This is where our naïve assumption comes in. If we expand **w** into individual features, we could rewrite this as $p(w_0,w_1,w_2..w_N|c_i)$. Our assumption that all the words were independently likely, and something called conditional independence, says we can calculate this probability as $p(w_0|c_i)p(w_1|c_i)p(w_2|c_i)...p(w_N|c_i)$. This makes our calculations a lot easier.

Pseudocode for this function would look like this:

```
Count the number of documents in each class for every training document:
for each class:
if a token appears in the document → increment the count for that token
increment the count for tokens
for each class:
for each token:
divide the token count by the total token count to get conditional probabilities return conditional probabilities for each class
```

The code in the following listing will do these calculations for us. Open your text editor and insert this code into bayes.py. This function uses some functions from NumPy, so make sure you add `from numpy import *` to the top of bayes.py.

Listing 4.2. Naïve Bayes classifier training function

```
def trainNB0(trainMatrix,trainCategory):
    numTrainDocs = len(trainMatrix)
    numWords = len(trainMatrix[0])
    pAbusive = sum(trainCategory)/float(numTrainDocs)
    p0Num = zeros(numWords); p1Num = zeros(numWords)
    p0Denom = 0.0; p1Denom = 0.0
    for i in range(numTrainDocs):
        if trainCategory[i] == 1:
            p1Num += trainMatrix[i]
            p1Denom += sum(trainMatrix[i])
        else:
            p0Num += trainMatrix[i]
            p0Denom += sum(trainMatrix[i])
    p1Vect = p1Num/p1Denom          #change to log()
    p0Vect = p0Num/p0Denom          #change to log()
    return p0Vect,p1Vect,pAbusive
```

1 Initialize probabilities

2 Vector addition

3 Element-wise division

The function in listing 4.2 takes a matrix of documents, `trainMatrix`, and a vector with the class labels for each of the documents, `trainCategory`. The first thing you do is calculate the probability the document is an abusive document (`class=1`). This is `P(1)` from above; because this is a two-class problem, you can get `P(0)` by `1-P(1)`. For more than a two-class problem, you'd need to modify this a little.

You initialize the numerator and denominator for the $p(w_i|c_1)$ and $p(w_i|c_0)$ calculations. 1 Since you have so many *w*s, you're going to use NumPy arrays to calculate these values quickly. The numerator is a NumPy array with the same number of elements as you have words in your vocabulary. In the `for` loop you loop over all the documents in `trainMatrix`, or our training set. Every time a word appears in a document, the count for that word (`p1Num` or `p0Num`) gets incremented, and the total number of words for a document gets summed up over all the documents. 2 You do this for both classes.

Finally, you divide every element by the total number of words for that class. 3 This is done compactly in NumPy by dividing an array by a float. This can't be done with regular Python lists. Try it out to see for yourself. Finally, the two vectors and one probability are returned.

Let's try this out. After you've added the code from listing 4.2 to bayes.py, open your Python shell and enter the following:

```
>>> from numpy import *
>>> reload(bayes)
<module 'bayes' from 'bayes.py'>
>>> listOPosts,listClasses = bayes.loadDataSet()
This loads the data from preloaded values.
>>> myVocabList = bayes.createVocabList(listOPosts)
You've now created a list of all our words in myVocab
>>> trainMat=[]
>>> for postinDoc in listOPosts:
...     trainMat.append(bayes.setOfWords2Vec(myVocabL
...
```

This `for` loop populates the `trainMat` list with word vectors. Now let's get the probabilities of being abusive and the two probability vectors:

```
>>> p0V,p1V,pAb=bayes.trainNB0(trainMat,listClasses)
```

Let's look inside each of these variables:

```
>>> pAb
0.5
This is just the probability of any document being ab
>>> p0V
array([ 0.04166667,  0.04166667,  0.04166667,  0.
                     .
                     .
        0.04166667,  0.        ,  0.04166667,  0.
        0.04166667,  0.125     ])
>>> p1V
array([ 0.         , 0.        ,  0.        ,  0.0526
                              .
                              .

        0.        ,  0.15789474,  0.        ,  0.0526
        0.        ,  0.         ])
```

First, you found the probability that a document was abusive: `pAb`; this is 0.5, which is correct. Next, you found the probabilities of the words from our vocabulary given the document class. Let's see if this makes sense. The first word in our vocabulary is *cute*. This appears once in the `0` class and never in the `1` class. The probabilities are 0.04166667 and 0.0. This makes sense. Let's look for the largest probability. That's 0.15789474 in the `P(1)` array at index 21. If you look at the word in `myVocabList` at index 26, you'll see that it's the word *stupid*. This tells you that the word *stupid* is most indicative of a class `1` (abusive).

Before we can go on to classification with this, we need to address a few flaws in the previous function.

4.5.3. Test: modifying the classifier for real-world conditions

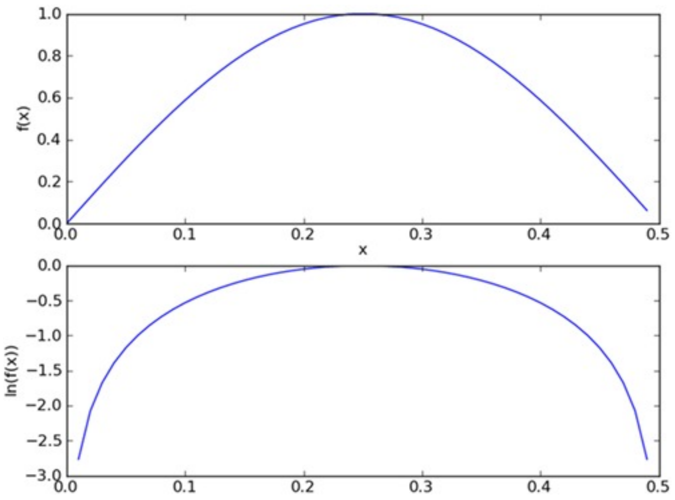
When we attempt to classify a document, we multiply a lot of probabilities together to get the probability that a document belongs to a given class. This will look something like $p(w_0|1)p(w_1|1)p(w_2|1)$. If any of these numbers are 0, then when we multiply them together we get 0. To lessen the impact of this, we'll initialize all of our occurrence counts to 1, and we'll initialize the denominators to 2.

Open bayes.py in your text editor, and change lines 4 and 5 of `trainNB0()` to

```
p0Num = ones(numWords); p1Num = ones(numWords)
p0Denom = 2.0; p1Denom = 2.0
```

Another problem is underflow: doing too many multiplications of small numbers. When we go to calculate the product $p(w_0|c_i)p(w_1|c_i)p(w_2|c_i)...p(w_N|c_i)$ and many of these numbers are very small, we'll get underflow, or an incorrect answer. (Try to multiply many small numbers in Python. Eventually it rounds off to 0.) One solution to this is to take the natural logarithm of this product. If you recall from algebra, `ln(a*b) = ln(a)+ln(b)`. Doing this allows us to avoid the underflow or round-off error problem. Do we lose anything by using the natural log of a number rather than the number itself? The answer is no. Figure 4.4 plots two functions, `f(x)` and `ln(f(x))`. If you examine both of these plots, you'll see that they increase and decrease in the same areas, and they have their peaks in the same areas. Their values are different, but that's fine. To modify our classifier to account for this, modify the last two lines before the return to look like this:

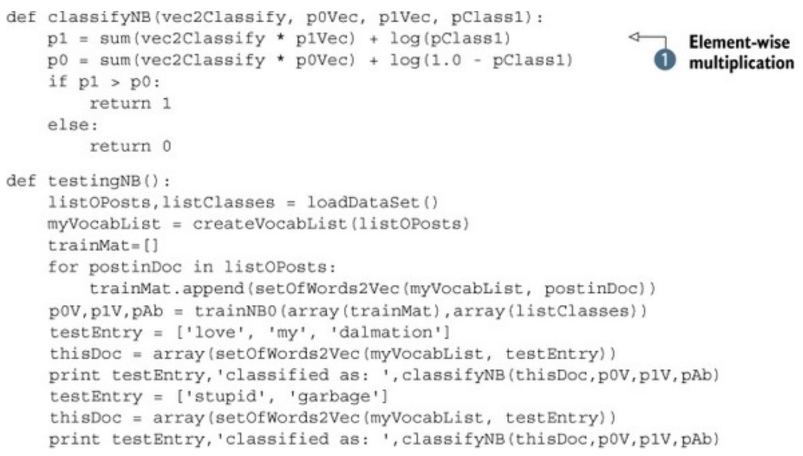
Figure 4.4. Arbitrary functions `f(x)` and `ln(f(x))` increasing together. This shows that the natural log of a function can be used in place of a function when you're interested in finding the maximum value of that function.



```
p1Vect = log(p1Num/p1Denom)
p0Vect = log(p0Num/p0Denom)
```

We're now ready to build the full classifier. It's quite simple when we're using vector math with NumPy. Open your text editor and add the code from the following listing to bayes.py.

Listing 4.3. Naïve Bayes classify function


1 Element-wise multiplication

```
def classifyNB(vec2Classify, p0Vec, p1Vec, pClass1):
    p1 = sum(vec2Classify * p1Vec) + log(pClass1)
    p0 = sum(vec2Classify * p0Vec) + log(1.0 - pClass1)
    if p1 > p0:
        return 1
    else:
        return 0

def testingNB():
    listOPosts,listClasses = loadDataSet()
    myVocabList = createVocabList(listOPosts)
    trainMat=[]
    for postinDoc in listOPosts:
        trainMat.append(setOfWords2Vec(myVocabList, postinDoc))
    p0V,p1V,pAb = trainNB0(array(trainMat),array(listClasses))
    testEntry = ['love', 'my', 'dalmation']
    thisDoc = array(setOfWords2Vec(myVocabList, testEntry))
    print testEntry,'classified as: ',classifyNB(thisDoc,p0V,p1V,pAb)
    testEntry = ['stupid', 'garbage']
    thisDoc = array(setOfWords2Vec(myVocabList, testEntry))
    print testEntry,'classified as: ',classifyNB(thisDoc,p0V,p1V,pAb)
```

The code in listing 4.3 takes four inputs: a vector to classify called vec2Classify and three probabilities calculated in the function `trainNB0()`. You use NumPy arrays to multiply two vectors. 1 The multiplication is element–wise; that is, you multiply the first elements of both vectors, then the second elements, and so on. You next add up the values for all of the words in our vocabulary and add this to the log probability of the class. Finally, you see which probability is greater and return the class label. That isn't too hard, is it?

The second function in listing 4.3 is a convenience function to wrap up everything properly and save you some time from typing all the code from section 4.3.1.

Let's try it out. After you've added the code from listing 4.3, enter the following into your Python shell:

```
>>> reload(bayes)
<module 'bayes' from 'bayes.pyc'>
>>>bayes.testingNB()
['love', 'my', 'dalmation'] classified as:  0
['stupid', 'garbage'] classified as:  1
```

Change the text and see what the classifier spits out. This example is overly simplistic, but it demonstrates how the naïve Bayes classifier works. We'll next make a few changes to it so that it will work even better.

4.5.4. Prepare: the bag-of-words document model

Up until this point we've treated the presence or absence of a word as a feature. This could be described as a set-of-words model. If a word appears more than once in a document, that might convey some sort of information about the document over just the word occurring in the document or not. This approach is known as a bag-of-words model. A *bag* of words can have multiple occurrences of each word, whereas a *set* of words can have only one occurrence of each word. To accommodate for this we need to slightly change the function `setOfWords2Vec()` and call it `bagOfWords2VecMN()`.

The code to use the bag-of-words model is given in the following listing. It's nearly identical to the function `setOfWords2Vec()` listed earlier, except every time it encounters a word, it increments the word vector rather than setting the word vector to 1 for a given index.

Listing 4.4. Naïve Bayes bag-of-words model

```
def bagOfWords2VecMN(vocabList, inputSet):
    returnVec = [0]*len(vocabList)
    for word in inputSet:
        if word in vocabList:
            returnVec[vocabList.index(word)] += 1
    return returnVec
```

Now that we have a classifier built, we should be able to put this into action classifying spam.

4.6. Example: classifying spam email with naïve Bayes

In the previous simple example we imported a list of strings. To use naïve Bayes on some real-life problems we'll need to be able to go from a body of text to a list of strings and then a word vector. In this example we're going to visit the famous use of naïve Bayes: email spam filtering. Let's first look at how we'd approach this problem with our general framework.

Example: using naïve Bayes to classify email

1. Collect: Text files provided.
2. Prepare: Parse text into token vectors.
3. Analyze: Inspect the tokens to make sure parsing was done correctly.
4. Train: Use `trainNB0()` that we created earlier.
5. Test: Use `classifyNB()` and create a new testing function to calculate the error rate over a set of documents.
6. Use: Build a complete program that will classify a group of documents and print misclassified documents to the screen.

First, we'll create some code to parse text into tokens. Next, we'll write a function that ties together the parsing and the classification code from earlier in this chapter. This function will also test the classifier and give us an error rate.

4.6.1. Prepare: tokenizing text

The previous section showed how to create word vectors and use naïve Bayes to classify with these word vectors. The word vectors in the previous section came premade. Let's see how to create your own lists of words from text documents.

If you have a text string, you can split it using the Python string `.split()` method. Let's see this in action. Enter the following into your Python shell:

```
>>> mySent='This book is the best book on Python or M
>>> mySent.split()
['This', 'book', 'is', 'the', 'best', 'book', 'on', '
 'I', 'have', 'ever', 'laid', 'eyes', 'upon.']
```

That works well, but the punctuation is considered part of the word. You can use regular expressions to split up the sentence on anything that isn't a word or number:

```
>>> import re
>>> regEx = re.compile('\\W*')
>>> listOfTokens = regEx.split(mySent)
>>> listOfTokens
['This', 'book', 'is', 'the', 'best', 'book', 'on', '
'L', '', 'I', 'have', 'ever', 'laid', 'eyes', 'upon',
```

Now you have a list of words. But you have some empty strings you need to get rid of. You can count the length of each string and return only the items greater than 0.

```
>>> [tok for tok in listOfTokens if len(tok) > 0]
```

Finally, the first word in the sentence is capitalized. If you were looking at sentences, this would be helpful. You're just looking at a bag of words, so you want all the words to look the same whether they're in the middle, end, or beginning of a sentence. Python has built-in methods for converting strings to all lowercase (`.lower()`) or all uppercase (`.upper()`). This will solve our problem. Let's change our list comprehension to the following:

```
>>> [tok.lower() for tok in listOfTokens if len(tok) 
['this', 'book', 'is', 'the', 'best', 'book', 'on', '
'l', 'i', 'have', 'ever', 'laid', 'eyes', 'upon']
```

Now let's see this in action with a full email from our email dataset. The email dataset is in a folder called email, with two subfolders called spam and ham.

```
>>> emailText = open('email/ham/6.txt').read()
>>> listOfTokens=regEx.split(emailText)
```

The file named 6.txt in the ham folder is quite long. It's from a company telling me that they no longer support something. One thing to notice is that we now have words like *en* and *py* because they were originally part of a URL: /answer.py?hl=en&answer=174623. When we split the URL we got a lot of words. We'd like to get rid of these words, so we'll filter out words with less than three characters. We used one blanket text-parsing rule for this example. In a real-world parsing program, you should have more advanced filters that look for things like HTML and URIs. Right now, a URI will wind up as one of our words; www.whitehouse.gov (http://www.whitehouse.gov) will wind up as three words. Text parsing can be an involved process. We'll create a bare-bones function, and you can modify as you see fit.

4.6.2. Test: cross validation with naïve Bayes

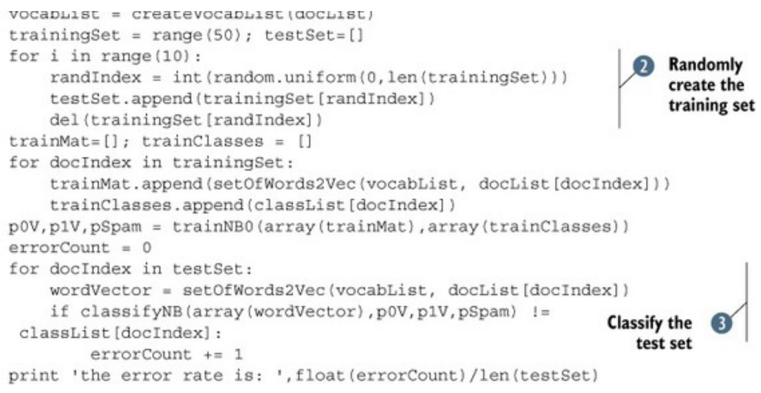
Let's put this text parser to work with a whole classifier. Open your text editor and add the code from this listing to bayes.py.

Listing 4.5. File parsing and full spam test functions

```
def textParse(bigString):
    import re
    listOfTokens = re.split(r'\W*', bigString)
    return [tok.lower() for tok in listOfTokens if len(tok) > 2]

def spamTest():
    docList=[]; classList = []; fullText =[]
    for i in range(1,26):
        wordList = textParse(open('email/spam/%d.txt' % i).read())
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(1)
        wordList = textParse(open('email/ham/%d.txt' % i).read())
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(0)
```

❶ Load and parse text files


```
    vocabList = createVocabList(docList)
    trainingSet = range(50); testSet=[]
    for i in range(10):
        randIndex = int(random.uniform(0,len(trainingSet)))
        testSet.append(trainingSet[randIndex])
        del(trainingSet[randIndex])
    trainMat=[]; trainClasses = []
    for docIndex in trainingSet:
        trainMat.append(setOfWords2Vec(vocabList, docList[docIndex]))
        trainClasses.append(classList[docIndex])
    p0V,p1V,pSpam = trainNB0(array(trainMat),array(trainClasses))
    errorCount = 0
    for docIndex in testSet:
        wordVector = setOfWords2Vec(vocabList, docList[docIndex])
        if classifyNB(array(wordVector),p0V,p1V,pSpam) !=
     classList[docIndex]:
            errorCount += 1
    print 'the error rate is: ',float(errorCount)/len(testSet)
```

The first function, `textParse()`, takes a big string and parses out the text into a list of strings. It eliminates anything under two characters long and converts everything to lowercase. There's a lot more parsing you could do in this function, but it's good enough for our purposes.

The second function, `spamTest()`, automates the naïve Bayes spam classifier. You load the spam and ham text files into word lists. 1 Next, you create a test set and a training set. The emails that go into the test set and the training set will be randomly selected. In this example, we have 50 emails total (not very many). Ten of the emails are randomly selected to be used in the test set. The probabilities will be computed from only the documents in the training set. The Python variable `trainingSet` is a list of integers from 0 to 49. Next, you randomly select 10 of those files. 2 As a number is selected, it's added to the test set and removed from the training set. This randomly selecting a portion of our data for the training set and a portion for the test set is called *hold-out cross validation*. You've done only one iteration, but to get a good estimate of our classifier's true error, you should do this multiple times and take the average error rate.

The next for loop iterates through all the items in the test set and creates word vectors from the words of each email and the vocabulary using `setOfWords2Vec()`. These words are used in `traindNB0()` to calculate the probabilities needed for classification. You then iterate through the test set and classify each email in the test set. 3 If the email isn't classified correctly, the error count is incremented, and finally the total percentage error is reported.

Give this a try. After you've entered the code from listing 4.5, enter the following into your Python shell:

```
>>> bayes.spamTest()
the error rate is:  0.0
>>> bayes.spamTest()
classification error ['home', 'based', 'business', 'o
'knocking', 'your', 'door', 'don', 'rude', 'and', 'le
'you', 'can', 'earn', 'great', 'income', 'and', 'find
'financial', 'life', 'transformed', 'learn', 'more',
'success', 'work', 'from', 'home', 'finder', 'experts
the error rate is:  0.1
```

The function `spamTest()` displays the error rate from 10 randomly selected emails. Since these are randomly selected, the results may be different each time. If there's an error, it will display the word list for that document to give you an idea of what was misclassified. To get a good estimate of the error rate, you should repeat this procedure multiple times, say 10, and average the results. I did that and got an average error rate of 6%.

The error that keeps appearing is a piece of spam that was misclassified as ham. It's better that a piece of spam sneaks through the filter than a valid email getting shoved into the spam folder. There are ways to bias the classifier to not make these errors, and we'll talk about these in chapter 7.

Now that we've used naïve Bayes to classify documents, we're going to look at another use for it. The next example will show how to interpret the knowledge acquired from training the naïve Bayes classifier.

4.7. Example: using naïve Bayes to reveal local attitudes from personal ads

Our next and final example is a fun one. We looked at two practical applications of the naïve Bayes classifier. The first one was to filter out malicious posts on a website, and the second was to filter out spam in email. There are a number of other uses for classification. I've seen someone take the naïve Bayes classifier and train it with social network profiles of women he liked and women he didn't like and then use the classifier to test how he would like an unknown person. The range of possibilities is limited only by your imagination. It's been shown that the older someone is, the better their vocabulary becomes. Could we guess a person's age by the words they use? Could we guess other factors about the person? Advertisers would love to know specific demographics about a person to better target the products they promote. Where would you get such training material? The internet abounds with training material. Almost every imaginable niche has a dedicated community where people have identified themselves as belonging to that community. The Dalmatian owners' site used in section 4.3.1 is a great example.

In this last example, we'll take some data from personals ads from multiple people for two different cities in the United States. We're going to see if people in different cities use different words. If they do, what are the words they use? Can the words people use give us some idea what's important to people in different cities?

Example: using naïve Bayes to find locally used words

1. Collect: Collect from RSS feeds. We'll need to build an interface to the RSS feeds.
2. Prepare: Parse text into token vectors.
3. Analyze: Inspect the tokens to make sure parsing was done correctly.
4. Train: Use `trainNB0()` that we created earlier.
5. Test: We'll look at the error rate to make sure this is actually working. We can make modifications to the tokenizer to improve the error rate and results.
6. Use: We'll build a complete program to wrap everything together. It will display the most common words given in two RSS feeds.

We're going to use the city that each ad comes from to train a classifier and then see how well it does. Finally, we're not going to use this to classify anything. We're going to look at the words and conditional probability scores to see if we can learn anything specific to one city over another.

4.7.1. Collect: importing RSS feeds

The first thing we're going to need to do is use Python to download the text. Luckily, the text is readily available in RSS form. Now all we need is an RSS reader. Universal Feed Parser is the most common RSS library for Python.

You can view documentation here: http://code.google.com/p/feedparser/ (http://code.google.com/p/feedparser/). You should be able to install it like other Python packages, by unzipping the downloaded package, changing your directory to the unzipped package, and then typing `>>python setup.py install` at the command prompt.

We're going to use the personal ads from Craigslist, and hopefully we'll stay Terms Of Service compliant. To open the RSS feed from Craigslist, enter the following at your Python shell:

```
>>> import feedparser
>>>ny=feedparser.parse('http://newyork.craigslist.org
```

I've decided to use the step, or strictly platonic, section from Craigslist because other sections can get a little lewd. You can play around with the feed and check out the great documentation at feedparser.org. To access a list of all the entries type

```
>>> ny['entries']
>>> len(ny['entries'])
100
```

You can create a function similar to `spamTest()` to automate your testing. Open your text editor and enter the code from the following listing.

Listing 4.6. RSS feed classifier and frequent word removal functions

```
def calcMostFreq(vocabList,fullText):
    import operator
    freqDict = {}
    for token in vocabList:
        freqDict[token]=fullText.count(token)
    sortedFreq = sorted(freqDict.iteritems(), key=operator.itemgetter(1),\
                 reverse=True)
    return sortedFreq[:30]

def localWords(feed1,feed0):
    import feedparser
    docList=[]; classList = []; fullText =[]
    minLen = min(len(feed1['entries']),len(feed0['entries']))
    for i in range(minLen):
        wordList = textParse(feed1['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(1)
        wordList = textParse(feed0['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)

        classList.append(0)
    vocabList = createVocabList(docList)
    top30Words = calcMostFreq(vocabList,fullText)
    for pairW in top30Words:
        if pairW[0] in vocabList: vocabList.remove(pairW[0])
    trainingSet = range(2*minLen); testSet=[]
    for i in range(20):
        randIndex = int(random.uniform(0,len(trainingSet)))
        testSet.append(trainingSet[randIndex])
        del(trainingSet[randIndex])
    trainMat=[]; trainClasses = []
    for docIndex in trainingSet:
        trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
        trainClasses.append(classList[docIndex])
    p0V,p1V,pSpam = trainNB0(array(trainMat),array(trainClasses))
    errorCount = 0
    for docIndex in testSet:
        wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
        if classifyNB(array(wordVector),p0V,p1V,pSpam) != \
            classList[docIndex]:
            errorCount += 1
    print 'the error rate is: ',float(errorCount)/len(testSet)
    return vocabList,p0V,p1V
```

1 Calculates frequency of occurrence

2 Accesses one feed at a time

3 Removes most frequently occurring words

The code in listing 4.6 is similar to the `spamTest()` function in listing 4.5

with some added features. One helper function is included in listing 4.6; the function is called `calcMostFreq()`. ❶ The helper function goes through every word in the vocabulary and counts how many times it appears in the text. The dictionary is then sorted by frequency from highest to lowest, and the top 100 words are returned. You'll see why this is important in a second.

The next function, `localWords()`, takes two feeds as arguments. The feeds should be loaded outside this function. The reason for doing this is that feeds can change over time, and if you want to make some changes to our code to see how it performs, you should have the same input data. Reloading the feeds will give you new data, and you won't be sure whether our code changed or new data changed our results. The function `localWords()` is mostly the same as `spamTest()` from listing 4.5. The differences are that you access feeds ❷ instead of files, and you call `calcMostFreq()` to get the top 100 words and then remove these words. ❸ The rest of the function is similar to `spamTest()`, except the last line returns values that you'll use later.

You can comment out the three lines that removed the most frequently used words and see the performance before and after. ❸ When I did this, I had an error rate of 54% without these lines and 70% with the lines included. An interesting observation is that the top 30 words in these posts make up close to 30% of all the words used. The size of the `vocabList` was ~3000 words when I was testing this. A small percentage of the total words makes up a large portion of the text. The reason for this is that a large percentage of language is redundancy and structural glue. Another common approach is to not just remove the most common words but to also remove this structural glue from a predefined list. This is known as a *stop word* list, and there are a number of sources of this available. (At the time of writing, http://www.ranks.nl/resources/stopwords.html (http://www.ranks.nl/resources/stopwords.html) has a good list of stop words in multiple languages.)

After you've entered the code from listing 4.6 into bayes.py, you can test it in Python by typing in the following:

```
>>> reload(bayes)
<module 'bayes' from 'bayes.py'>
>>>ny=feedparser.parse('http://newyork.craigslist.org
>>>sf=feedparser.parse('http://sfbay.craigslist.org/s
>>> vocabList,pSF,pNY=bayes.localWords(ny,sf)
the error rate is:  0.1
>>> vocabList,pSF,pNY=bayes.localWords(ny,sf)
the error rate is:  0.35
```

To get a good estimate of the error rate, you should do multiple trials of this and take the average. The error rate here is much higher than for the spam testing. That is not a huge problem because we're interested in the word probabilities, not actually classifying anything. You can play around the number of words removed by `caclMostFreq()` and see how the error rate changes.

4.7.2. Analyze: displaying locally used words

You can sort the vectors pSF and pNY and then print out the words from `vocabList` at the same index. There's one last piece of code that does this for you. Open bayes.py one more time and enter the code from the following listing.

Listing 4.7. Most descriptive word display function

```
def getTopWords(ny,sf):
    import operator
    vocabList,p0V,p1V=localWords(ny,sf)
    topNY=[]; topSF=[]
    for i in range(len(p0V)):
        if p0V[i] > -6.0 : topSF.append((vocabList[i]
        if p1V[i] > -6.0 : topNY.append((vocabList[i]
    sortedSF = sorted(topSF, key=lambda pair: pair[1]
    print "SF**SF**SF**SF**SF**SF**SF**SF**SF**SF
    for item in sortedSF:
        print item[0]
    sortedNY = sorted(topNY, key=lambda pair: pair[1]
    print "NY**NY**NY**NY**NY**NY**NY**NY**NY**NY
    for item in sortedNY:
        print item[0]
```

The function `getTopWords()` in listing 4.7 takes the two feeds and first trains and tests the naïve Bayes classifier. The probabilities used are returned. Next, you create two lists and store tuples inside the lists. Rather than just return the top *X* words, you return all words above a certain threshold. The tuples are then sorted by their conditional probabilities.

To see this in action, enter the following in your Python shell after you've saved bayes.py.

```
>>> reload(bayes)
<module 'bayes' from 'bayes.pyc'>
>>> bayes.getTopWords(ny,sf)
the error rate is:  0.2
SF**SF**SF**SF**SF**SF**SF**SF**SF**SF**SF**SF**S
love
time
will
there
hit
send
francisco
female
NY**NY**NY**NY**NY**NY**NY**NY**NY**NY**NY**NY**N
friend
people
will
single
sex
female
night
420
```

```
    relationship
    play
    hope
```

The words from this output are entertaining. One thing to note: a lot of stop words appear in the output. It would be interesting to see how things would change if you removed the fixed stop words. In my experience, the classification error will also go down.

4.8. Summary

Using probabilities can sometimes be more effective than using hard rules for classification. Bayesian probability and Bayes' rule gives us a way to estimate unknown probabilities from known values.

You can reduce the need for a lot of data by assuming conditional independence among the features in your data. The assumption we make is that the probability of one word doesn't depend on any other words in the document. We know this assumption is a little simple. That's why it's known as naïve Bayes. Despite its incorrect assumptions, naïve Bayes is effective at classification.

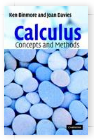
There are a number of practical considerations when implementing naïve Bayes in a modern programming language. Underflow is one problem that can be addressed by using the logarithm of probabilities in your calculations. The bag-of-words model is an improvement on the set-of-words model when approaching document classification. There are a number of other improvements, such as removing stop words, and you can spend a long time optimizing a tokenizer.

The probability theory you learned in this chapter will be used again later in the book, and this chapter was a great introduction to the full power of Bayesian probability theory. We're going to take a break from probability theory. You'll next see a classification method called logistic regression and some optimization algorithms.

PREV
Chapter 3. Splitting datasets one feature at a time: decision trees

NEXT
Chapter 5. Logistic regression

BOOK SECTION



Functions of one variable

from: Calculus: Concepts and Methods by Ken Binmore...

Released: February 2002

59 MINS

Math & Science

BOOK SECTION

Compilation of References

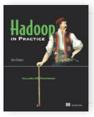
from: Next Generation Content Delivery Infrastructures by Giancarlo Fortino...

Released: June 2012

76 MINS

Enterprise Architecture

BOOK SECTION



Chapter 13. Testing and debugging

from: Hadoop in Practice by Alex Holmes

Released: October 2012

59 MINS

Hadoop

BOOK SECTION

JSL Syntax Rules

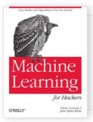
from: JMP 11 Scripting Guide, Second Edition, 2nd Edition by SAS Institute

Released: February 2014

58 MINS

Unix Shell

BOOK SECTION



Optimization: Breaking Codes

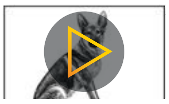
from: Machine Learning for Hackers by Drew Conway...

Released: February 2012

37 MINS

Software Development

VIDEO CLIP



Introduction

from: Hilary Mason: An Introduction to Machine Learning with Web Data by Hilary Mason

Released: May 2011

20 MINS

Software Development

BOOK SECTION

Producing Alpha Channel Video

from: Hands-On Guide to Flash Video by Stefan Richter...

Released: May 2007

21 MINS

Flash