,ch14.14922 Page 219 Thursday, June 25, 2009 2:34 PM

CHAPTER FOURTEEN

Natural Language Corpus Data

Peter Norvig

MOST OF THIS BOOK DEALS WITH DATA THAT IS BEAUTIFUL IN THE SENSE OF BAUDELAIRE: "ALL WHICH IS beautiful and noble is the result of reason and calculation." This chapter's data is beautiful in Thoreau's sense: "All men are really most attracted by the beauty of plain speech." The data we will examine is the plainest of speech: a trillion words of English, taken from publicly available web pages. All the banality of the Web—the spelling and grammatical errors, the LOL cats, the Rickrolling—but also the collected works of Twain, Dickens, Austen, and millions of other authors.

The trillion-word data set was published by Thorsten Brants and Alex Franz of Google in 2006 and is available through the Linguistic Data Consortium (*http://tinyurl.com/ngrams*). The data set summarizes the original texts by *counting* the number of appearances of each word, and of each two-, three-, four-, and five-word sequence. For example, "the" appears 23 billion times (2.2% of the trillion words), making it the most common word. The word "rebating" appears 12,750 times (a millionth of a percent), as does "fnuny" (apparently a misspelling of "funny"). In three-word sequences, "Find all posts" appears 13 million times (.001%), about as often as "each of the," but well below the 100 million of "All Rights Reserved" (.01%). Here's an excerpt from the three-word sequences:

```
outraged many African     63
outraged many Americans  203
outraged many Christians  56
outraged many Irais      58
outraged many Muslims     74
outraged many Pakistanis 124
outraged many Republicans 50
outraged many Turks      390
outraged many by          86
outraged many in         685
outraged many liberal     67
outraged many local       44
outraged many members     61
outraged many of         489
outraged many people     444
outraged many scientists  90
```

We see, for example, that Turks are the most outraged group (on the Web, at the time the data was collected), and that Republicans and liberals are outraged occasionally, but Democrats and conservatives don't make the list.

Why would I say this data is beautiful, and not merely mundane? Each individual count *is* mundane. But the *aggregation* of the counts—billions of counts—is beautiful, because it says so much, not just about the English language, but about the world that speakers inhabit. The data is beautiful because it represents much of what is worth saying.

Before seeing what we can do with the data, we need to talk the talk—learn a little bit of jargon. A collection of text is called a *corpus*. We treat the corpus as a sequence of *tokens*—words and punctuation. Each distinct token is called a *type*, so the text "Run, Lola Run" has four tokens (the comma counts as one) but only three types. The set of all types is called the *vocabulary*. The Google Corpus has a trillion tokens and 13 million types. English has only about a million dictionary words, but the corpus includes types such as "www.njstatelib.org". "+170.002", "1.5GHz/512MB/60GB", and "Abrahamovich". Most of the types are rare, however; the 10 most common types cover almost 1/3 of the tokens, the top 1,000 cover just over 2/3, and the top 100,000 cover 98%.

A 1-token sequence is a *unigram*, a 2-token sequence is a *bigram*, and an *n*-token sequence is an *n-gram*. P stands for *probability*, as in P(*the*) = .022, which means that the probability of the token "the" is .022, or 2.2%. If *W* is a sequence of tokens, then *W*3 is the third token, and *W*1:3 is the sequence of the first through third tokens. *P*(*W*i =*the* | *W*i–1=*of*) is the *conditional probability* of "the", given that "of" is the previous token.

Some details of the Google Corpus: words appearing fewer than 200 times are considered unknown and appear as the symbol `<UNK>`. *N*-grams that occur fewer than 40 times are discarded. This policy lessens the effect of typos and helps keep the data set to a mere 24 gigabytes (compressed). Finally, each sentence in the corpora is taken to start with the special symbol `<S>` and end with `</S>`.

We will now look at some tasks that can be accomplished using the data.

Word Segmentation

Consider the Chinese text 浮法像蝴蝶. This is the translation of the phrase "float like a butterfly." It consists of five characters, but there are no spaces between them, so a Chinese reader must perform the task of *word segmentation*: deciding where the word boundaries are. Readers of English don't normally perform this task, because we have spaces between words. However, some texts, such as URLs, don't have spaces, and sometimes writers make mistakes and leave a space out; how could a search engine or word processing program correct such a mistake?

Consider the English text "choosespain.com." This is a website hoping to convince you to choose Spain as a travel destination, but if you segment the name wrong, you get the less appealing name "chooses pain." Human readers are able to make the right choice by drawing upon years of experience; surely it would be an insurmountable task to encode that experience into a computer algorithm. Yet we can take a shortcut that works surprisingly well: look up each phrase in the bigram table. We see that "choose Spain" has a count of 3,210, whereas "chooses pain" does not appear in the table at all (which means it occurs fewer than 40 times in the trillion-word corpus). Thus "choose Spain" is at least 80 times more likely, and can be safely considered the right segmentation.

Suppose we were faced with the task of interpreting the phrase "insufficientnumbers." If we add together capitalized and lowercase versions of the words, the counts are:

```
insufficient numbers  20751
in sufficient numbers 32378
```

"In sufficient numbers" is 50% more frequent than "insufficient numbers" but that's hardly compelling evidence. We are left in a frustrating position: we can guess, but we can't be confident. In uncertain problems like this, we don't have any way of calculating a definitive correct answer, we don't have a complete model of what makes one answer right, and in fact human experts don't have a complete model, either, and can disagree on the answer. Still, there is an established methodology for solving uncertain problems:

1. **Define a probabilistic model.** We can't define all the factors (semantic, syntactic, lexical, and social) that make "choose Spain" a better candidate for a domain name, but we can define a simplified model that gives approximate probabilities. For short candidates like "choose Spain" we could just look up the *n*-gram in the corpus data and use that as the probability. For longer candidates we will need some way of composing an answer from smaller parts. For words we haven't seen before, we'll have to estimate the probability of an unknown word. The point is that we define a *language model*—a probability distribution over all the strings in the language—and learn the parameters of the model from our corpus data, then use the model to define the probability of each candidate.
2. **Enumerate candidates.** We may not be sure whether "insufficient numbers" or "in sufficient numbers" is more likely to be the intended phrase, but we can agree that they are both candidate segmentations, as is "in suffi cient numb ers," but that "hello

world" is not a valid candidate. In this step we withhold judgment and just enumerate possibilities—all the possibilities if we can, or else a carefully selected sample.

3. **Choose the most probable candidate.** Apply the language model to each candidate to get its probability, and choose the one with the highest probability.

If you prefer mathematical equations, the methodology is:

$$best = \mathrm{argmax}_{c \in \, candidates} \, \mathrm{P}(c)$$

Or, if you prefer computer code (we'll use Python), it would be:

```
best = max(candidates, key=P)
```

Let's apply the methodology to segmentation. We want to define a function, `segment`, which takes as input a string with no spaces and returns a list of words that is the best segmentation:

```
>>> segment('choosespain')
['choose', 'spain']
```

Let's start with step 1, the probabilistic language model. The probability of a sequence of words is the product of the probabilities of each word, given the word's context: all the preceding words. For those who like equations:

$$\mathrm{P}(W_{1:n}) = \Pi_{k=1:n}\mathrm{P}(W_k \mid W_{1:k-1})$$

We don't have the data to compute this exactly, so we can approximate the equation by using a smaller context. Since we have data for sequences up to 5-grams, it would be tempting to use the 5-grams, so that the probability of an *n*-word sequence would be the product of each word given the four previous words (not all previous words).

There are three difficulties with the 5-gram model. First, the 5-gram data is about 30 gigabytes, so it can't all fit in RAM. Second, many 5-gram counts will be 0, and we'd need some strategy for *backing off*, using shorter sequences to estimate the 5-gram probabilities. Third, the search space of candidates will be large because dependencies extend up to four words away. All three of these difficulties can be managed, with some effort. But instead, let's first consider a much simpler language model that solves all three difficulties at once: a unigram model, in which the probability of a sequence is just the product of the probability of each word by itself. In this model, the probability of each word is independent of the other words:

$$\mathrm{P}(W_{1:n}) = \Pi_{k=1:n}\mathrm{P}(W_k)$$

To segment `'wheninrome'`, we consider candidates such as `when in rome`, and compute $\mathrm{P}(when) \times \mathrm{P}(in) \times \mathrm{P}(rome)$. If the product is higher than any other candidate's product, then that's the best answer.

An *n*-character string has 2^{n-1} different segmentations (there are n–1 positions between characters, each of which can either be or not be a word boundary). Thus the string 'wheninthecourseofhumaneventsitbecomesnecessary' has 35 trillion segmentations. But I'm sure you were able to find the right segmentation in just a few seconds; clearly, you couldn't have enumerated all the candidates. You probably scanned "w", "wh", and "whe" and rejected them as improbable words, but accepted "when" as probable. Then you moved on to the remainder and found its best segmentation. Once we make the simplifying assumption that each word is independent of the others, it means that we don't have to consider all combinations of words.

That gives us a sketch of the segment function: consider every possible way to split the text into a first word and a remaining text (we can arbitrarily limit the longest possible word to, say, L=20 letters). For each possible split, find the best way to segment the remainder. Out of all the possible candidates, the one with the highest product of P(*first*)×P(*remaining*) is the best.

Here we show a table of choices for the first word, probability of the word, probability of the best segmentation of the remaining words, and probability of the whole (which is the product of the probabilities of the first and the remainder). We see that the segmentation starting with "when" is 50,000 times better than the second-best candidate.

first	**P(*first*)**	**P(*remaining*)**	**P(*first*) × P(*remaining*)**
w	$2•10^{-4}$	$2•10^{-33}$	$6•10^{-37}$
wh	$5•10^{-6}$	$6•10^{-33}$	$3•10^{-38}$
whe	$3•10^{-7}$	$3•10^{-32}$	$7•10^{-39}$
when	$6•10^{-4}$	$7•10^{-29}$	$4•10^{-32}$
wheni	$1•10^{-16}$	$3•10^{-30}$	$3•10^{-46}$
whenin	$1•10^{-17}$	$8•10^{-27}$	$8•10^{-44}$

We can implement segment in a few lines of Python:

```
@memo
def segment(text):
    "Return a list of words that is the best segmentation of text."
    if not text: return []
    candidates = ([first]+segment(rem) for first,rem in splits(text))
    return max(candidates, key=Pwords)

def splits(text, L=20):
    "Return a list of all possible (first, rem) pairs, len(first)<=L."
    return [(text[:i+1], text[i+1:])
            for i in range(min(len(text), L))]

def Pwords(words):
    "The Naive Bayes probability of a seuence of words."
    return product(Pw(w) for w in words)
```

This is the entire program—with three minor omissions: product is a utility function that multiplies together a list of numbers, memo is a decorator that caches the results of previous calls to a function so that they don't have to be recomputed, and Pw estimates the probability of a word by consulting the unigram count data.

Without memo, a call to segment for an *n*-character text makes 2^n recursive calls to segment; with memo it makes only *n* calls—memo makes this a fairly efficient dynamic programming algorithm. Each of the *n* calls considers O(*L*) splits, and evaluates each split by multiplying O(*n*) probabilities, so the whole algorithm is O(n^2L).

As for Pw, we read in the unigram counts from a datafile. If a word appears in the corpus, its estimated probability is Count(*word*)/N, where *N* is the corpus size. Actually, instead of using the full 13-million-type unigram datafile, I created vocab_common, which (a) is case-insensitive, so that the counts for "the", "The", and "THE" are added together under a single entry for "the"; (b) only has entries for words made out of letters, not numbers or punctuation (so "+170.002" is out, as is "can't"); and (c) lists only the most common 1/3 of a million words (which together cover 98% of the tokens).

The only tricky part of Pw is when a word has not been seen in the corpus. This happens sometimes even with a trillion-word corpus, so it would be a mistake to return 0 for the probability. But what should it be? The number of tokens in the corpus, *N*, is about a trillion, and the least common word in vocab_common has a count of 12,711. So a previously unseen word should have a probability of somewhere between 0 and 12,710/*N*. Not all unseen words are equally unlikely: a random sequence of 20 letters is less likely to be a word than a random sequence of 6 letters. We will define a class for probability distributions, Pdist, which loads a datafile of (key, count) pairs. By default, the probability of an unknown word is 1/*N*, but each instance of a Pdist can supply a custom function to override the default. We want to avoid having too high a probability for very long words, so we (rather arbitrarily) start at a probability of 10/*N*, and decrease by a factor of 10 for every letter in the candidate word. We then define Pw as a Pdist:

```
class Pdist(dict):
    "A probability distribution estimated from counts in datafile."
    def __init__(self, data, N=None, missingfn=None):
        for key,count in data:
            self[key] = self.get(key, 0) + int(count)
        self.N = float(N or sum(self.itervalues()))
        self.missingfn = missingfn or (lambda k, N: 1./N)
    def __call__(self, key):
        if key in self: return self[key]/self.N
        else: return self.missingfn(key, self.N)

def datafile(name, sep='\t'):
    "Read key,value pairs from file."
    for line in file(name):
        yield line.split(sep)

def avoid_long_words(word, N):
    "Estimate the probability of an unknown word."
    return 10./(N * 10**len(word))

N = 1024908267229 ## Number of tokens in corpus

Pw = Pdist(datafile('vocab_common'), N, avoid_long_words))
```

Note that `Pw[w]` is the raw count for word `w`, while `Pw(w)` is the probability. All the programs described in this article are available at *http://norvig.com/ngrams*.

So how well does this model do at segmentation? Here are some examples:

```
>>> segment('choosespain')
['choose', 'spain']
>>> segment('thisisatest')
['this', 'is', 'a', 'test']
>>> segment('wheninthecourseofhumaneventsitbecomesnecessary')
['when', 'in', 'the', 'course', 'of', 'human', 'events', 'it', 'becomes', 'necessary']
>>> segment('whorepresents')
['who', 'represents']
>>> segment('expertsexchange')
['experts', 'exchange']
>>> segment('speedofart')
['speed', 'of', 'art']
>>> segment('nowisthetimeforallgood')
['now', 'is', 'the', 'time', 'for', 'all', 'good']
>>> segment('itisatruthuniversallyacknowledged')
['it', 'is', 'a', 'truth', 'universally', 'acknowledged']
>>> segment('itwasabrightcolddayinaprilandtheclockswerestrikingthirteen')
['it', 'was', 'a', 'bright', 'cold', 'day', 'in', 'april', 'and', 'the', 'clocks',
'were', 'striking', 'thirteen']
>>> segment('itwasthebestoftimesitwastheworstoftimesitwastheageofwisdomitwastheage
offoolishness')
['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times',
'it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of',
'foolishness']
>>> segment('asgregorsamsaawokeonemorningfromuneasydreamshefoundhimselftransformed
inhisbedintoagiganticinsect')
['as', 'gregor', 'samsa', 'awoke', 'one', 'morning', 'from', 'uneasy', 'dreams', 'he',
'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'gigantic',
'insect']
>>> segment('inaholeinthegroundtherelivedahobbitnotanastydirtywetholefilledwiththe
endsofwormsandanoozysmellnoryetadrybaresandyholewithnothinginittositdownonortoeat
itwasahobbitholeandthatmeanscomfort')
['in', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'hobbit', 'not', 'a',
'nasty', 'dirty', 'wet', 'hole', 'filled', 'with', 'the', 'ends', 'of', 'worms', 'and',
'an', 'oozy', 'smell', 'nor', 'yet', 'a', 'dry', 'bare', 'sandy', 'hole', 'with',
'nothing', 'in', 'it', 'to', 'sitdown', 'on', 'or', 'to', 'eat', 'it', 'was', 'a',
'hobbit', 'hole', 'and', 'that', 'means', 'comfort']
>>> segment('faroutintheunchartedbackwatersoftheunfashionableendofthewesternspiral
armofthegalaxyliesasmallunregardedyellowsun')
['far', 'out', 'in', 'the', 'uncharted', 'backwaters', 'of', 'the', 'unfashionable',
'end', 'of', 'the', 'western', 'spiral', 'arm', 'of', 'the', 'galaxy', 'lies', 'a',
'small', 'un', 'regarded', 'yellow', 'sun']
```

The reader might be pleased to see the program correctly segmented such unusual words as “Samsa” and “oozy”. You shouldn’t be surprised: “Samsa” appears 42,000 times and “oozy” 13,000 times in the trillion-word corpus. Overall the results look good, but there are two errors: `'un','regarded'` should be one word, and `'sitdown'` should be two. Still, that’s a word precision rate of 157/159 = 98.7%; not too bad.

The first error is in part because “unregarded” does not appear in our 13-million-word vocabulary. (It *is* in the full 13-million-word vocabulary at position 1,005,493, with count 7,557.) If we put it in the vocabulary, we see that the segmentation is correct:

```
>>> Pw['unregarded'] = 7557
>>> segment('faroutintheunchartedbackwatersoftheunfashionableendofthewesternspiral
armofthegalaxyliesasmallunregardedyellowsun')
['far', 'out', 'in', 'the', 'uncharted', 'backwaters', 'of', 'the', 'unfashionable',
'end', 'of', 'the', 'western', 'spiral', 'arm', 'of', 'the', 'galaxy', 'lies', 'a',
'small', 'unregarded', 'yellow', 'sun']
```

That doesn’t prove we’ve solved the problem: we would have to put back all the other intervening words, not just the one we wanted, and we would have to then rerun all the test cases to make sure that adding the other words did not mess up any other result.

The second error happens because, although “sit” and “down” are common words (with probability .003% and .04%, respectively), the product of their two probabilities is just slightly less than the probability of “sitdown” by itself. However, the probability of the two-word sequence “sit down,” according to the bigram counts, is about 100 times greater. We can try to fix this problem by modeling bigrams; that is, considering the probability of each word, given the previous word:

$$P(W_{1:n}) = \Pi_{k=1:n} P(W_k \mid W_{k-1})$$

Of course the complete bigram table won’t fit into memory. If we keep only bigrams that appear 100,000 or more times, that works out to a little over 250,000 entries, which does fit. We can then estimate P(*down* | *sit*) as Count(*sit down*)/Count(*sit*). If a bigram does not appear in the table, then we just fall back on the unigram value. We can define `cPw`, the conditional probability of a word given the previous word, as:

```
def cPw(word, prev):
    "The conditional probability P(word | previous-word)."
    try:
        return P2w[prev + ' ' + word]/float(Pw[prev])
    except KeyError:
        return Pw(word)

P2w = Pdist(datafile('count2w'), N)
```

(Purists will note `cPw` is not a probability distribution, because the sum over all words for a given previous word can be greater than 1. This approach has the technical name *stupid backoff*, but it works well in practice, so we won’t worry about it.) We can now compare “sitdown” to “sit down” with a preceding “to”:

```
>>> cPw('sit', 'to')*cPw('down', 'sit') / cPw('sitdown', 'to')
1698.0002330199263
```

We see that “sit down” is 1,698 times more likely than “sitdown”, because “sit down” is a popular bigram, and because “to sit” is popular but “to sitdown” is not.

This looks promising; let’s implement a new version of `segment` using a bigram model. While we’re at it, we’ll fix two other issues:

1. When segment added one new word to a sequence of *n* words segmented in the remainder, it called Pwords to multiply together all *n*+1 probabilities. But segment had already multiplied all the probabilities in the remainder. It would be more efficient to remember the probability of the remainder and then just do one more multiplication.
2. There is a potential problem with arithmetic underflow. If we apply Pwords to a sequence consisting of the word "blah" repeated 61 times, we get $5.2 \cdot 10^{-321}$, but if we add one more "blah," we get 0.0. The smallest positive floating-point number that can be represented is about $4.9 \cdot 10^{-324}$; anything smaller than that rounds to 0.0. To avoid underflow, the simplest solution is to add logarithms of numbers rather than multiplying the numbers themselves.

We will define segment2, which differs from segment in three ways: first, it uses a conditional bigram language model, cPw, rather than the unigram model Pw. Second, the function signature is different. Instead of being passed a single argument (the text), segment2 is also passed the previous word. At the start of the sentence, the previous word is the special beginning-of-sentence marker, <S>. The return value is not just a list of words, but rather a pair of values: the probability of the segmentation, followed by the list of words. We return the probability so that it can be stored (by memo) and need not be recomputed; this fixes problem (1), the inefficiency. The function combine takes four inputs—the first word and the remaining words, plus their probabilities—and combines them by appending the first word to the remaining words, and by multiplying the probabilities—except that in order to solve problem (2), we introduce the third difference: we add logarithms of probabilities instead of multiplying the raw probabilities.

Here is the code for segment2:

```
from math import log10

@memo
def segment2(text, prev='<S>'):
    "Return (log P(words), words), where words is the best segmentation."
    if not text: return 0.0, []
    candidates = [combine(log10(cPw(first, prev)), first, segment2(rem, first))
                  for first,rem in splits(text)]
    return max(candidates)

def combine(Pfirst, first, (Prem, rem)):
    "Combine first and rem results into one (probability, words) pair."
    return Pfirst+Prem, [first]+rem
```

segment2 makes O(nL) recursive calls, and each one considers O(L) splits, so the whole algorithm is O(nL^2). In effect this is the *Viterbi* algorithm, with memo implicitly creating the Viterbi tables.

segment2 correctly segments the "sit down" example, and gets right all the examples that the first version got right. Neither version gets the "unregarded" example right.

Could we improve on this performance? Probably. We could create a more accurate model of unknown words. We could incorporate more data, and either keep more entries from the unigram or bigram data, or perhaps add trigram data.

Secret Codes

Our second challenge is to decode a message written in a secret code. We'll look at *substitution ciphers*, in which each letter is replaced by another. The description of what replaces what is called the *key*, which we can represent as a string of 26 letters; the first letter replaces "a", the second replaces "b", and so on. Here is the function to encode a message with a substitution cipher key (the Python library functions `maketrans` and `translate` do most of the work):

```
def encode(msg, key):
    "Encode a message with a substitution cipher."
    return msg.translate(string.maketrans(ul(alphabet), ul(key)))

def ul(text): return text.upper() + text.lower()

alphabet = 'abcdefghijklmnoprstuvwxyz'
```

Perhaps the simplest of all codes is the *shift cipher*, a substitution cipher in which each letter in the message is replaced by the letter *n* letters later in the alphabet. If *n* = 1, then "a" is replaced by "b" and "b" is replaced by "c", up to "z", which is replaced by "a". Shift ciphers are also called *Caesar ciphers*; they were state-of-the-art in 50 BC. The function `shift` encodes with a shift cipher:

```
def shift(msg, n=13):
    "Encode a message with a shift (Caesar) cipher."
    return encode(msg, alphabet[n:]+alphabet[:n])
```

We use the function like this:

```
>>> shift('Listen, do you want to know a secret?')
'Yvfgra, b lbh jnag gb xabj n frperg?'

>>> shift('HAL 9000 xyz', 1)
'IBM 9000 yza'
```

To decode a message without knowing the key, we follow the same methodology we did with segmentations: define a model (we'll stick with unigram word probabilities), enumerate candidates, and choose the most probable. There are only 26 candidate shifts to consider, so we can try them all.

To implement this we define `logPwords`, which is like `Pwords`, but returns the log of the probability, and accepts the input as either a long string of words or a list of words:

```
def logPwords(words):
    "The Naive Bayes probability of a string or seuence of words."
    if isinstance(words, str): words = allwords(words)
    return sum(log10(Pw(w)) for w in words)

def allwords(text):
    "Return a list of alphabetic words in text, lowercase."
    return re.findall('[a-z]+', text.lower())
```

Now we can decode by enumerating all candidates and picking the most probable:

```
def decode_shift(msg):
    "Find the best decoding of a message encoded with a shift cipher."
    candidates = [shift(msg, n) for n in range(len(alphabet))]
    return max(candidates, key=logPwords)
```

We can test to see that this works:

```
>>> decode_shift('Yvfgra, b lbh jnag gb xabj n frperg?')
'Listen, do you want to know a secret?'
```

This is all too easy. To see why, look at the 26 candidates, with their log-probabilities:

```
Yvfgra, b lbh jnag gb xabj n frperg? -84
Zwghsb, rc mci kobh hc ybck o gsfsh? -83
Axhitc, sd ndj lpci id zcdl p htrgti? -83
Byijud, te oek mdj je adem  iushuj? -77
Czjkve, uf pfl nrek kf befn r jvtivk? -85
Daklwf, vg gm osfl lg cfgo s kwujwl? -91
Eblmxg, wh rhn ptgm mh dghp t lxvkxm? -84
Fcmnyh, xi sio uhn ni ehi u mywlyn? -84
Gdnozi, yj tjp rvio oj fijr v nzxmzo? -86
Heopaj, zk uk swjp pk gjks w oaynap? -93
Ifpbk, al vlr txk l hklt x pbzob? -84
Jgrcl, bm wms uylr rm ilmu y capcr? -76
Khrsdm, cn xnt vzms sn jmnv z rdbds? -92
Listen, do you want to know a secret? -25
Mjtufo, ep zpv xbou up lopx b tfdsfu? -89
Nkuvgp, f aw ycpv v mpy c ugetgv? -87
Olvwh, gr brx zdw wr nrz d vhfuhw? -85
Pmwxir, hs csy aerx xs orsa e wigvix? -77
Qnxyjs, it dtz bfsy yt pstb f xjhwjy? -83
Royzkt, ju eua cgtz zu tuc g ykixkz? -85
Spzalu, kv fvb dhua av ruvd h zljyla? -85
Tabmv, lw gwc eivb bw svwe i amkzmb? -84
Urbcnw, mx hxd fjwc cx twxf j bnlanc? -92
Vscdox, ny iye gkxd dy uxyg k combod? -84
Wtdepy, oz jzf hlye ez vyzh l dpncpe? -91
Xuefz, pa kag imzf fa wzai m eodf? -83
```

As you scan the list, exactly one line stands out as English-like, and Pwords agrees with our intuition, giving that line a log-probability of –25 (that is, 10^{-25}), which is 10^{50} times more probable than any other candidate.

The code maker can make the code breaker's job harder by eliminating punctuation, spaces between words, and uppercase distinctions. That way the code breaker doesn't get clues from short words like "I," "a," and "the," nor from guessing that the character after an apostrophe should be "s" or "t". Here's an encryption scheme, shift2, that removes nonletters, converts everything to lowercase, and then applies a shift cipher:

```
def shift2(msg, n=13):
    "Encode with a shift (Caesar) cipher, yielding only letters [a-z]."
    return shift(just_letters(msg), n)

def just_letters(text):
    "Lowercase text and remove all characters except [a-z]."
    return re.sub('[^a-z]', '', text.lower())
```

And here's a way to break this code by enumerating each candidate, segmenting each one, and choosing the one with the highest probability:

```
def decode_shift2(msg):
    "Decode a message encoded with a shift cipher, with no spaces."
    candidates = [segment2(shift(msg, n)) for n in range(len(alphabet))]
    p, words = max(candidates)
    return ' '.join(words)
```

Let's see how well it works:

```
>>> shift2('Listen, do you want to know a secret?')
'yvfgrablbhjnaggbxabjnfrperg'

>>> decode_shift2('yvfgrablbhjnaggbxabjnfrperg')
'listen do you want to know a secret'

>>> decode_shift2(shift2('Rosebud'))
'rosebud'

>>> decode_shift2(shift2("Is it safe?"))
'is it safe'

>>> decode_shift2(shift2("What's the freuency, Kenneth?"))
'whats the freuency kenneth'

>>> msg = 'General Kenobi: Years ago, you served my father in the Clone
Wars; now he begs you to help him in his struggle against the Empire.'

>>> decode_shift2(shift2(msg))
'general kenobi years ago you served my father in the clone wars now he
begs you to help him in his struggle against the empire'
```

Still way too easy. Let's move on to a general substitution cipher, in which any letter can be substituted for any other. Now we can no longer enumerate the possibilities, because there are 26! keys (about 4×10^{26}), rather than just 26. *The Code Book* by Simon Singh (Anchor) offers five strategies (and we'll mention a sixth) for breaking ciphers:

1. Letter unigram frequencies. Match common letters in the message to common letters in English (like "e") and uncommon to uncommon (like "z").
2. Double letter analysis. A double in the coded message is still double in the decoded message. Consider the least and most common double letters.
3. Look for common words like "the," "and," and "of." One-letter words are most often "a" or "I."
4. If possible, get a frequency table made up of the type of messages you are dealing with. Military messages use military jargon, etc.
5. Guess a word or phrase. For example, if you can guess that the message will contain "your faithful servant," try it.
6. Use word patterns. For example, the coded word "abbccddedf" is very likely "bookkeeper," because there are no other words in the corpus with that pattern.

For messages that do not contain spaces between words, strategies 3 and 6 do not apply. Strategies 1 and 2 contain only 26 probabilities each, and seem targeted for a human analyst with limited memory and computing power, not for a computer program. Strategies 4 and 5 are for special-purpose, not general-purpose, decoders. It looks like we're on our own in coming up with a strategy. But we know the methodology.

I. Define a probabilistic model: We could evaluate candidates the same way we did for shift ciphers: segment the text and calculate the probability of the words. But considering step II of the methodology, our first few candidates (or few thousand) will likely be very poor ones. At the start of our exploration, we won't have anything resembling words, so it won't do much good to try to segment. However, we may (just by accident) have decoded a few letters in a row that make sense. So let's use *letter n-grams* rather than words for our language model. Should we look at letter bigrams? 3-grams? 5-grams? I chose 3-grams because they are the shortest that can represent common short words (strategy 3). I created the datafile `count_3l` by counting the letter 3-grams (with spaces and punctuation removed) within the word bigram datafiles (I couldn't just look at the vocabulary file, because I need to consider letter trigrams that cross word boundaries). All of the 26^3 = 17,576 trigrams appear. Here are the top and bottom 10:

```
the  2.763%          fz  0.0000004%
ing  1.471%          jv  0.0000004%
and  1.462%          jn  0.0000004%
ion  1.343%          zh  0.0000004%
tio  1.101%          jx  0.0000003%
ent  1.074%          jw  0.0000003%
for  0.884%          jy  0.0000003%
ati  0.852%          zy  0.0000003%
ter  0.728%          jz  0.0000002%
ate  0.672%          zg  0.0000002%
```

The letter trigram probability is computed like this:

```
def logPletters (text):
    "The log-probability of text using a letter 3-gram model."
    return sum(log10(P3l(g)) for g in ngrams(text, 3))

P3l = Pdist(datafile('count_3l'))
P2l = Pdist(datafile('count_2l')) ## We'll need it later
```

II. Enumerate candidates: We can't consider all 4×10^{26} possible keys, and there does not appear to be a way to systematically eliminate nonoptimal candidates, as there was in segmentation. That suggests a *local search* strategy, such as *hill climbing*. Suppose you wanted to reach maximum elevation, but had no map. With the hill-climbing strategy, you would start at a random location, *x*, and take a step to a neighboring location. If that location is higher, continue to hill-climb from there. If not, consider another neighbor of *x*. Of course, if you start at a random location on Earth and start walking uphill, you probably won't end up on top of Mt. Everest. More likely you'll end up on top of a small local hill, or get stuck wandering around a flat plain. Therefore we add *random restarts* to our hill-climbing algorithm: after we've taken a certain number of steps, we start all over again in a new random location.

Here is the general `hillclimb` algorithm. It takes a starting location, *x*, a function `f` that we are trying to optimize, a function `neighbors` that generates the neighbors of a location, and a maximum number of steps to take. (If the variable `debugging` is true, it prints the best *x* and its score.)

```
def hillclimb(x, f, neighbors, steps=10000):
    "Search for an x that miximizes f(x), considering neighbors(x)."
    fx = f(x)
    neighborhood = iter(neighbors(x))
    for i in range(steps):
        x2 = neighborhood.next( )
        fx2 = f(x2)
        if fx2 > fx:
            x, fx = x2, fx2
            neighborhood = iter(neighbors(x))
    if debugging: print 'hillclimb:', x, int(fx)
    return x

debugging = False
```

To use `hillclimb` for decoding, we need to specify the parameters. The locations we will be searching through will be plain-text (decoded) messages. We will attempt to maximize their letter trigram frequency, so `f` will be `logP3letters`. We'll start with *x* being the message decrypted with a random key. We'll do random restarts, but when we gather the candidates from each restart, we will choose the one best according to `segment2`, rather than by `logP3letters`:

```
def decode_subst(msg, steps=4000, restarts=20):
    "Decode a substitution cipher with random restart hillclimbing."
    msg = cat(allwords(msg))
    candidates = [hillclimb(encode(msg, key=cat(shuffled(alphabet))),
                            logP3letters, neighboring_msgs, steps)
                  for _ in range(restarts)]
    p, words = max(segment2(c) for c in candidates)
    return ' '.join(words)

def shuffled(se):
    "Return a randomly shuffled copy of the input seuence."
    se = list(se)
    random.shuffle(se)
    return se

cat = ''.join
```

Now we need to define `neighboring_msgs`, which generates decryptions of the message to try next. We first try to repair improbable letter bigrams. For example, the least frequent bigram, "jq", has probability 0.0001%, which is 50,000 times less than the most probable bigrams, "in" and "th". So if we see a "jq" in `msg`, we try swapping the "j" with each of the other letters, and also try swapping the "q". If a swap yields a more frequent bigram, then we generate the message that results from making the swap. After exhausting repairs of the 20 most improbable bigrams, we consider random swaps:

```
def neighboring_msgs(msg):
    "Generate nearby keys, hopefully better ones."
    def swap(a,b): return msg.translate(string.maketrans(a+b, b+a))
    for bigram in heap.nsmallest(20, set(ngrams(msg, 2)), P2l):
        b1,b2 = bigram
        for c in alphabet:
            if b1==b2:
                if P2l(c+c) > P2l(bigram): yield swap(c,b1)
            else:
                if P2l(c+b2) > P2l(bigram): yield swap(c,b1)
                if P2l(b1+c) > P2l(bigram): yield swap(c,b2)
    while True:
        yield swap(random.choice(alphabet), random.choice(alphabet))
```

Let's see how well this performs. We'll try it on some ciphers from Robert Raynard's book *Secret Code Breaker* (Smith and Daniel; see *http://secretcodebreaker.com*). First a warm-up message:

```
>>> msg = 'DSDRO XFIJV DIYSB ANQAL TAIMX VBDMB GASSA QRTRT CGGXJ MMTQC IPJSB AQPDR
SDIMS DUAMB CQCMS AQDRS DMRJN SBAGC IYTCY ASBCS MQXKS CICGX RSRCQ ACOGA SJPAS
AQHDI ASBAK GCDIS AWSJN CMDKB AQHAR RCYAE'

>>> decode_subst(msg)
'it is by knowing the freuency which letters usually occur and other distinctive
characteristics of the language that crypt analysts are able to determine the
plain text of a cipher message j'
```

This is correct except that "crypt analysts" should be one word. (It isn't in `Pw`, but it is in the 13-million-word vocabulary.) Note the last character ("E" in the cipher text) was added to make the blocks of five letters come out even.

Now an actual message from Baron August Schluga, a German spy in World War I:

```
>>> msg = 'NKDIF SERLJ MIBFK FKDLV NQIBR HLCJU KFTFL KSTEN YQNDQ NTTEB TTENM QLJFS
NOSUM MLQTL CTENC QNKRE BTTBR HKLQT ELCBQ QBSFS KLTML SSFAI NLKBR RLUKT LCJUK
FTFLK FKSUC CFRFN KRYXB'

>>> decode_subst(msg)
'english complaining over lack of munitions they regret that the promised support of
the french attack north of arras is not possible on account of munition insufficiency
wa'
```

Here's a 1992 message from the KGB to former CIA officer Aldrich Ames, who was convicted of spying in 1994:

```
>>> msg = 'CNLGV QVELH WTTAI LEHOT WEQVP CEBTQ FJNPP EDMFM LFCYF SQFSP NDHQF OEUTN
PPTPP CTDQN IFSQD TWHTN HHLFJ OLFSD HQFED HEGNQ TWVNQ HTNHH LFJWE BBITS PTHDT
XQQFO EUTYF SLFJE DEFDN IFSQG NLNGN PCTTQ EDOED FGQFI TLXNI'

>>> decode_subst(msg)
'march third week bridge with smile to pass info from you to us and to give assessment
about new dead drop ground to indicate what dead drop will be used next to give your
opinion about caracas meeting in october xab'
```

This 1943 message from German U-Boat command was intercepted and decoded, saving a convoy of Allied ships:

```
msg = 'WLJIU JYBRK PWFPF IJQSK PWRSS WEPTM MJRBS BJIRA BASPP IHBGP RWMWQ SOPSV PPIMJ
BISUF WIFOT HWBIS WBIQW FBJRB GPILP PXLPM SAJQQ PMJQS RJASW LSBLW GBHMJ
QSWIL PXWOL'

>>> decode_subst(msg)
'a cony ov is headed northeast take up positions fifteen miles apart between point yd
and bu maintain radio silence except for reports of tactical importance x abc'
```

This answer confuses the “y” and “v.” A human analyst would realize “cony ov” should be “convoy” and that therefore “point yd” should be “point vd.” Our program never considered that possibility, because the letter trigram probability of the correct text is less than the one shown here. We could perhaps fix the problem by inventing a better scoring function that does not get trapped in a local maximum. Or we could add a second level of hill-climbing search: take the candidates generated by the first search and do a brief search with `segment2` as the scoring function. We’ll leave that exercise to the reader.

Spelling Correction

Our final task is spelling correction: given a typed word, *w*, determine what word *c* was most likely intended. For example, if *w* is “acomodation”, c should be “accommodation”. (If *w* is “the”, then *c* too should be “the”.)

Following the standard methodology, we want to choose the *c* that maximizes P(*c* | *w*). But defining this probability is not straightforward. Consider *w* = “thew”. One candidate *c* is “the”—it’s the most common word, and we can imagine the typist’s finger slipping off the “e” key and hitting the “w”. Another candidate is “thaw”—a fairly common word (although 30,000 times less frequent than “the”), and it is common to substitute one vowel for another. Other candidates include “thew” itself (an obscure term for muscle or sinew), “threw”, and “Thwe”, a family name. Which should we choose? It seems that we are conflating two factors: how probable is *c* on its own, and how likely is it that *w* could be a typo for *c*, or a mispronunciation, or some other kind of misspelling. One might think we will have to combine these factors in some ad hoc fashion, but it turns out that there is a mathematical formula, Bayes’s theorem, that tells us precisely how to combine them to find the best candidate:

$$\text{argmax}_c\, P(c \mid w) = \text{argmax}_c\, P(w \mid c)\, P(c)$$

Here P(*c*), the probability that *c* is the intended word, is called the *language model*, and P(*w* | *c*), the probability that an author would type *w* when *c* is intended, is called the *error model* or *noisy channel model*. (The idea is that the ideal author intended to type *c*, but some noise or static on the line altered *c* to *w*.) Unfortunately, we don’t have an easy way to estimate this model from the corpus data we have—the corpus says nothing about what words are misspellings of others.

We can solve this problem with more data: a list of misspellings. Roger Mitton has a list of about 40,000 *c*,*w* pairs at *http://www.dcs.bbk.ac.uk/~ROGER/corpora.html*. But we can't hope to just look up P(*w*=thew | *c*=thaw) from this data; with only 40,000 examples, chances are slim that we will have seen this exact pair before. When data is sparse, we need to generalize. We can do that by ignoring the letters that are the same, the "th" and "w", leaving us with P(*w*=e | *c*=a), the probability that an "e" was typed when the correct letter was "a". This is one of the most common errors in the misspelling data, due to confusions like "consistent/consistant" and "inseparable/inseperable."

In the following table we consider five candidates for *c* when *w*=thew. One is thew itself, and the other four represent the four types of single edits that we will consider: (1) We can *delete* the letter "w" in "thew", yielding "the". (2) We can *insert* an "r" to get "threw". For both these edits, we condition on the previous letter. (3) We can replace "e" with "a" as mentioned earlier. (4) We can transpose two adjacent letters, swapping "ew" with "we". We say that these single edits are at *edit distance* 1; a candidate that requires two single edits is at edit distance 2. The table shows the words *w* and *c*, the edit *w* | *c*, the probability P(*w* | *c*), the probability P(*c*), and the product of the probabilities (scaled for readability).

w	***c***	***w* | *c***	**P(*w* | *c*)**	**P(*c*)**	**10^9 P(*w* | *c*) P(*c*)**
thew	the	ew | e	.000007	.02	144.
thew	thew		.95	.00000009	90.
thew	thaw	e | a	.001	.0000007	0.7
thew	threw	h | hr	.000008	.000004	0.03
thew	thwe	ew | we	.000003	.00000004	0.0001

We see from the table that "the" is the most likely correction. P(*c*) can be computed with `Pw`. For P(*w* | *c*) we need to create a new function, `Pedit`, which gives the probability of an edit, estimated from the misspelling corpus. For example, `Pedit('ew|e')` is .000007. More complicated edits are defined as a concatenation of single edits. For example, to get from "hallow" to "hello" we concatenate `a|e` with `ow|o`, so the whole edit is called `a|e+ow|o` (or `ow|o+a|e`, which in this case—but not always—is the same thing). The probability of a complex edit is taken to be the product of its components.

A problem: what the probability of the empty edit, `Pedit('')`? That is, given that the intended word is *c*, how likely is it that the author would actually type *c*, rather than one of the possible edits that yields an error? That depends on the skill of the typist and on whether any proofreading has been done. Rather arbitrarily, I assumed that a spelling error occurs once every 20 words. Note that if I had assumed errors occur only once in 50 words, then P(*w* | *c*) for *w*="thew" would be .98, not .95, and "thew" would become the most probable answer.

Finally, we're ready to show the code. There are two top-level functions, `correct`, which returns the best correction for a single word, and `corrections`, which applies `correct` to every word in a text, leaving the surrounding characters intact. The candidates are all the possible edits, and the best is the one with the highest P(*w* | *c*) P(*c*) score:

```
def corrections(text):
    "Spell-correct all words in text."
    return re.sub('[a-zA-Z]+', lambda m: correct(m.group(0)), text)

def correct(w):
    "Return the word that is the most likely spell correction of w."
    candidates = edits(w).items()
    c, edit = max(candidates, key=lambda (c,e): Pedit(e) * Pw(c))
    return c
```

P(*w* | *c*) is computed by `Pedit`:

```
def Pedit(edit):
    "The probability of an edit; can be '' or 'a|b' or 'a|b+c|d'."
    if edit == '': return (1. - p_spell_error)
    return p_spell_error*product(P1edit(e) for e in edit.split('+'))

p_spell_error = 1./20.

P1edit = Pdist(datafile('count1edit')) ## Probabilities of single edits
```

The candidates are generated by `edits`, which is passed a word, and returns a dict of {word: edit} pairs indicating the possible corrections. In general there will be several edits that arrive at a correction. (For example, we can get from “tel” to “tell” by inserting an “l” after the “e” or after the “l”.) We choose the edit with the highest probability. `edits` is the most complex function we've seen so far. In part this is inherent; it *is* complicated to generate four kinds of edits. But in part it is because we took some efforts to make `edits` efficient. (A slower but easier-to-read version is at *http://norvig.com/spell-correct.html*.) If we considered *all* edits, a word like “acommodations” would yield 233,166 candidates. But only 11 of these are in the vocabulary. So `edits` works by precomputing the set of all prefixes of all the words in the vocabulary. It then calls `editsR` recursively, splitting the word into a head and a tail (`hd` and `tl` in the code) and assuring that the head is always in the list of prefixes. The results are collected by adding to the dict `results`:

```
def edits(word, d=2):
    "Return a dict of {correct: edit} pairs within d edits of word."
    results = {}
    def editsR(hd, tl, d, edits):
        def ed(L,R): return edits+[R+'|'+L]
        C = hd+tl
        if C in Pw:
            e = '+'.join(edits)
            if C not in results: results[C] = e
            else: results[C] = max(results[C], e, key=Pedit)
        if d <= 0: return
        extensions = [hd+c for c in alphabet if hd+c in PREFIXES]
        p = (hd[-1] if hd else '<') ## previous character
        ## Insertion
        for h in extensions:
            editsR(h, tl, d-1, ed(p+h[-1], p))
        if not tl: return
        ## Deletion
        editsR(hd, tl[1:], d-1, ed(p, p+tl[0]))
        for h in extensions:
            if h[-1] == tl[0]: ## Match
                editsR(h, tl[1:], d, edits)
            else: ## Replacement
                editsR(h, tl[1:], d-1, ed(h[-1], tl[0]))
        ## Transpose
        if len(tl)>=2 and tl[0]!=tl[1] and hd+tl[1] in PREFIXES:
            editsR(hd+tl[1], tl[0]+tl[2:], d-1,
                   ed(tl[1]+tl[0], tl[0:2]))
    ## Body of edits:
    editsR('', word, d, [])
    return results

PREFIXES = set(w[:i] for w in Pw for i in range(len(w) + 1))
```

Here's an example of `edits`:

```
>>> edits('adiabatic', 2)
{'adiabatic': '', 'diabetic': '<a|<+a|e', 'diabatic': '<a|<'}
```

And here is the spell corrector at work:

```
>>> correct('vokabulary')
'vocabulary'

>>> correct('embracable')
'embraceable'

>>> corrections('thiss is a teyst of acommodations for korrections
of mispellings of particuler wurds.')
'this is a test of acommodations for corrections of mispellings
of particular words.'
```

Thirteen of the 15 words were handled correctly, but not "acommodations" and "mispellings". Why not? The unfortunate answer is that the Internet is full of lousy spelling. The incorrect "mispellings" appears 18,543 times in the corpus. Yes, the correct word, "misspellings", appears half a million times, but that was not enough to overcome the bias for no edit over a single edit. I suspect that most of the 96,759 occurrences of "thew" are also spelling errors.

There are many ways we could improve this spelling program. First, we could correct words in the context of the surrounding words, so that "they're" would be correct when it stands alone, but would be corrected to "their" when it appears in "in their words." A word bigram or trigram model would do the trick.

We really should clean up the lousy spelling in the corpus. Look at these misspellings:

```
misspellings    432354
mispellings     18543
misspelings     10148
mispelings      3937
```

The corpus uses the wrong word 7% of the time. I can think of three ways to fix this. First, we could acquire a list of dictionary words and only make a correction to a word in the dictionary. But a dictionary does not list all the newly coined words and proper names. (A compromise might be to force lowercase words into the dictionary vocabulary, but allow capitalized words that are not in the dictionary.) Second, we could acquire a corpus that has been carefully proofread, perhaps one restricted to books and periodicals from high-quality publishers. Third, we could spell-correct the corpus we have. It may seem like circular reasoning to require spell-correction of the corpus before we can use it for spell-correction, but it can be done. For this application we would start by grouping together words that are a small edit distance from each other. For each pair of close words, we would then check to see if one is much more common than the other. If it is, we would then check the bigram (or trigram) counts to see if the two words had similar distributions of neighboring words. For example, here are four bigram counts for "mispellings" and "misspellings":

```
mispellings allowed     99          misspellings allowed    2410
mispellings as          50          misspellings as          749
mispellings for        122          misspellings for       11600
mispellings of        7360          misspellings of        16943
```

The two words share many bigram neighbors, always with "misspellings" being more common, so that is good evidence that "mispellings" is a misspelling. Preliminary tests show that this approach works well—but there is a problem: it would require hundreds of CPU hours of computation. It is appropriate for a cluster of machines, not a single computer.

How does the data-driven approach compare to a more traditional software development process wherein the programmer codes explicit rules? To address that, we'll peek at the spelling correction code from the ht://Dig project, an excellent open source intranet search engine. Given a word, ht://Dig's `metaphone` routine produces a *key* representing the sound of the word. For example, both "tough" and "tuff" map to the key "TF", and thus would be candidates for misspellings of each other. Here is part of the `metaphone` code for the letter "G":

```
case 'G':
        /*
         * F if in -GH and not B--GH, D--GH,
         * -H--GH, -H---GH else dropped if
         * -GNED, -GN, -DGE-, -DGI-, -DGY-
         * else J if in -GE-, -GI-, -GY- and
         * not GG else K
         */
      if ((*(n + 1) != 'G' || vowel(*(n + 2))) &&
          (*(n + 1) != 'N' || (*(n + 1) &&
                              (*(n + 2) != 'E' ||
                               *(n + 3) != 'D'))) &&
          (*(n - 1) != 'D' || !frontv(*(n + 1))))
        if (frontv(*(n + 1)) && *(n + 2) != 'G')
          key << 'J';
        else
          key << 'K';
      else if (*(n + 1) == 'H' && !noghf(*(n - 3)) &&
               *(n - 4) != 'H')
               key << 'F';
      break;
```

,ch14.14922 Page 239 Thursday, June 25, 2009 2:34 PM

This code correctly maps "GH" to "F" in "TOUGH" and not in "BOUGH". But have the rules correctly captured all the cases? What about "OUGHT"? Or "COUGH" versus "HICCOUGH"? We can write test cases to verify each branch of the code, but even then we won't know what words were *not* covered. What happens when a new word like "iPhone" is introduced? Clearly, the handwritten rules are difficult to develop and maintain. The big advantage of the data-driven method is that so much knowledge is encoded in the data, and new knowledge can be added just by collecting more data. But another advantage is that, while the data can be massive, the code is succinct—about 50 lines for `correct`, compared to over 1,500 for ht://Dig's spelling code. As the great ex-programmer Bill Gates once said, "Measuring programming progress by lines of code is like measuring aircraft building progress by weight." Gates knew that lines of code are more a liability than an asset. The probabilistic data-driven methodology is the ultimate in agile programming.

Another issue is portability. If we wanted a Latvian spelling-corrector, the English `metaphone` rules would be of little use. To port the data-driven `correct` algorithm to another language, all we need is a large corpus of Latvian; the code remains unchanged.

Other Tasks

Here are some more tasks that have been handled with probabilistic language models.

Language Identification

There are web protocols for declaring what human language a page is written in. In fact there are at least two protocols, one in HTML and one in HTTP, but sometimes the protocols disagree, and sometimes they both lie, so search engines usually classify pages based on the actual content, after collecting some samples for each known language. Your task is to write such a classifier. State of the art is over 99% accuracy.

Spam Detection and Other Classification Tasks

It is estimated that 100 billion spam email messages are sent every day. Given two corpora of spam and nonspam messages, your task is to classify incoming messages. The best spam classifiers have models for word *n*-grams (a message with "10,000,000.00 will be released" and "our country Nigeria" is probably spam) and character *n*-grams ("v1agra" is probably spam), among other features. State of the art on this task is also over 99%, which keeps the spam blockers slightly ahead of the spammers. Once you can classify documents as spam/nonspam, it is a short step to do other types of classification, such as urgent/nonurgent email messages, or politics/business/sports/etc. for news articles, or favorable/neutral/unfavorable for product reviews.

Author Identification (Stylometry)

Language models have been used to try to identify the disputed authors of the Federalist Papers, Shakespeare's poems, and Biblical verses. Similar techniques are used in tracking terrorist groups and in criminal law, to identify and link perpetrators. This field is less mature; we don't yet know for sure what the best practices are, nor what accuracy rate to expect, although the winner of a 2004 competition had 71% accuracy. The best performers in the competition were linguistically simple but statistically sophisticated.

Document Unshredding and DNA Sequencing

In Vernor Vinge's science fiction novel *Rainbows End* (Tor Books), the Librareome project digitizes an entire library by tossing the books into a tree shredder, photographing the pieces, and using computer algorithms to reassemble the images. In real life, the German government's E-Puzzler project is reconstructing 45 million pages of documents shredded by the former East German secret police, the Stasi. Both these projects rely on sophisticated computer vision techniques. But once the images have been converted to characters, language models and hill-climbing search can be used to reassemble the pieces. Similar techniques can be used to read the language of life: the Human Genome Project used a technique called shotgun sequencing to reassemble shreds of DNA. So-called "next generation sequencing" shifts even more of the burden away from the wet lab to large-scale parallel reassembly algorithms.

Machine Translation

The Google *n*-gram corpus was created by researchers in the machine translation group. Translating from a foreign language (f) into English (e) is similar to correcting misspelled words. The best English translation is modeled as:

$$best = \text{argmax}_e \, P(e \mid f) = \text{argmax}_e \, P(f \mid e) \, P(e)$$

where $P(e)$ is the language model for English, which is estimated by the word *n*-gram data, and $P(f \mid e)$ is the translation model, which is learned from a bilingual corpus: a corpus where pairs of documents are marked as translations of each other. Although the top systems make use of many linguistic features, including parts of speech and syntactic parses of the sentences, it appears that the majority of the knowledge necessary for translation resides in the *n*-gram data.

Discussion and Conclusion

We have shown the power of a software development methodology that uses large amounts of data to solve ill-posed problems in uncertain environments. In this chapter it was language data, but many of the same lessons apply to other data.

In the examples we have explored, the programs are simple and succinct because the probabilistic models are simple. These simple models ignore so much of what humans know—clearly, to segment "choosespain.com", we draw on much specific knowledge of how the travel business works and other factors, but the surprising result is that a program does not have to explicitly represent all that knowledge; it gets much of the knowledge implicitly from the *n*-grams, which reflect what other humans have chosen to talk about. In the past, probabilistic models were more complex because they relied on less data.

There was an emphasis on statistically sophisticated forms of *smoothing* when data is missing. Now that very large corpora are available, we use approaches like stupid backoff and no longer worry as much about the smoothing model.

Most of the complexity in the programs we studied in this chapter was due to the search strategy. We saw three classes of search strategy:

Exhaustive
: For shift ciphers, there are only 26 candidates; we can test them all.

Guaranteed
: For segmentation, there are 2^n candidates, but most can be proved nonoptimal (given the independence assumption) without examining them.

Heuristic
: For full substitution ciphers, we can't guarantee we've found the best candidate, but we can search a representative subset that gives us a good chance of finding the maxima. For many problems, the majority of the work will be in picking the right function to maximize, understanding the topology of the search space, and discovering a good order to enumerate the neighbors.

If we are to base our models on large amounts of data, we'll need data that is readily available "in the wild." *N*-gram counts have this property: we can easily harvest a trillion words of naturally occurring text from the Web. On the other hand, labeled spelling corrections do not occur naturally, and thus we found only 40,000 of them. It is not a coincidence that the two most successful applications of natural language—machine translation and speech recognition—enjoy large corpora of examples available in the wild. In contrast, the task of syntactic parsing of sentences remains largely unrealized, in part because there is no large corpus of naturally occurring parsed sentences.

It should be mentioned that our probabilistic data-driven methodology—maximize the probability over all candidates—is a special case of the *rational* data-driven methodology—maximize expected utility over all candidates. The *expected utility* of an action is its average value to the user over all possible outcomes. For example, a rational spelling correction program should know that there are some taboo naughty words, and that suggesting them when they were not intended causes embarrassment for the user, a negative effect that is much worse than just spelling a word wrong. The rational program takes into account both the probability that a word is correct or wrong, and the positive or negative value of suggesting each word.

Uncertain problems require good discipline in validation and testing. To evaluate a solution to an uncertain problem, one should divide the data into three sets: (1) A *training* set used to build the probabilistic model. (2) A *validation* set that the developer uses to evaluate several different approaches, seeing which ones score better, and getting ideas for new algorithms. (3) A *test* set, which is used at the end of development to accurately judge how well the algorithm will do on new, unseen data. After the test set has been used once it ideally should be discarded, just as a teacher cannot give the same test twice to a class of students. But in practice, data is expensive, and there is a trade-off between paying to acquire new data and reusing old data in a way that does not cause your model to *overfit* to the data. Note that the need for an independent test set is inherent to uncertain problems themselves, not to the type of solution chosen—you should use proper test methodology even if you decide to solve the problem with ad hoc rules rather than a probabilistic model.

In conclusion, as more data is available online, and as computing capacity increases, I believe that the probabilistic data-driven methodology will become a major approach for solving complex problems in uncertain domains.

Acknowledgments

Thanks to Darius Bacon, Thorsten Brants, Andy Golding, Mark Paskin, and Casey Whitelaw for comments, corrections, and code.