6.863 – Natural Language Processing Notes for Lectures 2 and 3, Fall 2012

MENU: on board

- **RR#1: Google smackdown**
- **Goal – Land you in jail (again): build a language 'identifier' (language ID) like Google's**
- **Method: build 2 'language models', Spanish, English**
    - **What is a language model? (LM)**
    - **Trigram. How to use it for Language ID? *p*(text|eng) vs. *p*(text |Spanish)**
    - **How do we learn language models from training data?**
    - **How to be perplexed: how do we evaluate LMs?**
    - **Never trust a sample under 30: how to be smooth & the Chinese restaurant syndrome**
    - **The adventures of Robin Hood**
    - **Revering Reverend Bayes**

We've just explored in our discussion one of the simplest ways to formulate what's called a *language model n*-grams. We'll get to the formal definition in a moment, but you have already been 'sensitized' that a 2-gram (a first order Markov model), uses one word of preceding context, a 3-gram uses the 2 preceding words.

Now, a *language model* is simply any probability distribution over sentences. Now, n-grams look pretty simple. And they have their weaknesses, both conceptual and practical. So here's one: what about a sentence like this:

Which pigeons/pigeon that John said he would chase are/is in the Stata Center?

Q: can an n-gram model work on the connection between *pigeons/pigeon* and *are/is*? A so-called 'unbounded dependency'
Ans: Yes, if context is long enough… but…note in between *pigeons/pigeon* we don't *care* what the context is, yet an *n*-gram forces us to estimate it all…so this model is not quite right!)
Q: what's the *longest* such unbounded dependency found *in practice*?
A: Proust's 3-volume "In Search of Lost Time" (à la Recherche du Temps Perdue) – first sentence, "Longtemps, je me suis couché de bonne heure"… last sentence: "…many, many days, in the dimension of time" *temps*, connects to the very first, about a million words later… do we want/need a million-gram? (We probably want *different* levels of description)

Strengths: In fact, as it turns out, in practice it can be pretty hard to beat trigrams. Besides, the methods used to build n-gram (or trigram) language models will be used again and again in more complex models, defined over more than just 'beads on a string' (structures, hierarchical objects, syntax).

To be concrete here, let's follow on from where we closed last time, with Google Translate. We will aim to build one part of it we quickly mentioned last time: you see the little "Detect language" tab? How does that work? You'll see how to reverse engineer this – again throwing you to the mercy of the Google patent attorneys.

Here's the overview.
1. Build trigram language model trained on English, and others on (all the other) languages (Spanish, Elvish, etc.) What's a *language model*? Estimate parameters for the model, $q_{\text{ENGLISH}}(w_i \mid w_{i-2}, w_{i-1})$ and $q_{\text{SPANISH}}(w_i \mid w_{i-2}, w_{i-1})$ How? From training data. Now, each of these models assigns a probability to any sequence of words.

2. So, given some previously *unseen* test data sequence we then calculate the probability of the test data as estimated by the English language model, and the Spanish language model i.e.,

$$P_{English}(\textit{test-data}|\text{ english}),\ P_{Spanish}(\textit{test-data}|\text{spanish}),$$

3. See which probability is larger – that's what the language of the test data is.
Easy, no? (There are some improvements we will come to later.) We can use this same idea for many things: spam detection, dialog or speech detection.

Summary, the 4-step program to build a language detector:
1. Define a (probabilistic) language model;
2. Show specifically how to build a trigram language model
3. Show how to learn (estimate) the parameters from training data, including how to deal with the issue of sparse data
4. Show how to evaluate or test two language models, to see which one is better than another.

So: we can use a *trigram model* as our *language model*, and get a system that works quite well at detecting languages. Our goal will be a probabilistic model that we *train* on a sample of English, Spanish, Welsh, Elvish, whatever, but for our purposes just English. So something like this, broken out into letters. (why broken into letters?)

English training data (1K, 10K, 50K words…)

```
n e d SPACE i n SPACE 1 9 5 4 . SPACE I t s SPACE c e n t r a l SPACE l a b o r a t
o r i e s SPACE a r e SPACE s t i l l SPACE l o c a t e d SPACE h e r e . SPACE T h
e SPACE b u i l d i n g SPACE w a s SPACE d o n a t e d SPACE b y SPACE t h e SPACE
G o v e r n m e n t SPACE o f SPACE G u a t e m a l a . SPACE A l t h o u g h
SPACE I N C A P SPACE i s SPACE f u n d
e d SPACE p r i m a r i l y SPACE t h r o u g h SPACE q u o t a SPACE c o n
t r i b u t i o n s SPACE f r o m SPACE P A H O SPACE m e m b e r SPACE c o
u n t r i e s , SPACE i t SPACE a l s o SPACE r e c e i v e s SPACE l
o a n s SPACE f r o m SPACE v a r i o u s SPACE a g e n c i e s SPACE c o n
c e r n e d
```

Spanish training data:

```
SPACE l a SPACE p r e v e n c i o n SPACE d e l SPACE S I D A SPACE p o r
SPACE m e d i o SPACE d e l SPACE P r o y e c t o SPACE P r e v e n c I o n
SPACE d e l SPACE S I D A SPACE e n SPACE e l SPACE t r a b a j o . SPACE
E n SPACE l a SPACE i n v e s t i g a c i o n : SPACE P r o m o c i o n
SPACE y SPACE p a r t i c i p a c i o n SPACE e n SPACE p r o y e c t o s
SPACE
```

d e SPACE i n v e s t i g a c i o n SPACE s o b r e SPACE l a SPACE u t I l i z a c i o n SPACE d e SPACE n u e v o s SPACE m e d i c a m e n t o s , SPACE y SPACE e n SPACE l o s SPACE e s t u d i o s SPACE s o b r e SPACE l a SPACE i n f e c c i o n SPACE p o r SPACE V I H SPACE q u e SPACE r e a l i z a SPACE e l SPACE E s t a d o SPACE d e SPACE S a o SPACE P a u l o . SPACE E j e c u c i o n SPACE d e SPACE a c t i v i d a d e s

OK, so the idea is that we train *two* models on these *two* sets of data, one English, one Spanish, and the intuition is that if we now get a bit of *test* data, where we don't know the language, say this:

u s s a space i s space t a space t o u space m i n e space t I n d e r a space v e space t o u . E r u space i l y a space t o u space m i n e space f i r y a space k u r u v a r ! space w a s s u p space o space t a ?

This, of course, is Elvish. I meant this:

i o n SPACE a n d SPACE c h a r a c t e r i s t i c s SPACE o f SPACE p r o p o s a l s SPACE i n SPACE t h e SPACE a r e a SPACE o f SPACE W H D SPACE a n d SPACE p r o p o s a l s SPACE w i t h SPACE a SPACE c o m p o n e n t

All we are computing is the likelihood that the test string would be generated from the trigram model for English, and the trigram model for Spanish, and then seeing which likelihood is greater.

First we must ask: just what *is* a **language model?** To begin, assume we have a finite set of sentences, a *corpus*, say, the last 3 years of the Wall Street Journal. Given a corpus, a (finite!) set of sentences, we want to estimate a probability over those sentences. That's a language model. (Note: perhaps this isn't such a good idea…shouldn't the corpus be infinite…?) A language model is just a probability distribution over **sentences**, where **sentences are defined as sequences of words over the vocabulary.**

A language model thus has **three** parts: (1) a Vocabulary; (2) a way to define sentences over the vocabulary; and (3) a probability distribution over the possible sentences.

Our first first step is to specify the set of all **words** in the language. We will say that this is a **finite set *V*.** (Why did we make that decision? Ans: smoothing methods have to allocate probability, and this turns out to be easier if the vocabulary is finite.)

But how can *V* be finite? Aren't there infinitely many possible words in English? Yes! Suppose we see in the WSJ one day, "I saw the snuffleupagus on TV" where "snuffleupagus" isn't known. (So it's in our *test* data.) What should we do?

We will use a standard Trick – we will **force** the vocabulary *V* to be finite by adding a special 'word' **OOV** that stands for 'out of vocabulary'. We set our *V* to be all the words in the training corpus, and then add 1 special new word, oov. So all unknown words get mapped to this. You can think of an unknown word as nothing more than various spellings of the OOV symbol. So when you are considering the test sentence:

i saw the fruitylicious soup on the tv

What you will actually compute is the probability of:

i saw the oov soup on the tv

which is really the total probability of all sentences of the form:

i saw the [some out-of-vocabulary-word] soup on the tv

Q: is there an issue here? Yes, a bit: this total probability is higher than the probability of the *particular* sentence involving *fruitylicious*. You should think about why this is, or is not OK.

In fact, this is usually OK, since usually we are just comparing the sentence with *fruitylicious* under 2 (or more) models for the language. Raising the probability of the *fruitylicious* sentence raises the probability under *all* the models at once, so it does not invalidate the comparison.

Q: What did we finesse here in the case of English vs. Spanish – look at the training data.
A: We removed the accent marks in Spanish – this makes distinguishing the two languages harder (otherwise it's a bit too easy).

In general, when one does the language ID task, one must merge *all* the alphabets together!

If we replace any unseen word in the training data by oov, then this works because then any code you write doesn't have to actually detect oov words: if it sees in the test data: "I saw the fruitylicious", it can just look up the count in the usual way, and you'll get the right answer, 0, since fruitylicious never appeared in the training corpus.

In any case. V'= OOV $\cup$ V. We'll define V^* to be the set of *all* sentences with vocabulary V; this is an infinite set.
We'll also need to add a special symbol STOP$\notin V'$ that will always mark the end of a sentence, so we have: V''= STOP $\cup$ V'. So, some example sentences:

Mr. Rogers saw them STOP
the STOP
the neighborhood STOP
sweater sweater STOP

More commonly in practice, we will use the xml symbol </s> for the 'stop' symbol. We'll also use <s> for the 'start of sentence' symbol (and we'll need a special symbol for the symbol *before* the start symbol – since we'll be doing trigrams.)

Now we can proceed to our definition:
Definition. A *language model consists of a finite set V, and a function* $p(x_1, x_2, \ldots, x_n)$ such that:

(1) for any sequence $< x_1 \ldots x_n > \in V^*$, $p(x_1, \ldots, x_n) \geq 0$

$$(2) \quad \sum_{<x_1,\ldots x_n>\in V^*} p(x_1,x_2,\ldots x_n) = 0$$

So p is a probability distribution over the sentences in V^*.

2. Estimating language models – the general picture.

To actually find a model, we have to *estimate* this distribution p. A *really bad* estimation simply says that $p(x_1, x_2,\ldots,x_n)$ is just the $count(x_1 x_2,\ldots x_n)$ divided by N, the total # of sentences in the corpus.

Q: Why is this bad?
A: It gives a probability of 0 to any example sentence that is not in the corpus.

The *simplest* estimate for any single word is just the 'zero knowledge' estimate: if there are $|V|$ items in the vocabulary, then the probability of observing any individual word is just $1/|V|$. This is called the uniform probability estimate. (The prior probability of observing any single **word.)** Clearly, if the sentence is n words long, the probability is just $(1/|V|)^n$.

Q: suppose we see 19,999 different words in our training data, plus the oov word, so this comes to 20,000 for $|V|$. What would go wrong with the UNIFORM estimate if we mistakenly took V=19,999?

A: The probabilities will add up to more than 1 for any text that has an oov word. (each non-oov word is given probability mass 1/19,999 > 1/20,000. If we include the oov word in our calculations, the total probability mass allocated is 20,000/19,999> 1. Not good!)

This is a model of language as 'beads on a string' – obviously not quite right. (Example: *What did you say Mary wanted want to buy?; deep blue sky; unlockable*)

So, given a training corpus, how do we learn the function p? Do this by first defining *Markov models*, and then *trigram language models*, that use these ideas.

1. Markov models

Original source: Andrey Markov, ."Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga". Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete, 2-ya seriya, tom 15, pp. 135–156, 1906.

Markov analyzed the frequencies of word use in Pushkin's *Eugene Onegin*, the first 20K words. (Without a computer!)

Consider a sequence of random variables $X_1, X_2,\ldots,X_n$, where each X_i can take on one of the values in a finite set V. For now, assume length fixed, n=50, say. (We will easily generalize this by adding the STOP or </s> symbol.)

Goal: model probability of any sequence $x_1, x_2,\ldots,x_n$ where $n\geq 1$ and $x_i \in V$, for i=1, 2, …, n, that is, the joint probability

$$P(X_1=x_1, X_2=x_2,\ldots, X_n = x_n)$$

There are $|V^n|$ possible sequences, so we can't list them all. Thus, we will want a more compact model than this!

In a first-order Markov model, we make the following assumption: that the identity of the next word in a sequence depends *only* on the identity of the previous word. So we take the exact formula for a conditional probability, where the i^{th} word depends on *all* the preceding words that came before it, and 'back off' this estimate to the approximation that it just depends on the immediately preceding word:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) =$$

$$P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i \mid X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) \approx$$

$$P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i \mid X_{i-1} = x_{i-1})$$

While the first equation is exact, the second is only an approximation, because the second step makes the assumption that for any $i \in \{2, \ldots, n\}$, for any $x_1, \ldots, x_n$,

$$P(X_i = x_i \mid X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) = P(X_i = x_i \mid X_{i-1} = x_{i-1})$$

This is a so-called *first order Markov* assumption. (The value of X_i is conditionally independent of any words before the immediately preceding one.)

A *second order Markov* process, or a *trigram model*, we assume that each word depends on the preceding two.

$$P(X_i = x_i \mid X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) \approx P(X_i = x_i \mid X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

Since a trigram model must always be able to look at the *two* preceding symbols in a sentence, even at the beginning, we will define a special start symbol *, so that x_{-1}=*, and x_0=0. (But in practice, as mentioned people often use the xml tag <s> for x_0.)

Let's relax assumption that sequences are fixed at length n. Since the length of a sentence can actually vary (n is itself a random variable), we will assume that the n^{th} word in any sequence is the special symbol STOP (or </s>). Now each sequence is a set of n random variables.

We can imagine *generating* the sequence by selecting each x_i from the distribution:

$$P(X_i = x_i \mid X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

We select either from V or STOP (or </s>) at each step. If we hit STOP, we STOP.

So we have that for any i, for any x_{i-2}, x_{i-1}, x_i,

$$P(X_i = x_i \mid X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) = q(x_i \mid x_{i-1}, x_{i-2})$$

where $q(w|u,v)$ for any (u,v,w) is a parameter of the model. We will need to derive estimates for each of the q's from our training corpus. The model then takes the form (for any sequence $x_1 x_2 \ldots, x_n$):

$$p(x_1 \ldots x_n) = \prod_{i=1}^{n} q(x_i \mid x_{i-2}, x_{i-1})$$

Trigram Language model.
In particular, the 'preceding 2 words' context works very well in many NLP applications. So we will drill down a bit into this. First, we see how to define a trigram language model; then how to train it; then how to test it.

Definition. *A trigram language model consists of a finite set V and a parameter* $q(w \mid u,v)$ *for each trigram u,v,w such that* $w \in V \cup \{</s>\}$ *and* $u,v \in V \cup \{<s>\}$. *The value* $q(w \mid u,v)$ *can be interpreted as the probability of seeing the word w immediately after seeing words u, v in that order i.e., after the bigram* (u,v). *For any sentence* $x_1, x_2, \ldots x_n$ *where* $x_i \in V$ *for* $i=1,\ldots,n-1$ *and* $x_n = </s>$, *the probability of the sentence under the trigram language model is*

$$p(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} q(x_i \mid x_{i-2}, x_{i-1})$$

where $x_{-1}=x_0=*$. (NB: again, in practice, for x_0 we use <s>, and, say, * for x_{-1}.

Let's check our knowledge about the notation with a simple example. You turn on the radio to NPR and here are the first descriptions of three possible transcriptions in our notation, A, B, and C. Match these with their correct verbal descriptions 1, 2, and 3 that follow.

(A) p(do) p(you | do) p(think | do, you)
(B) p(do|*, <s>) p(you |<s>, do) p(think | do, you) p(</s> | you, think)
(C) p(do|*, <s>) p(you |<s>, do) p(think | do, you)

(1) The first **complete sentence** you hear is: 'do you think' (B)
(2) The first **three words** you hear are: 'do you think' (A)
(3) The first sentence you hear **starts with** 'do you think' (C)

Here's a simple example.

p(*it is a truth universally STOP*) =
q(*it* | *, *)
x q(*is* | *, *it*)
x q(*a* | *it, is*)
x q(*truth* | *is, a*)
x q(*universally* | *a, truth*)
x q(STOP | *truth, universally*)

Estimation problem: how do we learn (estimate) the *q*'s from training data?

Simple-minded case: use maximum-likelihood estimates (MLE).

- MLE estimate for unigram = count(w_i)/count(), where count()= total # words in corpus
- MLE estimate for bigram = $q_{ML}(w_i \mid w_{i-1})$= bigram count/unigram count $\frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$
- MLE estimate for trigram:
  $q_{ML}(w_i \mid w_{i-2}, w_{i-1})$= trigram count/preceding bigram = $\frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})}$
  e.g., $q_{ML}(truth \mid is, a)$ = count(*is, a, truth*)/count(*is, a*)

MLE estimate will work great on the *training* data (it is guaranteed to get the 'true' answer as the amount of data goes to infinity) but will do *lousy* on test data.

Q: Why does MLE do so badly? Why is this such a bad idea in practice?
There are $|V|^3$ parameters (all the different trigrams).
Suppose $|V| = 10^4$ or 10^6, there are then 10^{12} or 10^{18} parameters to estimate.

Problems:
(1) MOST counts in the numerator, for most trigrams, will be 0, so estimates will be 0 for these;
(2) If a bigram is 0, the estimate is undefined.
In short: MLE estimates assume perfect knowledge acquired from training data.
In fact, most word sequences are very 'rare' events – even if we don't realize it, so what feels like a perfectly common sequence of words may be too rare to have actually occurred in a training corpus, for instance:

In a ___ the last ___ shot a ____ open the ____
With a ___ over my ___ keep tabs ____

Example Corpus: five Jane Austen novels

- N = 617,091 words (# tokens)
- V = 14,585 unique words (# types)
- Task: predict the next word of the trigram "inferior to _____"
- But from test data, *Persuasion:* "[In person, she was] inferior to **both** [**sisters**.]"
- Question: how many possible bigrams are there?

Ans: $V^2 = 212 \times 10^6$ but only 0.5M words, 10^3 so most trigrams will have count 0!
But that is our MLE prediction – *it is just plain wrong.*

Google answer: Why not just collect more data?
But: the event space in NLP is really unbounded! And if you *do* have enough data to estimate all those parameters, you should probably boost the *n* in your *n*-gram model, to get a better model, and then you'll face the same problem all over again.

In fact, one could say that if data sparsity is *not* a problem for you in NLP, then your model is *too simple*. We'll return to this issue in a moment.

First, let's be positive. As crude as they are, simple bigram and trigram models *do* capture a range of interesting facts about language – or rather, *use of language*. (An important distinction.)

For example, the following are some of the *bigram* estimates from a 80,000 word corpus of dialogue, from Berkeley.

1. P(english|want) = .0011
2. P(chinese|want) = .0065
3. P(to|want) = .66
4. P(eat | to) = .28
5. P(food | to) = 0
6. P(want | spend) = 0
7. P (i | <s>) = .25

Q: Which are these bigrams reflect syntax type constraints? (3, 4)
Which of these reflect discourse constraints? (7)
What about 1 and 2? (world knowledge) "I want chinese food". But nobody ever says: "I want English food" (there's no such thing – ask anyone from England – aside from "chippies" – 'fish and chips')

Before moving on to dealing with estimation in these models, let us see how to evaluate a language model in general. How do we tell that one model is better than another? If we think of our English/Spanish detector, then what we want to do is to see which one is better at predicting previously *unseen* sentences. A very common measure is the *perplexity* of the model on unseen (held-out) data, the *test* data. **Perplexity is a per-word average of the probability with which the language model generates the (previously unseen) test data set, where the average is over the # of words in the test data set.** Let's see how this works in more detail.

OK, assume we have *test* data, that has been held out, *unseen*, and *not* part of the training corpus. There are *m* sentences in the test data, $s_1, s_2 \ldots, s_m$.

We now look at how well the language model *predicts* these sentences – does it assign high probability to the *test* sentences – the closer the better. To find the *overall* score of our model, we just multiply the probability the language model assigns to all the test sentences together, so the *larger* the probability value, the *better.* Just add up all the probabilities the model assigns to the *test* corpus:

$$p(s_1)+p(s_2)+ \textit{ etc.}$$

$$\prod_{i=1}^{m} p(s_i)$$

This will get very small very fast, so let's work in log space instead (log to base 2):

$$\log_2 \prod_{i=1}^{m} p(s_i) = \sum_{i=1}^{m} \log_2 p(s_i)$$

(Typically, in implementations, you convert to log space, then untransform back on the way out.)

Now we take the *average* of this value over the entire corpus by dividing by the total # of words in the test corpus, *M* (the length of the testcorpus in words):

$$\frac{1}{M}\sum_{i=1}^{m}\log_2 p(s_i)$$

Call this average log prob value *l*. Then *perplexity* is defined as the negative of this raised to the power of 2:

$$2^{-l}$$

where

$$l=\frac{1}{M}\sum_{i=1}^{m}\log p(s_i)$$

Alternatively:

$$Perplexity= 2^{-\frac{1}{M}\sum_{i=1}^{m}\log p(s_i)}$$

We take the **negative** of the average log probability, and raise it to the power of 2. Perplexity is thus a positive number. The *smaller* the perplexity, the *better* the language model is at modeling unseen data.

Q: What is the <u>best</u> perplexity can get? If *prob*(test-data)=1 according to the model, the model can <u>perfectly reproduce</u> the test data, so $\log_2(1)=0$, therefore perplexity is $2^0=1$.
Q: What is the worst case for perplexity? If the model assigns 0 probability to *any* sentence, then log(0) is +infinity! Very bad! (You should try to AVOID this in your grammar competition late on!)

As the average log-2 probability over the test sentences gets higher, the language model is doing a better and better job of predicting the test sentences. Then 1/avg log-2 prob is getting *smaller*, approaching 1.

An intuition behind perplexity: Perplexity as a measure of 'compression'.
The intuition is that *perplexity* is the effective branching size of the vocabulary.

Suppose that the size of the vocabulary is $|V|=N$, and a 'dumb' model just predicts the uniform probability for every parameter, $q(w|u, v)=1/N$. Then the probability of the corpus under this uniform model is just M x $1/N$, so the average probability is just $1/N$. The log of this is log $1/N$. By the definition of perplexity, this means that the perplexity = $2^{-\log 1/N}= N$. So, the perplexity under the model that assigns a uniform probability to every vocabulary item is just the size of the vocabulary, *N*.

That is, we get a branching factor of *N* if every word is equally likely. But every word is NOT equally likely – they clump together! – and that is precisely what the language model captures. (We never see 'the the'.)

Perplexity is thus the *effective branching factor* – how much the model 'compresses' the vocabulary lower than *N*. For example, if the vocabulary size is 10^4, if the perplexity of a probability model for over a corpus using that vocabulary is 110, the branching factor is reduced from 10,000 to 100.
Note that if *any* estimate for $q(w|u,v)$ in the test data is 0 – that is, the model predicts a probability of 0 for some test data parameter – then the perplexity for the *entire* corpus goes to +infinity (average of the log probability goes to +infinity). ***Thus it is essential to avoid zero estimates*!**

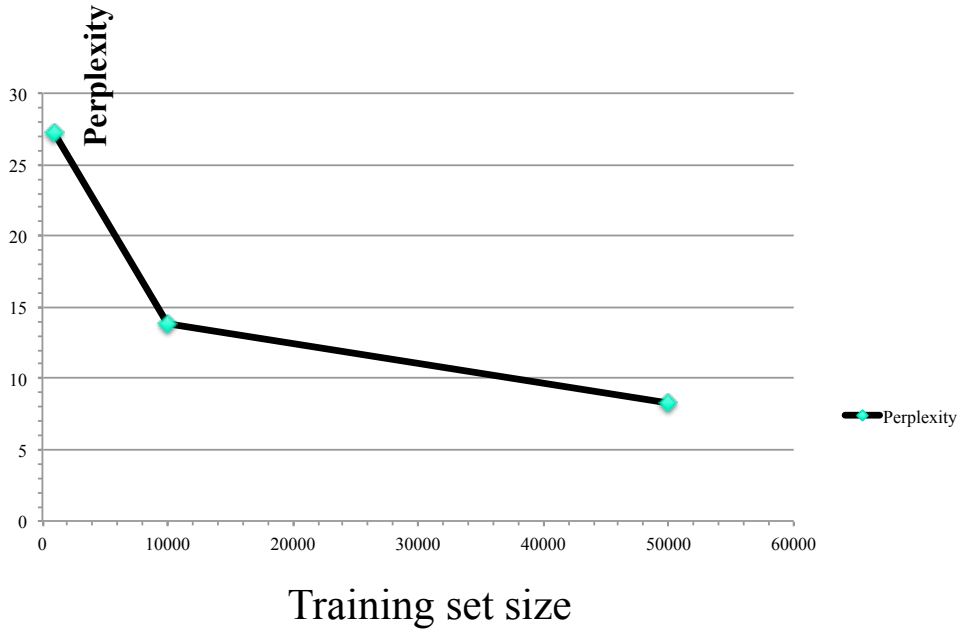
So, if we IMPROVE a language model, the perplexity should go DOWN.

Let's see this at work w/ our trigram English language modeling…with 2 different size training sets, one much larger than the other.

(1) First calculate perplexity using a 1K sample of English words, then testing in on a (small) unseen set.

Perplexity and training corpus size, using Add-1 smoothing: (why didn't we use MLE?)

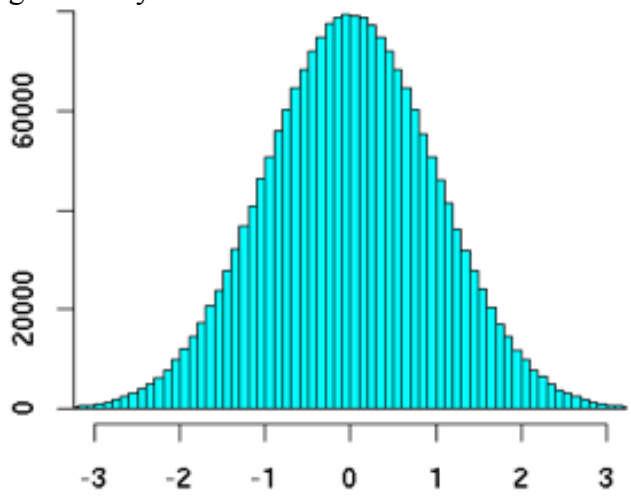
```
1K training set
./prob2.py
add1.0  ../english_spanish/train/en.1K ../english_spanish/dev/english/*/en.500.19
LogProb: -2377.93 Perplexity: 27.1973     NumWords: 499

10K training set:
./prob2.py
add1.0  ../english_spanish/train/en.10K ../english_spanish/dev/english/*/en.500.19
LogProb: -1892.34 Perplexity: 13.8544     NumWords: 499


./prob2.py
add1.0  ../english_spanish/train/en.50K ../english_spanish/dev/english/*/en.500.19
Vocabulary size is 73 types including OOV and EOC
Finished training on 49362 tokens

LogProb: -1518.95 Perplexity: 8.24772     NumWords: 499

1K, 27,2; 5K, 13.9  50K, 8.3
```



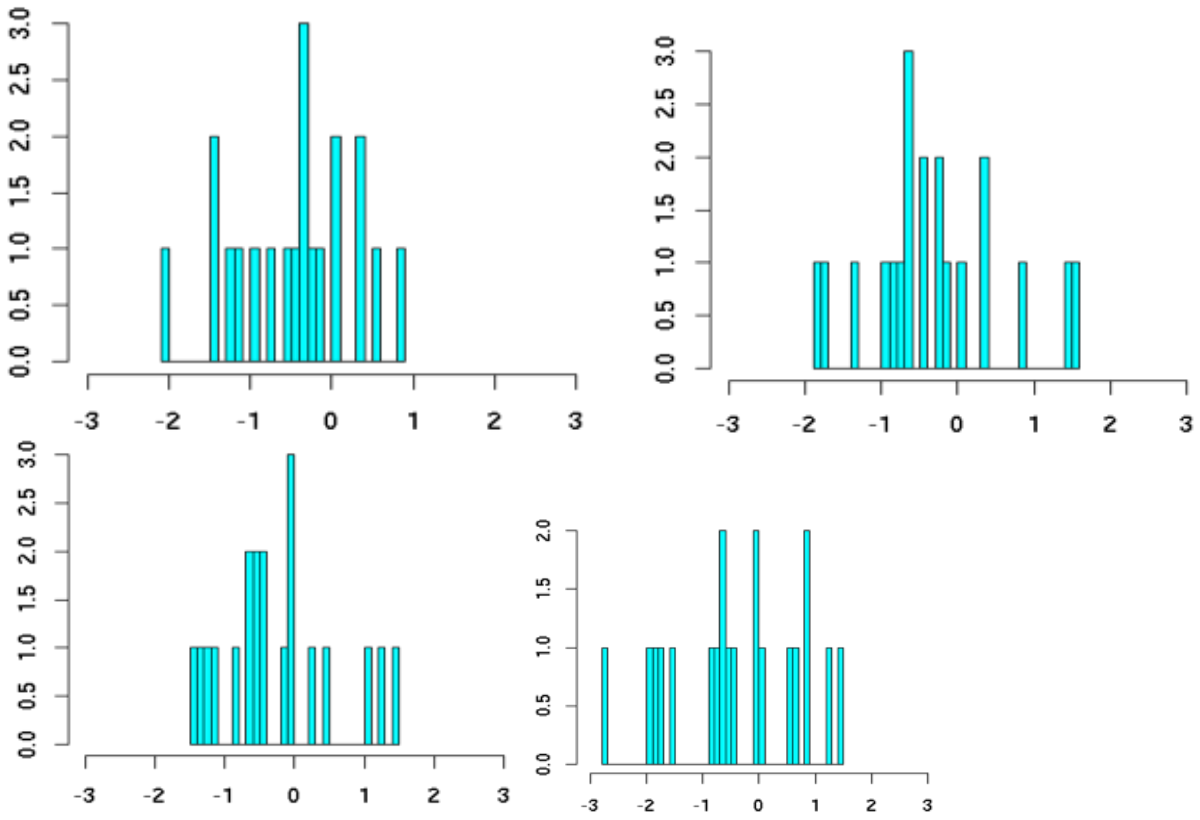
Training set size

III. Smoothing: Never trust a sample under 30!

Smoothing. Here's a picture of what smoothing is doing…what happens when we take 4 different samples of size 20 from this curve? We get this. No single draw looks like a bell curve, but put together they do – variance on individual draw is HIGH



We get this:



So, how do we avoid this problem?

1. The simplest idea: Laplace, 1790: Add-1 smoothing.
(Arose in the context of betting, naturally…)
His solution: add 1 to ALL counts (not just the zero count bigrams)
Q: Why does this 'make sense' (even though it's not good in practice..)
A: it's a kind of 'interpolation' between '0 knowledge' distribution and MLE (though a bad one…) Let's see this for the unigram case

Suppose we have "zero knowledge" we haven't even **looked** at the training data, we don't even know how big the training data is – then what is our **best** estimate of the probability of a single word drawn from a vocabulary of size $|V|$?

A: It is 1/|V|. (The uniform distribution)

Q: What would be the estimate for the probability of a word drawn from this vocabulary if we had *infinite* data?
A: It's the maximum likelihood estimate, ie, the # times the word occurs/total # words in corpus.

In practice, let N be the total length of our corpus in words.

This suggests that we take some weighted average of the two estimates.

$$\lambda D_{Z} + (1-\lambda) D_{MLE} = \lambda\, 1/|V| + (1-\lambda)\, c(w)/N$$

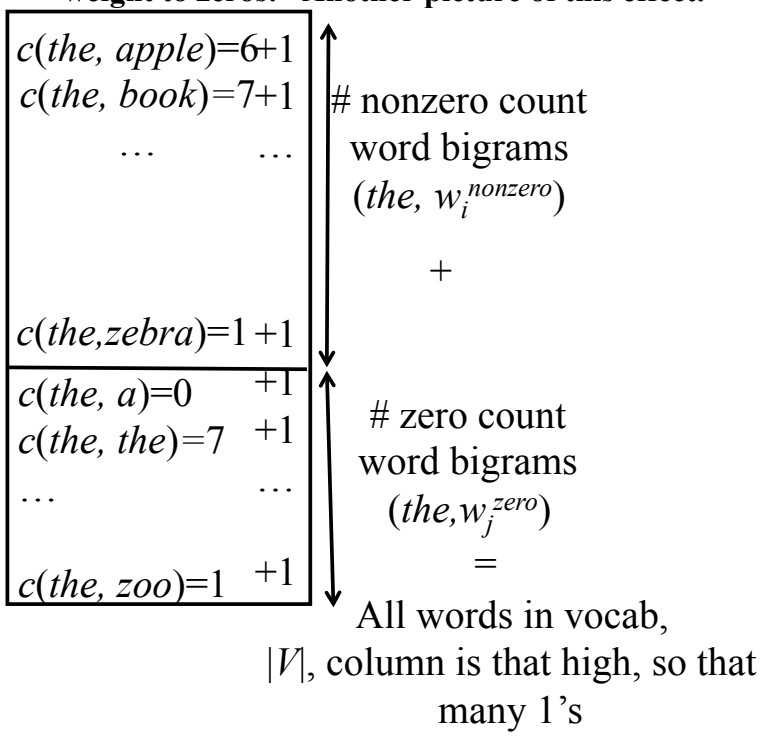
What should lambda be? Let's make lambda equal to the proportion of V out of $|V|+N$, i.e., $|V|/(N+|V|)$. (Laplace pulled this out of thin air…) As N gets very big, then lambda goes to 0, and the zero knowledge term is weighted less and less. Since together the two distributions have to form a legitimate probability distribution, they must add up to one. So the weight for the MLE term must be 1–lambda, or:

$1-|V|/(N+|V|) = [(N+|V|)-|V|]/(N+|V|) = N/(N+|V|)$. Our weighted average is then:

$$\frac{|V|}{|V|+N}\cdot\frac{1}{|V|}+\frac{N}{N+|V|}\cdot\frac{count(w)}{N}=$$
$$\frac{count(w)+1}{N+|V|}$$

That is, we add-1 to the original count, and then add V to the denominator to make sure the probability mass still adds up to 1. (Because there are $|V|$ possible words.)

Let's see how this works with the Berkeley restaurant data. What do you notice? Too much weight to zeros! Another picture of this effect.



So, this was Laplace's answer. *Every* count – not just the missing ones gets at least count 1. Because there are $|V|$ possible words, we must adjust the denominator by this additional factor to make sure it still sums to 1 and forms a legitimate probability distribution. (Below we'll see a method to calculate lambda, instead of just fixing it to be $|V|/(|V|+N)$.)

(Note we can do the same for bigrams, i.e., every bigram at least 1).

The formula here to adjust bigrams is: $p(w_i \mid w_{i-1})$= (1+count(w_{i-1},w_i)+1)/(|*V*|+*count*(w_{i-1})). Do you see why we have to renormalize by |*V*|? Above is the picture of what adding 1 does. We have to add on |*V*| 1's to *every* count. (Both zero count bigrams AND the non-zero count bigrams.)

Q: What's <u>wrong</u> with this correction? At least two things.
A1: it assigns **too much** probability mass to unseen events. (zero counts)

Example. Consider the bigrams (*burnish* | *you*) vs. (*burnish* | *thou*). If the count for both of these is 0, then Add-1 will give both the same probability. But we know that *p*(*you*) > *p*(*thou*). This suggests that what we **really** want to do is to interpolate between the unigram and bigram models, but in a smarter way than what we just did.

A2 (similar): it smooths *all* unseen words (or bigrams…) in the *same* way
So, we must really reduce the count for unknown events to something *less than* 1!

Practical result: Typically, add-1 is *horrible*!

Here is the result of doing add-1 smoothing from the English-Spanish test:

```
python textcat4g.py
add1.0 ../english_spanish/train/en.1K ../english_spanish/train/sp.1K
../english_spanish/dev/english/*/*

108 looked more like english (90.00 %)
12 looked more like spanish (10.00 %)

7 looked more like english (5.88 %)
112 looked more like spanish (94.12 %)

So error rate = 19/239 = 7.9%  with ADD-1 smoothing.
```

Let's see if tweaking lambda helps.

What should we do? Here is the general formulation, where lambda is 1 for the Laplace method:

$$q(u,v,w) = \frac{count(u,v,w) + \lambda}{count(u,v) + \lambda V}$$

General idea: Linear interpolation method.
In Laplace's method, we just set the weight given to the bigram vs. unigram estimates using his |*V*|/(|*V*|+*N*) idea. But that generally won't work well.

Suppose we weight the unigram-bigram estimates differently. 1 is TOO MUCH to add to an unseen event, so let's add something much smaller, like 0.1 or 0.01, etc. So instead of Add-1, use Add-0.1 or Add-0.01. (In general: add-lambda).

Here's the generalized formula, for a trigram:

$$q(w \mid u,v) = \frac{count(u,v,w) + \lambda}{count(u,v) + \lambda V}$$

So, if lambda = 1, this is ADD1.0 (Laplace)
Question: what is this method if lambda = 0?
(Try it – you get divide errors, since most MLE estimates are zero!)

OK, what about finding the 'best' lambda value to add?

Here's what we get with the language ID data. **What data** should we use to 'tweak' lambda? NOT the test data!!! We use what is typically called "development data" – we shouldn't even LOOK at the test data!

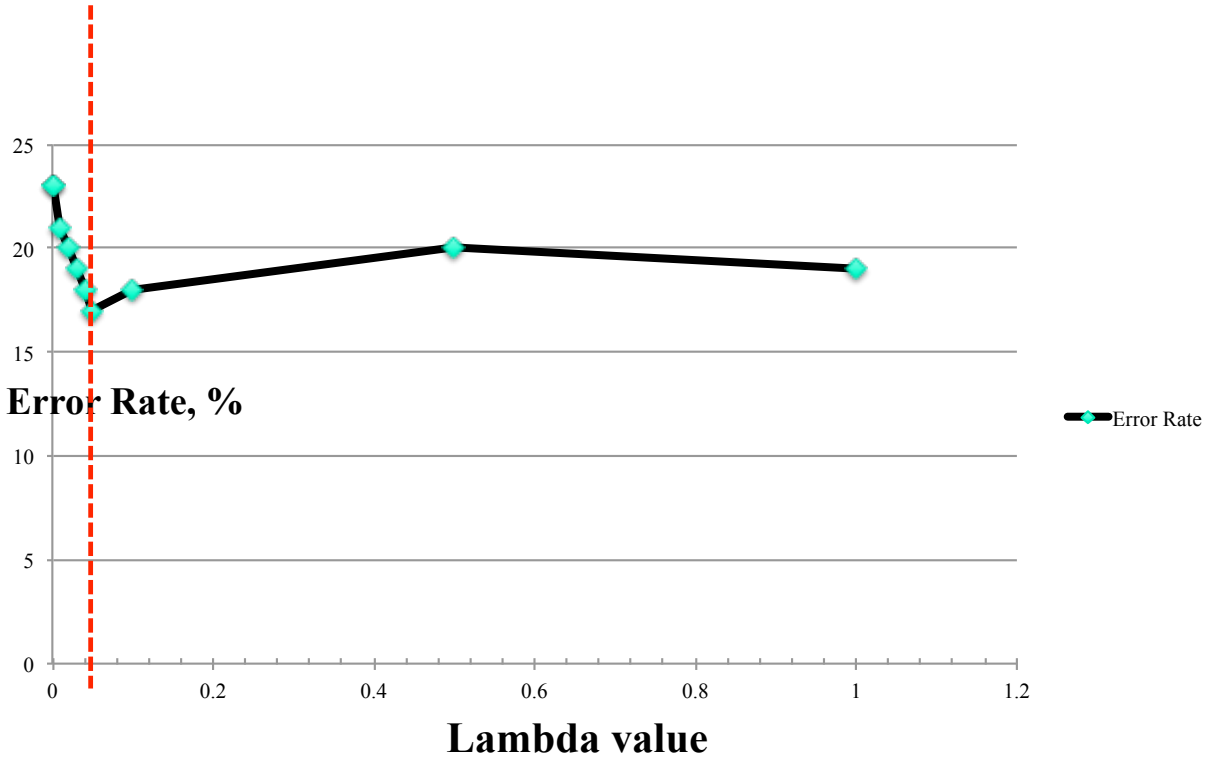
What does add-lambda-0.01 mean? It means that a REAL training data point is worth 100x more than the 'smoothed' not-yet-seen training data point

If we do this for the English-Spanish data, here's what we get. So, the 'best' value for lambda is the one that makes the error rate on the development data smallest.
The numbers look like this:

Lambda value	Errors rate (# misclassified,	English as Spanish or vv)
1.0	19	7.9
0.1	18	7.5
0.05	17	7.1
0.04	18	7.5
0.03	19	7.9
0.02	20	8.4
0.01	21	8.8
0.001	23	9.6
0.0001	23	9.6

Here, the best value is at 0.05, # errors= 17. Above or below that lambda value, the error rate is greater.

But how can we find the optimal lambda in general? NOTE that the curve is concave! (In fact, thisgenerally true…for the 'likelihood' surface.) So we can use a simple iterative method – simple hill-climbing will work. Set the lambda value at a random place and start it off. Take tiny steps. Here's the graph.



I'll leave this as an exercise in simple AI gradient ascent.

Of course, we can **extend** this linear interpolation approach to the case of trigrams. Now we have **three** lambda values (that must add up to 1, as before).

$$q(w_i \mid w_{i-2}, w_{i-1}) = \lambda_1 \times q_{ML}(w_i \mid w_{i-2}, w_{i-1}) + \lambda_2 \times q_{ML}(w_i \mid w_{i-1}) + \lambda_3 \times q_{ML}(w_i)$$

where all the lambdas are greater than or equal to 0, and they add up to 1 (as in the bigram case).

In short, the idea of an **interpolation method** is to use the information about the 'lower order' models all the time – not only when the counts of, say, a trigram are 0, but even when the trigram count is non-zero. We estimate the trigram probability as the weighted sum of the trigram, bigram, and unigram maximum likelihood estimates.

There is a second approach to smoothing, called **backoff.** The idea of **backoff** is simple: if we have **no** information about a trigram estimate, because the count is 0, then we use the bigram estimate; if we have no information about the corresponding bigram, then we use the unigram estimate (just the word itself). Thus, backoff is used to estimate probabilities for higher-order events that have 0 counts, it leaves the lower-order non-zero counts alone. So, 'lower order' n-grams are used when the 'higher order' n-grams have 0 counts.

This suggests the following combination of interpolation and backoff. (There are many other backooff methods, as described in you textbooks.)

Suppose both *w* and *w′* are seen very rarely in the context (*u,v*). These small trigram counts are not reliable, so we'd like to rely mostly on the backed-off bigram estimates to distinguish *w* from *w′*. The problem is that add-lambda treats *all* bigrams the same, when they are not. So let's be smarter than just using a *constant* lambda. What we want is a better estimate here, one that is different for different bigrams. Here is our add-lambda equation again:

$$q(u,v,w) = \frac{count(u,v,w) + \lambda}{count(u,v) + \lambda \cdot |V|}$$

We now replace lambda with a backed-off bigram $q(w|v)$:

$$q(u,v,w) = \frac{count(u,v,w) + \lambda \cdot |V| \cdot q(w \mid v)}{count(u,v) + \lambda \cdot |V|}$$

How is this different from interpolation? We are adding in our knowledge of the bigram *w*|*v* rather than just a 'constant' lambda. That's what we call a "BACK-OFF" term.

What about estimating $q(w,v)$ though? If our count is nonzero, great; otherwise, we can model this, in turn, via an estimate for $q(w)$ (a unigram estimate).

For our estimate of the unigram, we can do this: (multiply out |*V*| x 1/|*V*|)

$$q(w) = \frac{count(w) + \lambda}{N + \lambda |V|} = \frac{count(w) + \lambda |V| \left(\frac{1}{|V|} \right)}{count() + \lambda |V|}$$

Which is just our old add-lambda backoff scheme. Putting this together recursively…

$$q(w \mid v) = \frac{count(v,w) + \lambda |V| \cdot q(w)}{count(v) + \lambda |V|}$$

See how the term added to *count* is now a backed-off unigram estimate for *w,* and as we've just shown, $q(w)$ is just the usual add-lambda formula. Plugging in the estimated value for $q(w)$ recursively we get this:

$$q(w \mid v) = \frac{count(v,w) + \lambda |V| \cdot \left(\frac{count(w) + \lambda}{count() + \lambda |V|} \right)}{count(v) + \lambda |V|}$$

So, one more recursive step up to the trigram, we can get the estimated value for $q(w \mid u,v)$ as follows, from substitution for $q(w|v)$ in our formula:

$$q(u,v,w) = \frac{count(u,v,w) + \lambda |V| \cdot q(w \mid v)}{count(u,v) + \lambda |V|}$$

So putting this all together:

$$q(u,v,w) = \frac{count(u,v,w) + \lambda |V| \cdot \frac{count(v,w) + \lambda |V| \cdot \left(\frac{count(w) + \lambda}{count() + \lambda |V|} \right)}{count(v) + \lambda |V|}}{count(u,v) + \lambda |V|}$$

This looks like a BIG MESS, but it is very easy to do from the 'bottom up' by a computer program!

What's the payoff? This backoff smoother will better model the *other* language (or, e.g., *spam*), because salient features of the other language are often specific unigrams and bigrams. Knowledge of these is now explicitly incorporated. NOTE how this DIFFERS from 'interpolation' – in interpolation, we do a weighted average between *count*(*u,v,w*), *count*(*u,v*), and *count*(*u*); in backoff, we do the weighting between trigrams (as before), but now bigrams of (*v,w*) and unigrams of (*w*) – that's the difference! Can we get a lower error rate that our previous best of 7.1%? In fact, it's pretty hard with this dataset. (More training data helps much more: 50K drops the error to 2.5%.)

Reverence for Reverend Bayes

What else can we do to improve our English-Spanish identification? Have we been making use of all our knowledge? Suppose we know *a priori* that our text will be Spanish, with probability 30%? (Alternatively: if 30% of our email is spam…) How can we make use of this prior knowledge?

Calculating the probabilities with respect to the two models, p_{ENGLISH}(*test-text*) and p_{SPANISH} (*test-text*) makes the assumption that the probability of generating a particular *test-text* after training on English is higher–it makes sense to calculate this as the classification: the *posterior* distribution over the two models *m* given the sample text *t*:

$$p(m = \{English, Spanish\} \mid t)$$

Now we apply Bayes' rule to obtain:

$$p(english \mid t) = \frac{p(english) \cdot p(t \mid english)}{p(t)}$$

and

$$p(spanish \mid t) = \frac{p(spanish) \cdot p(t \mid spanish)}{p(t)}$$

If we now multiply the left-hand side of both equations by $p(t)$ we get:

$$p(english \mid t)p(t) = p(english) \cdot p(t \mid english)$$
$$p(spanish \mid t)p(t) = p(spanish) \cdot p(t \mid spanish)$$

But the lefthand sides are just the following, by the definition of condition probability:

$$p(english, t) = p(english) \cdot p(t \mid english)$$

$$p(english,t) = p(english) \cdot p(t \mid english)$$
$$p(spanish,t) = p(spanish) \cdot p(t \mid spanish)$$

Now we can apply our prior probability knowledge of English vs. Spanish p(english)=2/3; p(spanish)=1/3. So we plug those values into the above get this:

$$p(spanish,t) = 1/3 \cdot p(t \mid spanish)$$
$$p(english,t) = 2/3 \cdot p(t \mid english)$$

In other words, the probability of a text being English is scaled by a 2:1 ratio over its being Spanish.

This is what we would expect from the prior odds ratio. It has the effect of increasing a bias in favor of English. (Texts formerly classified as Spanish might now be called English, but texts that were previously English will stay that way.) The overall effect is to reduce the error rate slightly on our example data.