600.465 — Natural Language Processing
Assignment 3: Smoothed Language Modeling, Conditional Log-Linear Modeling, and their Applications

Prof. J. Eisner — Fall 2011
Due date: Wednesday 5 October, 2 pm

This assignment will try to convince you that statistical models—even simplistic and linguistically stupid ones like n-gram models—can be very useful, provided their parameters are estimated carefully. Almost all speech recognition systems use some form of trigram model. While trigram models can be enhanced in various ways, it's surprisingly hard to improve much on their performance. (And alternative approaches that *don't* look at the trigrams just do worse.)

In addition, you will get some experience in running corpus experiments over training, development, and test sets.

Why is this assignment absurdly long? Because the assignments are your most important reading for the class. They're shorter and more interactive than a textbook. :-) The textbook readings are usually quite helpful, and you should have at least skimmed the readings on smoothing by now, but it is not mandatory that you know them in full detail.

Programming language: You may work in any language that you like. However, we will give you some useful code as a starting point.[1] This code is currently provided in Java and Python. There are also versions in some other languages, but they are not fully updated for this year's version of the assignment. If you want to use another language, then feel free to translate (or ignore?) our short code before continuing with the assignment. Please send your translation to the course staff so that we can make it available to the whole class.

On getting programming help: Since this is a 400-level NLP class, not a programming class, I don't want you wasting time on low-level issues like how to handle I/O or hash tables of arrays. If you are doing so, then by all means seek help from someone who knows the language better! Your responsibility is the NLP stuff—you do have to design, write, and debug the interesting code and data structures **on your own**. But I don't consider it cheating if another hacker (or the TA) helps you with your I/O routines or language syntax or compiler warning messages. These aren't Interesting™.

[1] It counts word n-grams in a corpus, using hash tables, and uses the counts to calculate simple probability estimates. We also supply a method that calls an optimization library to maximize a function.

How to hand in your work: Same procedure as before. You must test that your programs run with no problems on the `ugrad` machines (named `ugrad1`–`ugrad24`) before submitting them. You probably want to develop them there in the first place, since that's where the corpora are stored. (However, in principle you could copy the corpora elsewhere for your convenience.)

Again, besides the comments you embed in your source files, put all other notes, documentation, and answers to questions in a `README` file. The file should be editable so that we can insert comments and give it back to you. For this reason, we strongly prefer a plain ASCII file `README`, or a LaTeX file `README.tex` (in which case please also submit `README.pdf`). If you must use a word processor, please save as `README.rtf` in the portable, non-proprietary RTF format.

If your programs are in some language other than the ones we used, or if we need to know something special about how to compile or run them, please explain in a plain ASCII file `HOW-TO`.

Your source files, the `README` file, the `HOW-TO` file, and anything else you are submitting will all need to be placed in a single submission directory.

Remember from the previous assignment that a **language model** estimates *the probability of any string* $\vec{w}$. In a trigram model, we use the chain rule and backoff to assume that

$$p(\vec{w}) = \prod_{i=1}^{N} p(w_i \mid w_{i-2}, w_{i-1})$$

with start and end symbols `<s>` and `</s>` handled as in the previous assignment.

You now know enough about probability to build and use some trigram language models. You will experiment with different types of smoothing. **Your starting point is the sample program `fileprob`.**

But first, let's talk about "out of vocabulary" words. All the smoothing methods assume a finite vocabulary, so that they can easily allocate probability to all the words. But is this assumption justified? Aren't there *infinitely* many potential words of English that might show up in a test corpus (like "xyzzy" and "JacrobinsteinIndustries" and "fruitylicious")?

Yes there are ... so we will *force* the vocabulary to be finite by a standard trick. Choose some fixed, finite vocabulary at the start. Then add one special symbol OOV that represents all other words. When you subsequently see these "out of vocabulary" (OOV) words during training or test, you should regard them as nothing more than variant spellings of the OOV symbol.

For example, when you are considering the test sentence

```
i saw snuffleupagus on the tv
```

what you will actually compute is the probability of

`i saw` OOV `on the tv`

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

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

Admittedly, this total probability is higher than the probability of the *particular* sentence involving `snuffleupagus`. But in most of this assignment, we only wish to compare the probability of the snuffleupagus sentence under different models. Replacing `snuffleupagus` with OOV raises the sentence's probability under all the models at once, so it need not invalidate the comparison.[2]

We do have to make sure that if `snuffleupagus` is regarded as OOV by one model, then it is regarded as OOV by all the other models, too. It's not appropriate to compare p_{model1}(`i saw` OOV `on the tv`) with p_{model2}(`i saw snuffleupagus on the tv`), since the former is actually the total probability of many sentences, and so will tend to be larger.

So all the models must have the *same* finite vocabulary, chosen up front. In principle, this shared vocabulary could be *any* list of words that you pick by *any* means, perhaps using some external dictionary.

Though the context "OOV `on`" never appeared in the training corpus, the smoothing method is required to give a reasonable value anyway to $p(\texttt{the} \mid \text{OOV}, \texttt{on})$, for example by backing off to $p(\texttt{the} \mid \texttt{on})$.

Similarly, the smoothing method must give a reasonable (non-zero) probability to $p(\text{OOV} \mid \texttt{i}, \texttt{saw})$. Because we're merging all out-of-vocabulary words into a single word OOV, we avoid having to decide how to split this probability among them.

How should you choose the vocabulary? For this assignment, simply take it to be the set of words that appeared *at least once* anywhere in *training* data. Then augment this set with the OOV symbol. Let V be the size of the resulting set (including OOV).

Thanks to this (common) way of choosing the vocabulary, you won't need special code to detect that `snuffleupagus` is OOV and hence look up the count of "`i saw` OOV". Simply look up the count of "`I saw snuffleupagus`" in the usual way, and you'll get the same answer, zero, since "`snuffleupagus`" never appeared in the training corpus. (He's shy.)

But to help you understand/debug your programs, we have grafted brackets onto all out-of-vocabulary words in *one* of the datasets (the `speech` directory). This lets you identify the OOV words at a glance. In this particular dataset, you will actually see the test sentence

`i saw [snuffleupagus] on the tv`

and therefore look up the count of the trigram "`i saw [snuffleupagus]`," which is 0.[3] You can experiment on that dataset, starting with the following problem.

---

[2] Problem 12 explores a prettier approach that may also work better for text categorization.

[3] Your program does not have to do any special handling for the brackets. Although as it happens, the count of "`i saw snuffleupagus`" must also be 0, as both `snuffleupagus` and `[snuffleupagus]` are OOV.

1. The sample program is on the ugrad machines in the directory

   `/usr/local/data/cs465/hw-lm/code`

   There are subdirectories corresponding to different programming languages. Choose one as you like. Or, port to a new language (see page 1 of the assignment).

   Each language-specific subdirectory contains an `INSTRUCTIONS` file explaining how to get the program running. Those instructions will let you automatically compute the $\log_2$-probability of three sample files (`sample1`, `sample2`, `sample3`). Try it![4]

   Next, you should spend a little while looking at those sample files yourself, and in general, browsing around the `/usr/local/data/cs465/hw-lm` directory to see what's there. There are corpora for three tasks: language identification, spam detection, and speech recognition. Each corpus has already been divided into training, development ("held-out"), and test data, and also has a `README` file that you should look at.

   If a language model is built from the `switchboard-small` corpus, using add-0.01 smoothing, what is the model's *perplexity per word* on each of the three sample files? (You can compute this from the $\log_2$-probability that `fileprob` prints out, as discussed in class and in your textbook. Use the command `wc -w` on a file to find out how many words it contains.)

   What happens to the $\log_2$-probabilities and perplexities if you train instead on the larger `switchboard` corpus? Why?

2. Modify `fileprob` to obtain a new program `textcat` that does text categorization. See the `INSTRUCTIONS` file for programming-language-specific directions about which files to copy, alter, and submit for this problem.

   `textcat` should be run from the command line exactly like `fileprob`, except that the command line should specify *two* training corpora rather than 1: *train1* and *train2*.

   `textcat` should classify each file f listed on the command line: that is, it should print the name of the training corpus (*train1* or *train2*) that yields the higher value of $p(f)$. Finally, it should summarize by printing the percentage of files classified each way.

   Sample input (please allow this format; `gen` and `spam` are the training corpora, corresponding to "genuine" and spam emails):

[4] There is one little problem with our setup. To simplify the command-line syntax, I've assumed that the training corpus comes in a *single* big file. That means that there is only one `<s>` and one `</s>` in the whole training corpus—whereas `<s>` and one `</s>` are much more common in test data. This is a case of *domain mismatch*, where the training data is somewhat unlike the test data. Domain mismatch is a common source of errors in applied NLP. In this case, the only practical implication is that your language models won't do a very good job of modeling what tends to happen at the very start or very end of a file.

```
textcat add1 gen spam foo.txt bar.txt baz.txt
```

Sample output (please use this format; send any tracing output to the stderr stream):

```
spam    foo.txt
spam    bar.txt
gen     baz.txt
1 looked more like gen (33.33%)
2 looked more like spam (66.67%)
```

Use add1 smoothing as shown above.

As discussed earlier, both language models built by `textcat` should use the same finite vocabulary. Define this vocabulary to consist of all words that appeared in *either* training corpus (i.e., the union of two sets), plus OOV. Your model doesn't actually need to store the set of words in the vocabulary, but it does need to know its size V, because the add-1 smoothing method estimates $p(z \mid xy)$ as $\frac{c(xyz)+1}{c(xy)+V}$. We've provided code to find V for you—see the `INSTRUCTIONS` file for details.

3. In this question, you will evaluate your `textcat` program on ONE of two problems. You can do either language identification (the `english_spanish` directory) or else spam detection (the `gen_spam` directory). Have a look at the development data in both directories to see which one floats your boat. **(Don't peek at the test data!)**

   (It may be convenient to use symbolic links to avoid typing long filenames. E.g., `ln -s /usr/local/data/cs465/hw-lm/english_spanish/train ~/estrain` will create a subdirectory `estrain` under your home directory; this subdirectory is really just a shortcut to the official training directory.)

   Run `textcat` on all the development data for your chosen problem:

   - For the language ID problem, classify the files `english_spanish/dev/english/*/*` using the training corpora `en.1K` and `sp.1K`.
     Then classify `english_spanish/dev/spanish/*/*` similarly. Note that for this corpus, the "words" are actually letters.
   - Or, for the spam detection problem, classify the files `gen_spam/dev/gen/*` using the training corpora `gen` and `spam`.
     Then classify `gen_spam/dev/spam/*` similarly.

   From the results, you should be able to compute a total error rate for the technique: that is, what percentage of the test files were classified incorrectly?

   Now try add-λ smoothing for $\lambda \neq 1$. Experiment by hand with different values of $\lambda > 0$. (You'll be asked to discuss in 4b why $\lambda = 0$ probably won't work well.)

(a) What is the lowest error rate you could achieve on development data?

(b) What value of λ gave you that rate? Call this λ_0: for simplicity, you will use $\lambda = \lambda_0$ throughout the rest of this assignment. (An alternative would be to use the λ that minimizes the *cross-entropy* of development data.)

(c) Using add-λ_0 smoothing, what is the error rate on test data? (Before now, you should not have done anything with the test data!)

(d) Each of the development and test files has a length. For language ID, the length in characters is given by the directory name and is also embedded in the filename (as the first number). For spam detection, the length in words is embedded in the filename (as the first number).

Using your results from problem 3a, come up with some way to quantify or graph the relation between development file length and classification accuracy. (Feel free to use Piazza to discuss how to do this.) Write up your results.

You may find the `xgraph` utility useful; it is very easy to use. Type `man xgraph` for documentation. To include a graph in your writeup, just give us instructions about how to see it on the ugrad machines (e.g., `xgraph < mydatafile` or `gv mygraph.ps`). A more powerful choice would be gnuplot: see http://comp.ling.utexas.edu/wiki/doku.php/tips_and_tricks#graphing.

(e) Now try increasing the amount of *training* data. Compute the overall error rate on development data for training sets of different sizes. Graph the training size versus classification accuracy.

- For the language ID problem, use training corpora of 6 different sizes: `en.1K` vs. `sp.1K` (1000 characters each); `en.2K` vs. `sp.2K` (2000 characters each); and similarly for `5K`, `10K`, `20K`, and `50K`.
- Or, for the spam detection problem, use training corpora of 4 different sizes: `gen` vs. `spam`; `gen-times2` vs. `spam-times2` (twice as much training data); and similarly for ...`-times4` and ...`-times8`.

Here are the smoothing techniques we'll consider, writing $\hat{p}$ for our *smoothed estimate* of p:

uniform distribution (UNIFORM) $\hat{p}(z \mid xy)$ is the same for every xyz; namely,

$$\hat{p}(z \mid xy) = 1/V$$

where V is the size of the vocabulary *including* OOV.

add-λ (ADDL) Add a constant $\lambda \geq 0$ to every trigram count $c(xyz)$:

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda}{c(xy) + \lambda V}$$

where V is defined as above. (Observe that $\lambda = 1$ gives the add-one estimate. And $\lambda = 0$ gives the naive historical estimate $c(xyz)/c(xy)$.)

add-λ backoff (BACKOFF_ADDL) Suppose both z and z' have rarely been seen in context xy. These small trigram counts are unreliable, so we'd like to rely largely on backed-off bigram estimates to distinguish z from z':

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda V \cdot \hat{p}(z \mid y)}{c(xy) + \lambda V}$$

where $\hat{p}(z \mid y)$ is a backed-off bigram estimate, which is estimated recursively by a similar formula. (If $\hat{p}(z \mid y)$ were the UNIFORM estimate $1/V$ instead, this scheme would be identical to ADDL.)

So the formula for $\hat{p}(z \mid xy)$ backs off to $\hat{p}(z \mid y)$, whose formula backs off to $\hat{p}(z)$, whose formula backs off to ... what??

Witten-Bell backoff (BACKOFF_WB) As mentioned in class, this is a backoff scheme where we explicitly reduce ("discount") the probabilities of things we've seen, and divide up the resulting probability mass among only the things we *haven't* seen.

$$\begin{aligned}
\hat{p}(z \mid xy) &= \begin{cases} p_{\text{disc}}(z \mid xy) & \text{if } c(xyz) > 0 \\ \alpha(xy)\hat{p}(z \mid y) & \text{otherwise} \end{cases} \\
\hat{p}(z \mid y) &= \begin{cases} p_{\text{disc}}(z \mid y) & \text{if } c(yz) > 0 \\ \alpha(y)\hat{p}(z) & \text{otherwise} \end{cases} \\
\hat{p}(z) &= \begin{cases} p_{\text{disc}}(z) & \text{if } c(z) > 0 \\ \alpha() & \text{otherwise} \end{cases}
\end{aligned}$$

Some new notation appears in the above formulas. The *discounted probabilities* p_{disc} are defined by using the Witten-Bell discounting technique:

$$\begin{aligned}
p_{\text{disc}}(z \mid xy) &= \frac{c(xyz)}{c(xy) + T(xy)} \\
p_{\text{disc}}(z \mid y) &= \frac{c(yz)}{c(y) + T(y)} \\
p_{\text{disc}}(z) &= \frac{c(z)}{c() + T()}
\end{aligned}$$

where

- $T(xy)$ is the number of different word types z that have been observed to follow the context xy
- $T(y)$ is the number of different word types z that have been observed to follow the context y
- $T()$ is the number of different word types z that have been observed at all (this is the same as V except that it doesn't include OOV)
- $c()$ is the number of tokens in the training corpus, otherwise known as N.

Given all the above definitions, the values $\alpha(xy)$, $\alpha(y)$, and $\alpha()$ will be chosen so as to ensure that $\sum_z \hat{p}(z \mid xy) = 1$, $\sum_z \hat{p}(z \mid y) = 1$, and $\sum_z \hat{p}(z) = 1$, respectively.

conditional log-linear modeling (LOGLIN) We'll follow the very useful general scheme given in lecture for conditional log-linear modeling. It will be a bit trickier here, so make sure you figure out how the specific formulas below are an instantiation of that general scheme.[5] We estimate

$$\hat{p}(z \mid xy) = \frac{u(xyz)}{Z(xy)}$$

where we define

$$u(xyz) \stackrel{\text{def}}{=} \exp \sum_k \theta_k \cdot f_k(xyz) = \exp(\vec{\theta} \cdot \vec{f}(xyz)) \qquad \text{(“unnormalized probability”)}$$

$$Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz) \qquad \text{(“normalizing constant” that ensures } \textstyle\sum_z \hat{p}(z \mid xy) = 1)$$

Here k ranges over features, $\vec{f}(xyz)$ is the vector of features associated with xyz, and $\vec{\theta}$ is a vector of feature *weights*. To take a simple example, suppose we had a model with only two features (so $k \in \{1, 2\}$):

$$f_1(xyz) = \begin{cases} 1 \text{ if } xyz = \texttt{qrs} \\ 0 \text{ otherwise} \end{cases} \qquad f_2(xyz) = \begin{cases} 1 \text{ if } yz = \texttt{rs} \\ 0 \text{ otherwise} \end{cases}$$

What conditional probability $\hat{p}(\texttt{s} \mid \texttt{er})$ is assigned to the trigram ers? We have $\vec{f}(\texttt{ers}) = (f_1(\texttt{ers}), f_2(\texttt{ers})) = (0, 1)$. So if $\vec{\theta} = (\theta_1, \theta_2) = (-0.3, 1.2)$, then the unnormalized probability $u(\texttt{ers}) = \exp(-0.3 \cdot 0 + 1.2 \cdot 1) = \exp 1.2 \approx 3.32$. Of course this is greater than 1, but we normalize the probabilities to sum to 1: $\hat{p}(\texttt{s} \mid \texttt{er}) \stackrel{\text{def}}{=} u(\texttt{ers})/Z(\texttt{er}) = u(\texttt{ers})/(u(\texttt{era}) + u(\texttt{erb}) + \cdots + u(\texttt{erz}))$.

[5] Unfortunately, the notation in lecture also used the variable names x and y, but to mean something different than they mean in this assignment. The notation in lecture is standard in machine learning.

Of course, our real model will have a lot more than two features! However, for a given trigram such as `ers`, almost all of them will be 0, so computing $u(\texttt{ers})$ will still be very fast: in the summation $\sum_k \theta_k \cdot f_k(xyz)$, you can loop over only those few k for which $f_k(xyz) \neq 0$. These are called the features of xyz, or the features that "fire" on xyz. (Computing $Z(\texttt{er})$ quickly will require a bit more ingenuity.)

$\vec{\theta}$ is incredibly important: the way we pick it will determine all the probabilities in the model! We'll set $\vec{\theta}$ to the most likely weight vector given the training corpus $\vec{w}$. That is, we set $\vec{\theta}$ so as to maximize $p(\vec{\theta} \mid \vec{w})$. Equivalently, we set $\vec{\theta}$ to maximize[6]

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \log \left(\underbrace{p(\vec{w} \mid \vec{\theta})}_{\textit{likelihood}} \cdot \underbrace{p(\vec{\theta})}_{\textit{prior}} \right) = \underbrace{\left(\sum_{i=1}^{N} \log \hat{p}(w_i \mid w_{i-2}\, w_{i-1}) \right)}_{\textit{log likelihood}} - \underbrace{\left(\frac{1}{2\sigma^2} \sum_k \theta_k^2 \right)}_{\textit{log of a certain prior}}$$

Both terms above depend on $\vec{\theta}$. Notice that the $\sum_{i=1}^{N} \cdots$ term is just the log-probability of the training corpus $\vec{w}$. But it can also be regarded as an instance of the general approach to conditional log-linear modeling that we saw in class: our training data consists of N events $w_1, \ldots, w_N$ in context, and we want the product of their contextual probabilities to be large. The above expression is equivalent to

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \left(\sum_{xyz} c(xyz) \log \hat{p}(z \mid xy) \right) - \left(\frac{1}{2\sigma^2} \sum_k \theta_k^2 \right)$$

which uses the $c(xyz)$ notation like the rest of this assignment. In this version, $\sum_{xyz}$ loops over all $O(V^3)$ possible trigram *types* xyz, but it counts each type in proportion to its number of observed *tokens*, $c(xyz)$. So it's equivalent to the previous $\sum_{i=1}^{N}$ version that just loops over the N trigram tokens one by one. You might think $\sum_{xyz}$ would be much slower, since V^3 is enormous. But actually it is at least as fast, since you only have to loop over the $\leq N$ *non-zero* summands—those for which $c(xyz) \neq 0$.

The "regularization term" encourages $\vec{\theta}$ to stay close to zero, because the maximization objective is $-\frac{1}{2\sigma^2} \sum_k \theta_k^2$ becomes quite negative for "large" values of $\vec{\theta}$. As a result, it is such $\vec{\theta}$ result, the maximization procedure will tend to pick a $\vec{\theta}$ that is relatively close to 0.[7] The constant $\sigma^2 > 0$ controls the amount of smoothing, just as λ does in the previous method. Like λ, your σ^2 can be calibrated on development

[6] Why is this equivalent? We can increase $p(\vec{\theta} \mid \vec{w})$ by increasing $p(\vec{w} \mid \vec{\theta}) \cdot p(\vec{\theta})$ (which is proportional to it, by Bayes' Theorem). And we can increase any positive quantity Q by increasing $\log Q$, since larger numbers have larger logs. We prefer maximizing the log to avoid underflow; also, because the derivatives of the log happen to have a simple form (the log sort of cancels out the exp in the definition of $u(xyz)$).

[7] As noted above, it can be interpreted as $\log p(\vec{\theta})$ for a certain choice of prior distribution $p(\theta)$ that says that $\vec{\theta}$ is probably close to 0. A drawing of this prior distribution looks like a bell curve in k dimensions,

data. Smaller values of σ^2 lead to more smoothing (see why?). Smaller N also leads to more smoothing (see why?).

For this assignment, we will employ a "binary" feature (i.e., a feature whose value can be 0 or 1) for *each trigram, bigram, or unigram that appears in training data*. Thus, up to 3 features will "fire" on the trigram xyz: namely xyz, $?yz$, and $??z$. This means that $f_k(xyz) = 1$ for at most 3 features k, while $f_k(xyz) = 0$ for all other features k.

Note: Numerous other smoothing schemes exist. In past years, for example, our course assignments have used Katz backoff with Good-Turing discounting. (We discussed Good-Turing in class: it is attractive but a bit tricky in practice, and has to be combined with backoff.) The most popular scheme nowadays is something called modified Kneser-Ney, or a more principled Bayesian formulation of it based on the hierarchical Pitman-Yor process.

Remember: While these techniques are effective, a really good language model would do more than just smooth n-gram probabilities well. To predict a word sequence, it would go beyond the whole n-gram family to consider syntax, semantics, and topic. This is an active area of research.

---

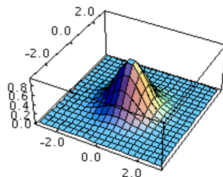
4. (a) Above, V is carefully defined to include OOV. So if you saw 19,999 different word types in training data, then $V = 20,000$. What would go wrong with the UNIFORM estimate if you mistakenly took $V = 19,999$? What would go wrong with the ADDL estimate?

   (b) What would go wrong with the ADDL estimate if we set $\lambda = 0$? (Remark: This naive historical estimate is commonly called the *maximum-likelihood estimate*, because it maximizes the probability of the training corpus.)

   (c) If $c(xyz) = c(xyz') = 0$, than what are the BACKOFF_ADDL estimates of $\hat{p}(z \mid xy)$ and $\hat{p}(z' \mid xy)$? What are they if $c(xyz) = c(xyz') = 1$?

   (d) In the BACKOFF_ADDL scheme, how does increasing λ affect the probability estimates? (Think about your answer to the previous question.)

---

with most of the probability near the origin: 

It's called a spherical Gaussian with mean 0 and variance σ^2. This choice explicitly defines the prior probability of $\vec{\theta}$ to be proportional to $\exp -\frac{1}{2\sigma^2}\sum_k \theta_k^2$. Clearly, this is just a convenient prior assumption about where the true value of $\vec{\theta}$ is most likely to fall. (Similarly, add-λ smoothing (even with backoff) can be derived from a certain prior assumption about the true probabilities $p(z \mid xy)$. Ask me if you're curious.)

5. The code provided to you implements some smoothing techniques, but others are left for you to implement – currently they just trigger error messages. Let's fix that!

   (a) Implement add-λ smoothing with backoff. See the `INSTRUCTIONS` file for language-specific instructions about which files to modify and submit.

   This should be just a few lines of code. You will only need to understand how to look up counts in the hash tables. Just study how the existing methods do it.

   Hint: So $\hat{p}(z \mid xy)$ should back off to $\hat{p}(z \mid y)$, which should back off to $\hat{p}(z)$, which backs off to ... what???

   (b) How does switching to this smoothing method (with $\lambda = \lambda_0$) affect the cross-entropies you found in question 1, and the text categorization error rate you found in question 3c?

   (c) *Extra credit:* Use development data to find a λ_1 that works better than λ_0 with this new smoothing method. How much better does it do on test data? Is λ_1 bigger or smaller than λ_0? Why?

6. Conditional log-linear models are used to predict all kinds of things in NLP and elsewhere in machine learning. They are very flexible because you can define feature functions f_k that are linguistically appropriate for the kind of prediction you're making.

   Such a model claims that for all outcomes and contexts,

$$p(\text{outcome} \mid \text{context}) = \frac{1}{Z(\text{context})} \exp \sum_k \theta_k f_k(\text{context}, \text{outcome})$$

   for some value of the parameter vector $\vec{\theta}$.[8] By estimating $\vec{\theta}$ from training data, we obtain an estimated probability $\hat{p}(\text{outcome} \mid \text{context})$. This might be used to make a prediction directly, or might be combined with other probabilities in a larger model such as a language model.

   > (Traditionally the context and outcome are called x and y, respectively. We used this $p(y \mid x)$ notation in lecture. Unfortunately, we're already using x and y in this assignment to refer to context words in the case of language modeling.)

   (a) Consider a particular fixed context. Suppose you increase your estimate of θ_5. What happens to your estimated conditional probability of outcomes for which $f_5(\text{context}, \text{outcome}) = 1$?

   (You may assume that f_5 is a binary feature, so it always returns 0 or 1.)

[8] And for appropriate values of $Z(\text{context})$. These Z values are dictated by the choice of $\vec{\theta}$, since they must be chosen so that $\sum_{\text{outcome}} p(\text{outcome} \mid \text{context}) = 1$.

(b) And what happens to your estimated conditional probability of outcomes for which $f_5(\text{context}, \text{outcome}) = 0$? (*Hint:* Remember that probabilities sum to 1.)

(c) What is the effect of raising θ_5 if for this context, $f_5(\text{context}, \text{outcome}) = 1$ for *all* outcomes? In other words, θ_5 does not *distinguish* among different outcomes in this context.

For each of the following tasks, imagine what features might actually be useful. Be thoughtful; be creative! To illustrate your ideas, give a few example feature functions f_k such that $f_k(\text{context}, \text{prediction})$ might reasonably be $\neq 0$ for the particular example probability that is given.

(d) Horse racing: p(winner = `Revere` | knowledge = it rained last night, Revere has an injured hoof, Epitaph's great-grandfather is Equipoise, . . .).

(*Hint:* Some feature functions that fire in this winner = `Revere` example may *also* be able to fire for (e.g.) winner = `Valentine`, either in this context or in other contexts.)

(*Hint:* Remember from part (b) that this probability may be affected even by features that do *not* fire in this case.)

(e) Text categorization *without* language modeling: p(category = `spam` | text = x)

(*Hint:* Have you personally noticed any interesting features of spam messages?)

(*Hint:* Be careful: remember part (c).)

(f) Noisy channel errors: p(character you typed = `d` | character you intended = `f`)

(*Hint:* Keyboard layout might be relevant.)

(g) Probability that `randsent` selects a specific syntactic rewrite rule such as `S` → `NP VP`, when there are too many rules to define these probabilities one by one: p(right-hand-side = `NP[num=plural,head=Yoda] VP[num=plural,head=mumble]` | left-hand-side = `S[head=mumble]`)

(*Hint:* The most important question is whether the pattern `S` → `NP VP` is likely overall. However, this probability might be adjusted by the "fine-grained" properties—whether we are using a singular or plural version of this rule, whether we have number agreement, whether `Yoda` is a common head word for an `NP` and a good subject for `mumble`, etc.)

(h) Web search: p(most relevant page = `http://www.example.com` | query = q)

(i) Translation: p(English sentence = `I love you` | French sentence = `Je t'aime`)

Lots of great software is available for log-linear modeling. It is included in some software libraries for statistics or NLP. For standalone use, I recommend MegaM. You just construct a text file that has one line per training example: each line lists

the names of the features that fire on that example. Given this training file, MegaM can efficiently estimate $\vec{\theta}$ and make predictions.

However, there are cases in NLP where the generic software is *not* efficient. Translation (above) is one such example: computing $Z(\texttt{Je t'aime})$ requires summing over many English translations. The outcome space here is either infinite (all English sentences) or specific to this context (a set of plausible English translations of `Je t'aime`). So while translation systems definitely use log-linear models, they can't rely on MegaM.

Unfortunately, language modeling is another case where the generic software is not efficient. That's why you can't just use MegaM for this assignment—you'll have to write some annoying code that takes advantage of the structure of our particular model of $p(z \mid xy)$, which was given on page 8. Consider that model:

(j) What is the outcome space and how big is it?

(k) Amos says that we will have to learn 3 feature weights (unigram, bigram, trigram). Bess says no, we'll have to learn $O(V^3)$ feature weights. They're both wrong. Why? About how many feature weights will we have to learn?

(l) What other features might be useful for modeling $p(z \mid xy)$, even though we're not considering them in this assignment?

7. Our model of $p(z \mid xy)$ has a large outcome space. We are trying to predict a probability distribution over a large vocabulary of V words, not just over a small set of decisions like $\{\texttt{gen}, \texttt{spam}\}$.

   As noted above, this leads to a computational slowdown in finding the *normalizing constant* Z. Recall from page 8 that $\hat{p}(z \mid xy) \stackrel{\text{def}}{=} u(xyz)/Z(xy)$. The numerator is fast to compute, since xyz has at most 3 features in our setup. The denominator, however, requires a sum over the whole vocabulary, since $Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz)$. That requires $O(V)$ time: a lot of work to compute a single probability.

   At test time, we may be called upon to compute $\hat{p}(z \mid xy)$ for *any* trigram xyz, so we will need $Z(xy)$ for *any* xy. If we had to precompute $Z(xy)$ for each of the V^2 possible bigram contexts xy, the total time would be $O(V^3)$, which is very large.

   However, most of these V^3 trigram types have never been observed ($N \ll V^3$). So perhaps there are some computational shortcuts? At most $c(xy)$ of the V summands in $Z(xy) = \sum_z u(xyz)$ correspond to observed trigrams. So perhaps there is a fast way to compute this sum. Particularly when $c(xy) = 0$, which is the usual case (since at most N of the V^2 possible bigram types were observed).

   The crucial insight is that when $c(xyz) = 0$, which is true for **most** trigram types xyz, then no **trigram** feature will fire on xyz. (Recall that our trigram features

correspond to *observed* trigrams only.[9]) Thus, $u(xyz)$ is determined **entirely** by the unigram and bigram features that fire on yz. There are many trigrams like xyz: any other unobserved trigram of the form $x'yz$ has identical features to xyz, so $u(x'yz)$ must have the same value as $u(xyz)$. It is convenient to write this common value as just $u(yz)$.[10]

We can now write

$$Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz) = \sum_z u(yz) + \sum_{z:\, c(xyz)>0} (u(xyz) - u(yz))$$

which sums over $u(yz)$ rather than $u(xyz)$ and then corrects the summands for the few trigrams where $u(yz) \neq u(xyz)$. The sub-computation $\sum_z u(yz)$ here is independent of x, so we can reuse it for other values of x. Let's call it $Z(y)$, and then speed up the computation of $Z(y)$ in turn by the same trick:

$$\begin{aligned}
Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz) \quad &= \underbrace{\sum_z u(yz)}_{\text{compute as } Z(y)} + \sum_{z:\, c(xyz)>0} (u(xyz) - u(yz)) \\
Z(y) \stackrel{\text{def}}{=} \sum_z u(yz) \quad &= \underbrace{\sum_z u(z)}_{\text{compute as } Z()} + \sum_{z:\, c(yz)>0} (u(yz) - u(z)) \\
Z() \stackrel{\text{def}}{=} \sum_z u(z) \quad &= \underbrace{\sum_z u()}_{\text{compute as } V} + \sum_{z:\, c(z)>0} (u(z) - u())
\end{aligned}$$

where $u() = \exp 0 = 1$ is the score of a trigram on which no features fire, i.e., even the final word z has never been observed.

In short, we can compute $Z()$ by looping once over all observed unigrams z, then compute all values of $Z(y)$ for observed y by looping once over all observed bigrams yz, and finally compute all values of $Z(xy)$ for observed xy by looping once over all observed bigrams xyz. The total runtime is $O(N)$.

(a) At the start of this question, I noted that we might need $Z(xy)$ at test time for any of the V^2 possible bigrams xy. But above, we've only computed it for $\leq N$ values of xy that were *observed* during training. So what can you do at test time

[9] Adding unobserved trigrams would make it trickier (though still possible) to preserve efficiency, and would not improve the model's ability to fit the data.

[10] We could formally define $u(yz) \stackrel{\text{def}}{=} u(\text{OOV}\ yz)$, or $u(yz) \stackrel{\text{def}}{=} \exp \sum_k \theta_k \cdot f_k(yz)$, where $\vec{f}$ has been extended to consider features of a bigram. The bigram yz is considered to have only features yz and z (if they exist).

to compute $Z(xy)$ efficiently for an *unobserved* bigram xy—one that you never saw during training ($c(xy) = 0$)? Answer in your `README`. (*Hint:* Look at our new formula for $Z(xy)$.)

(b) Add support for this log-linear model to the code. See the `INSTRUCTIONS` file for language-specific instructions.

Specifically, your code will need to compute $\hat{p}(z \mid xy)$. It should refer to the current parameters $\vec{\theta}$, and to a table of $Z(\cdots)$ values.

Your features k should correspond to the observed unigrams, bigrams, and trigrams in training data. For now, your weight vector $\vec{\theta}$ can be random or 0. This should be enough to make sure your code runs, and you can check that $\sum_z p(z \mid xy) = 1$ for some values of xy (if not, you have a bug).

It will be inconvenient to number your features $k = 0, 1, 2, \ldots$, since then to look up a trigram feature's weight, you would have to know its number. We recommend instead that you refer to your features by *name*—in fact, the maximization method that we are providing you assumes this design.

For example, you could choose to name the bigram feature "see the" by a string `"see the"` or some kind of n-gram object representing the pair (`"see"`,`"the"`). You can then implement $\vec{\theta}$ as a **map** from names k to numeric weights θ_k. Use your favorite map class; a hash table is a good choice.

When $\vec{\theta}$ is first set, and whenever it is changed, you will need to create or update the Z table. So make sure you only set $\vec{\theta}$ through an accessor method, which recomputes Z as a side effect. This ensures up-to-date Z values when they are looked up from within $\hat{p}(z \mid xy)$.

Don't hand in your code yet.

8. To implement conditional log-linear modeling, the main work is to train $\vec{\theta}$. Recall that $\hat{p}(z \mid xy)$ depends on your estimate of $\vec{\theta}$. Also recall from page 9 that given $\sigma^2 > 0$, you will estimate $\vec{\theta}$ by picking the value that maximizes

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \left(\sum_{xyz} c(xyz) \log \hat{p}(z \mid xy) \right) - \left(\frac{1}{2\sigma^2} \sum_k \theta_k^2 \right)$$

Fortunately, convex functions like $F(\vec{\theta})$ are "easy" to maximize. We provide you with a friendly maximization method (which calls a public optimization library such as SciPy (from Python) or Apache Commons Math (from Java)). This method requires

- an *objective function* to maximize, $F(\vec{\theta})$
- the *gradient* of the objective function, $\nabla F(\vec{\theta})$, so that it knows how to adjust $\vec{\theta}$ to improve $F(\vec{\theta})$

- an *initial guess* for $\vec{\theta}$, typically the vector $(0, 0, 0, \ldots)$

As for any conditional log-linear model, the gradient turns out to have a beautiful form, as follows:

$$\nabla F(\vec{\theta}) = \underbrace{\sum_{xyz} c(xyz)\vec{f}(xyz)}_{\textit{observed feature counts}} - \underbrace{\sum_{xyz} c(xy)\hat{p}(z \mid xy)\vec{f}(xyz)}_{\textit{expected feature counts}} \quad \underbrace{- \frac{1}{\sigma^2}\vec{\theta}}_{\textit{pulls } \vec{\theta} \textit{ toward 0}}$$

This obtains the gradient vector $\nabla F(\vec{\theta}) \in \mathbb{R}^N$ by summing multiples of three other vectors in $\mathbb{R}^N$: the observed feature counts in the training data, the expected feature counts if the current $\hat{p}$ (based on the current $\vec{\theta}$) were correct, and the current $\vec{\theta}$ itself. Notice that each feature vector $\vec{f}(xyz)$ is in $\mathbb{R}^N$ as well.

Questions to answer in your `README`:

(a) Try out the maximization routine on a simple polynomial function: just follow the directions in the `INSTRUCTIONS` file. What is the maximum value of this function, and for what $\vec{\theta}$ was it achieved?

(b) For simplicity, suppose that $\sigma^2 = \infty$ (no smoothing). In terms of page 8, this corresponds to just maximizing the likelihood $p(\vec{w} \mid \vec{\theta})$, rather than the likelihood times a prior.

Remember from basic calculus that at the maximum of F, its derivatives are 0. So if $\vec{\theta}$ maximizes $F(\vec{\theta})$, then $\nabla F(\vec{\theta}) = 0$. At this maximum, what must the expected feature counts look like? What can you say, therefore, about the probability distribution $\hat{p}(z \mid xy)$ that is constructed from this $\vec{\theta}$?

(c) According to the unsmoothed $\vec{\theta}$ from your previous answer, what is the probability of $\hat{p}(\texttt{zygote} \mid \texttt{see the})$, provided that $c(\texttt{see the zygote}) = 0$?

(You may assume that $c(\texttt{see the}) > 0$ and $c(\texttt{the zygote}) > 0$.)

(d) Still continuing with the unsmoothed $\vec{\theta}$ from your previous answer, what can you say about the weights of the three features of "`see the` OOV"? (Consider this carefully. Remember every xyz has trigram, bigram, and unigram features: xyz, $?yz$, and $??z$. All three features contribute to the probabilities.)

(e) Trickier: What can you say about the weights of the features of "`see` OOV `zygote`"? (For purposes of this question, assume our model includes features for all n-grams, not just observed n-grams, so that the feature of xyz are always xyz, yz, and z.)

(f) Now suppose $\sigma^2 < \infty$. How does this affect your answers to (c), (d), and (e)?

The gradient formula above was derived as follows (this works for *any* conditional log-linear model). A gradient $\nabla F(\vec{\theta})$ is merely the vector of partial derivatives $\left(\frac{\partial F(\vec{\theta})}{\partial \theta_1}, \frac{\partial F(\vec{\theta})}{\partial \theta_2}, \ldots\right)$. We can find the kth partial derivative by consulting the definition of $F(\vec{\theta})$ from question 8 and expanding the definition $\hat{p}(z \mid xy) \stackrel{\text{def}}{=} u(xyz)/Z(xy)$:

$$\begin{aligned}
\frac{\partial}{\partial \theta_k} F(\vec{\theta}) &= \left(\sum_{xyz} c(xyz) \frac{\partial}{\partial \theta_k} \log \hat{p}(z \mid xy)\right) - \left(\theta_k/\sigma^2\right) \\
&= \left(\sum_{xyz} c(xyz) \left(\frac{\partial}{\partial \theta_k} \log u(xyz) - \frac{\partial}{\partial \theta_k} \log Z(xy)\right)\right) - \left(\theta_k/\sigma^2\right) \\
&= \left(\sum_{xyz} c(xyz) \left(\frac{\partial}{\partial \theta_k} \log u(xyz)\right)\right) - \left(\sum_{xy} \sum_{z} c(xyz) \left(\frac{\partial}{\partial \theta_k} \log Z(xy)\right)\right) - \left(\theta_k/\sigma^2\right) \\
&= \underbrace{\left(\sum_{xyz} c(xyz) f_k(xyz)\right)}_{\textit{observed count of feature } k} - \underbrace{\left(\sum_{xy} c(xy) \left(\sum_{z} \hat{p}(z \mid xy) \cdot f_k(xyz)\right)\right)}_{\textit{expected count of feature } k} - \underbrace{\left(\theta_k/\sigma^2\right)}_{\textit{pulls } \theta_k \textit{ toward } 0}
\end{aligned}$$

Our beautiful expression for $\nabla F(\vec{\theta})$ in question 8 was just a concise way to write the above formula for all features k "in parallel," using vector notation.

We got the last step above by expanding the definitions of $\hat{p}$'s numerator $u(xyz) \stackrel{\text{def}}{=} \exp(\vec{\theta} \cdot \vec{f}(xyz))$ and denominator $Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz)$ and applying basic rules of differentiation:

$$\begin{aligned}
\frac{\partial}{\partial \theta_k} \log u(xyz) &= \frac{\partial}{\partial \theta_k} \left(\vec{\theta} \cdot \vec{f}(xyz)\right) \\
&= \frac{\partial}{\partial \theta_k} \left(\theta_k \cdot f_k(xyz) + \text{constant not involving } \theta_k\right) = f_k(xyz) \\
\frac{\partial}{\partial \theta_k} \log Z(xy) &= \frac{1}{Z(xy)} \frac{\partial}{\partial \theta_k} Z(xy) = \frac{1}{Z(xy)} \sum_z \frac{\partial}{\partial \theta_k} u(xyz) \\
&= \frac{1}{Z(xy)} \sum_z \frac{\partial}{\partial \theta_k} \exp(\vec{\theta} \cdot \vec{f}(xyz)) \\
&= \frac{1}{Z(xy)} \sum_z \left(\exp(\vec{\theta} \cdot \vec{f}(xyz))\right) \left(\frac{\partial}{\partial \theta_k} (\vec{\theta} \cdot \vec{f}(xyz))\right) \\
&= \frac{1}{Z(xy)} \sum_z u(xyz) \cdot f_k(xyz) \\
&= \sum_z \hat{p}(z \mid xy) \cdot f_k(xyz)
\end{aligned}$$

9. When computing the gradient from question 8, the expensive part is the expected feature counts, $\sum_{xyz} c(xy)\hat{p}(z \mid xy)\vec{f}(xyz)$. Expanding the definition of $\hat{p}(z \mid xy)$ (see page 8) and rearranging slightly, we can move some of the work out of the z loop to get

$$\sum_{xy} \frac{c(xy)}{Z(xy)} \sum_{z} u(xyz)\vec{f}(xyz)$$

The outer summation only has to loop over the N unique bigrams xy for which $c(xy) > 0$. We can look up $c(xy)$ and $Z(xy)$ rapidly, as done previously. But the inner summation has to loop over all of the V possible values of z, so the overall runtime is $O(NV)$:

```
for each feature k                // initialize sums to 0
   expected[k] = 0
for each unique bigram xy         // at most N of these
   coefficient = c(xy) / Z(xy)
   for each z in the vocabulary   // exactly V of these
      expected_count = coefficient * u(xyz)    // fast to compute
      for each feature k of xyz   // at most 3 of these
         expected[k] += expected_count
```

This expense of summing over the whole vocabulary should look familar. We had to worry about it before when we were computing the *denominator* in question 7. Our expected feature counts pose the same challenge, because they arise from the *derivative of that denominator*.

$O(NV)$ is not so fast for a large vocabulary, and we would dearly like to get that runtime down to $O(N)$. Fortunately, we can do so, using the same kind of trick that we used to compute $Z(xy)$ efficiently:[11]

[11] As it happens, these techniques were originally developed at JHU (using different notation), by Wu & Khudanpur (2000). The basic version only handles the basic feature set we use in this assignment (just unigrams, bigrams, and trigrams), but they showed how to extend it to certain larger sets of features.

$$
\begin{aligned}
&\sum_{xy} \frac{c(xy)}{Z(xy)} \sum_{z} u(xyz)\vec{f}(xyz) \\
&\quad = \sum_{xy} \frac{c(xy)}{Z(xy)} \Bigg(\sum_{z} u()\vec{f}() + \sum_{z\colon c(z)>0} (u(z)\vec{f}(z) - u()\vec{f}()) \\
&\qquad\qquad\qquad\qquad + \sum_{z\colon c(yz)>0} (u(yz)\vec{f}(yz) - u(z)\vec{f}(z)) \\
&\qquad\qquad\qquad\qquad + \sum_{z\colon c(xyz)>0} (u(xyz)\vec{f}(xyz) - u(yz)\vec{f}(yz)) \Bigg)
\end{aligned}
$$

Originally we defined $\vec{f}$ to extract the feature vector of a 3-gram, but here we are also applying $\vec{f}$ to 0-grams, 1-grams, and 2-grams, with the meaning explained in footnote 10. In particular, $\vec{f}()$ is the 0 vector where no features fire (this covers trigrams like "`see the` OOV"), so the terms $u()\vec{f}()$ vanish. Reading upward from the bottom, the inner sums $\sum_{z\colon c(xyz)>0}, \sum_{z\colon c(yz)>0}, \sum_{z\colon c(z)>0}$ are increasingly expensive—but they are also increasingly independent of the choice of xy, so they can be shared over more xy values in the outer sum $\sum_{xy}$. We therefore rearrange the above into

$$
\begin{aligned}
& R() \sum_{z\colon c(z)>0} u(z)\vec{f}(z) \\
&+ \sum_{y} R(y) \sum_{z\colon c(yz)>0} (u(yz)\vec{f}(yz) - u(z)\vec{f}(z)) \\
&+ \sum_{xy} R(xy) \sum_{z\colon c(xyz)>0} (u(xyz)\vec{f}(xyz) - u(yz)\vec{f}(yz))
\end{aligned}
$$

where

$$
\begin{aligned}
R() &\stackrel{\text{def}}{=} \sum_{xy} \frac{c(xy)}{Z(xy)} && \left(= \sum_{y} R(y)\right) \\
R(y) &\stackrel{\text{def}}{=} \sum_{x} \frac{c(xy)}{Z(xy)} && \left(= \sum_{x} R(xy)\right) \\
R(xy) &\stackrel{\text{def}}{=} \frac{c(xy)}{Z(xy)}
\end{aligned}
$$

So after you compute the Z table, you can loop over the observed bigram and unigram *types* to compute R tables, and then over the observed unigram, bigram, and trigram

types to compute the expected counts as desired. Of course, you only have to loop over types whose summands will be non-zero.

Note that the final computation of expected counts is adding up a bunch of positive and negative multiples of $\vec{f}(\cdots)$ vectors. To add a multiple of a particular $\vec{f}(\cdots)$ vector to the running total, you only have to iterate through its few nonzero elements (corresponding to the few features that fire on $\cdots$). We saw this technique earlier in the naive pseudocode at the start of this question.

(a) Modify your code so that it sets $\vec{\theta}$ to maximize the objective function from question 8, which looks at the counts on training data as well as $\vec{\theta}$ and σ^2. Hand this code in.

The main job is to write methods that compute $F(\vec{\theta})$ and $\nabla F(\vec{\theta})$. The maximization routine that we gave you will try calling these methods with *various* values of $\vec{\theta}$. So each time one of these methods is called, it should probably start by recomputing the Z table and (in the case of ∇F) the R table.

(b) You should now be able to measure cross-entropies and text categorization error rates under your new log-linear backoff language model! Your text categorization program should work as before: it will construct two log-linear language models as above, and then compare the probabilities of a test document under these models.

Try your program out. Report results with $\sigma^2 = 1$, and experiment with other values of $\sigma^2 > 0$, including a large value such as $\sigma^2 = 20$. How and when did you use the training, development, and test data? What did you find? How do your results compare to add-λ backoff smoothing?

(c) Which would you expect to be larger in general, unigram or trigram weights? Explain your answer. (*Hint:* Usually $c(z) \gg c(xyz)$.) Now look at $\vec{\theta}$: were you right?

(d) Do you find any other interesting patterns in $\vec{\theta}$? For example, you might expect that the unigram feature for z will have a high positive weight if $c(z)$ is large. Does this seem to be true in practice? What if $c(yz) \approx c(z)$?

(e) What will tend to happen to the estimated probability of *test* data as $\sigma^2 \to \infty$? (Try it with `fileprob`!)

10. Suppose you expect *a priori* that $\frac{1}{3}$ of your test emails will be spam. (In fact this is true for the test and development data!)

    Or, in the language ID task, assume all the documents you view are in either Spanish or English, and you expect *a priori* that $\frac{1}{3}$ of them will be in Spanish.

How should this affect how `textcat.c` does its classification, and why? (Just give a formula, don't implement it.) *Hint:* Bayes' Theorem.

Do you need to know the number $\frac{1}{3}$ when you train the language model? Or is it only used at test time, so that each individual user could adjust it at runtime (without having to retrain) to match the fraction of spam that they *actually* currently get?

Extra credit: Implement this change and measure how it affects performance on the spam detection task with various smoothing methods (since, in fact, $\frac{1}{3}$ of the test data for that task *are* spam, making the *a priori* expectation correct). Does it help? Why or why not? What happens for values other than $\frac{1}{3}$ (perhaps try adjusting the prior gradually from 0 to 1)?

11. Finally, we turn to speech recognition. Here, instead of choosing the best model for a given string, you will choose the best string for a given model.

    The data are in the `speech` subdirectory. As usual, a development set and a test set are available to you; you may experiment on the development set before getting your final results from the test set. You should use the `switchboard` corpus as your training.

    Here is a sample file (`dev/easy/easy025`):

```
8       i found that to be %hesitation very helpful
0.375   -3524.81656881726       8       <s> i found that the uh it's very helpful </s>
0.250   -3517.43670278477       9       <s> i i found that to be a very helpful </s>
0.125   -3517.19721540798       8       <s> i found that to be a very helpful </s>
0.375   -3524.07213817617       9       <s> oh i found out to be a very helpful </s>
0.375   -3521.50317920669       9       <s> i i've found out to be a very helpful </s>
0.375   -3525.89570470785       9       <s> but i found out to be a very helpful </s>
0.250   -3515.75259677371       8       <s> i've found that to be a very helpful </s>
0.125   -3517.19721540798       8       <s> i found that to be a very helpful </s>
0.500   -3513.58278343221       7       <s> i've found that's be a very helpful </s>
```

    Each file has 10 lines and represents a single audio-recorded utterance U. The first line of the file is the correct transcription, preceded by its length in words. The remaining 9 lines are some of the possible transcriptions that were considered by a speech recognition system—including the one that the system actually chose to output. You will similarly write a program that chooses among those 9 candidates.

    Consider the last line of the sample file. The line shows a 7-word transcription $\vec{w}$ surrounded by `<s>`... `</s>` and preceded by its length, namely 7. The number -3513.58 was the speech recognizer's estimate of $\log_2 p(U \mid \vec{w})$: that is, if someone really were trying to say $\vec{w}$, what is the log-probability that it would have come out of their mouth sounding like U?[12] Finally, $0.500 = \frac{4}{8}$ is the **word error rate** of this transcription,

[12] Actually, the real estimate was 15 times as large. Speech recognizers are really rather bad at estimating

which had 4 errors against the 8-word true transcription on the first line of the file.[13]

(a) According to Bayes' Theorem, how should you choose among the 9 candidates? That is, what quantity are you trying to maximize, and how should you compute it?

(*Hint:* You want to pick a candidate that both looks like English and looks like the audio utterance U. Your trigram model tells you about the former, and -3513.58 is an estimate of the latter.)

(b) Modify `fileprob` to obtain a new program `speechrec` that chooses this best candidate. As usual, see `INSTRUCTIONS` for details.

The program should look at each utterance file listed on the command line, choose one of the 9 transcriptions according to Bayes' Theorem, and report the word error rate of that transcription (as given in the first column). Finally, it should summarize the overall word error rate over all the utterances—the *total* number of errors divided by the *total* number of words in the correct transcriptions.

Of course, the program is not allowed to cheat: when choosing the transcription, it must ignore each file's first row and first column!

Sample input (please allow this format; `switchboard` is the training corpus):

```
speechrec add1 switchboard easy025 easy034
```

Sample output (please use this format—but you are not required to get the same numbers):

```
0.125   easy025
0.037   easy034
0.057   OVERALL
```

Notice that the overall error rate 0.057 is not an equal average of 0.125 and 0.037; this is because `easy034` is a longer utterance and counts more heavily.

Hints about how to read the file:

- For all lines but the first, you should read a few numbers, and then as many words as the integer told you to read. (Alternatively, you could read the whole line at once and break it up into an array of whitespace-delimited strings.)

$\log p(U \mid \vec{w})$, so they all use a horrible hack of dividing this value by about 15 to prevent it from influencing the choice of transcription too much! But for the sake of this question, just pretend that no hack was necessary and -3513.58 was the actual value of $\log_2 p(U \mid \vec{w})$ as stated above.

[13] The word error rate of each transcription has already been computed by a scoring program. The correct transcription on the first line sometimes contains special notation that the scorer paid attention to. For example, `%hesitation` on the first line told the scorer to count either `uh` or `um` as correct.

- For the first line, you should read the initial integer, then read the rest of the line. The rest of the line is only there for your interest, so you can throw it away. The scorer has already considered the first line when computing the scores that start each remaining line.
  Warning: For the first line, the notational conventions are bizarre, so in this case the initial integer *does not necessarily tell you* how many whitespace-delimited words are on the line. Be sure to throw away the rest of the line anyway! (If necessary, read and discard characters up through the end-of-line symbol `\n`.)

(c) What is your program's overall error rate on the carefully chosen utterances in `test/easy`? How about on the random sample of utterances in `test/unrestricted`? Answer for 3-gram, 2-gram, and 1-gram models.

To get your answer, you need to choose a smoothing method, so pick one that seems to work well on the development data `dev/easy` and `dev/unrestricted`. Be sure to tell us which method you picked and why! What would be an *unfair* way to choose a smoothing method?

Hint: Some options for handling the 2-gram and 1-gram models:

- You'll already have a `probs`(x, y, z) function. You could add `probs`(y, z) and `probs`(z).
- You could give `probs`(x, y, z) an extra argument that controls which kind of model it computes. For example, for a 2-gram model, it would ignore x.
- Or you could make life easy for yourself, and just call the existing `probs()` function with arguments that will make it return a bigram or unigram probability. For example, if you choose a backoff smoothing method, then $p(z \mid \text{OOV}, y)$ will back off completely to $p(z \mid y)$, which is the bigram probability that you want!

---

12. *Extra credit:* We have been assuming a finite vocabulary by replacing all unknown words with a special OOV symbol. But notice that if the *alphabet* is finite, you could predict the probability of an unknown word by using ... you got it, a letter n-gram model! Such a prediction is sensitive to the spelling and length of the unknown word. As longer words will generally receive lower probabilities, it is possible for the probabilities of all unknown words to sum to 1, even though there are infinitely many of them. (Just as $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$.)

    Devise a sensible way to estimate the word trigram probability $p(z \mid xy)$ by backing off to a letter n-gram model of z if z is an unknown word. Also describe how you would train the letter n-gram model.

Just give the formulas for your estimate—you don't have to implement and test your idea, although that would be nice too!

Notes:

- x and/or y and/or z may be unknown; be sure you make sensible estimates of $p(z \mid xy)$ in all these cases
- be sure that $\sum_z p(z \mid xy) = 1$

13. *Extra credit:* Previous students had to implement Witten-Bell backoff instead of a log-linear model. You are welcome to do this too, for extra credit. Even if you don't do the whole problem, it's worth spending a little time reading it.

    Witten-Bell backoff as described on page 7 is conceptually pretty simple. The tricky part will be finding the right α values to make the probabilities sum to 1. This is sort of like finding $\vec{\theta}$ in the log-linear model: again, you have to work out a few formulas, and then rearrange the loops in order to reduce a large runtime to $O(N)$.

    Even if you don't do this extra-credit problem in full,

    (a) Witten-Bell discounting will discount some probabilities more than others. When is $p_{\text{disc}}(z \mid xy)$ very close to the naive historical estimate $c(xyz)/c(xy)$? When is it far less (i.e., heavily discounted)? Give a practical justification for this policy.

    (b) What if we changed the Witten-Bell discounting formulas to make all T values be zero? What would happen to the discounted estimates? What would the α values have to be, in order to make the distributions sum to 1?

    (c) Observe that the set of zero-count words $\{z : c(z) = 0\}$ has size $V - T()$.[14] What is the simple formula for $\alpha()$?

    (d) Now let's consider $\alpha(xy)$. Let $Z(xy)$ be the set $\{z : \; c(xyz) > 0\}$. Observe that

$$\begin{aligned}
\sum_z p(z \mid xy) &= \left(\sum_{z \in Z(xy)} p_{\text{disc}}(z \mid xy) \right) + \left(\alpha(xy) \cdot \sum_{z \notin Z(xy)} p(z \mid y) \right) \\
&= \left(\sum_{z \in Z(xy)} p_{\text{disc}}(z \mid xy) \right) + \alpha(xy) \cdot \left(1 - \sum_{z \in Z(xy)} p(z \mid y) \right)
\end{aligned}$$

[14] You might think that this set is just {OOV}, but that depends on how the finite vocabulary was chosen. There might be other zero-count words as well: this is true for your `gen` and `spam` (or `english` and `spanish`) models, since the vocabulary is taken from the union of both corpora. Conversely, it is possible for $c(\text{OOV}) > 0$, since in general one might decide to omit rarely observed words from one's vocabulary, treating them as OOV when they appear in training.

To make $\sum_z p(z \mid xy) = 1$, solving the equation shows that you will need[15]

$$\alpha(xy) = \frac{1 - \sum_{z \in Z(xy)} p_{\text{disc}}(z \mid xy)}{1 - \sum_{z \in Z(xy)} p(z \mid y)}$$

Got that? Now, the first step in the derivation above assumed that $\sum_z p(z \mid y) = 1$. Give a formula for $\alpha(y)$ that ensures this. The formula will be analogous to the one we just derived for $\alpha(xy)$. (*Hint:* Start by defining the set $Z(y)$.)

(e) Finally, we should figure out how the above formula for $\alpha(xy)$ can be *computed efficiently*. Smoothing code can take up a lot of the execution time. It's always important to look for ways to speed up critical code—in this case by cleverly rearranging the formula. The slow part is those two summations ...

i. Simplify the subexpression $\sum_{z \in Z(xy)} p_{\text{disc}}(z \mid xy)$ in the numerator, by using the definition of p_{disc} and any facts you know about $c(xy)$ and $c(xyz)$. You should be able to eliminate the $\sum$ sign altogether.

ii. Now consider the $\sum$ sign in the denominator. Argue that $c(yz) > 0$ for every $z \in Z(xy)$. That allows the following simplification: $\sum_{z \in Z(xy)} p(z \mid y) = \sum_{z \in Z(xy)} p_{\text{disc}}(z \mid y) = \frac{\sum_{z \in Z(xy)} c(yz)}{c(y)+T(y)}$.
(*Warning*: You can't use this simplification when it leads to 0/0. But in that special case, what can you say about the context xy? What follows about $\alpha(xy)$?)

iii. The above simplification still leaves you with a sum in the denominator. But you can compute this sum efficiently in advance.
Write a few lines of pseudocode that show how to compute $\sum_{z \in Z(xy)} c(yz)$ for every observed bigram xy. You can compute and store these sums immediately *after* you finish reading in the training corpus. At that point, you will have a list of trigrams xyz that have actually been observed (the provided code helpfully accumulates such a list for you), and you will know $c(yz)$ for each such trigram.
Armed with these sums, you will be able to compute $\alpha(xy)$ in $O(1)$ time when you need it during testing. You should not have to do any summation during testing.

Remark: Of course, another way to avoid summation during testing would be for training to precompute $p(z \mid xy)$ for all possible trigrams xyz. However, since there are V^3 possible trigrams, that would take a lot of time

[15] Should we worry about division by 0 (in which case the equation has no solution)? Since $p(z \mid y)$ is smoothed to be > 0 for all z, this problem occurs if and only if *every* z in the vocabulary, including OOV, has appeared following xy. Fortunately, you defined the vocabulary to include all words that were actually observed, so no OOV words can ever have appeared following xy. So the problem cannot occur for you.

and memory. Instead, you'd like training for an n-gram model to only take time about proportional to the number of *tokens* N in the training data (which is usually far less than V^n, and does not grow as you increase n), and memory that is about proportional to the number of n-gram *types* that are actually observed in training (which is even smaller than N). That's what you just achieved by rearranging the computation.

(f) Explain how to compute the formula for $\alpha(y)$ efficiently. Just use the same techniques as you did for $\alpha(xy)$ above. This is easy, but it's helpful to write out the solution before you start coding.

(g) Now implement Witten-Bell backoff using the techniques above. How does this smoothing method (which does not use any λ) affect your error rate when you repeat the text categorization test in 3c?

The `INSTRUCTIONS` file gives language-specific instructions about which files to modify and submit.

Hint: Here are two techniques to check that you are computing the α values correctly:

- Write a loop that checks that $\sum_z p(z \mid xy) = 1$ for all x, y. (This check will slow things down since it takes $O(V^3)$ time, so only use it for testing and debugging.)
- Use a tiny 5-word training corpus. Then you will be able to check your smoothed probabilities by hand.

14. *Extra credit:* Problem 13a asked you to justify Witten-Bell discounting. Suppose you redefined $T(xy)$ in Witten-Bell discounting to be the number of word types z that have been observed *exactly once* following xy in the training corpus. What is the intuition behind this change? Why might it help (or hurt, or not matter much)? If you dare, try it out and report how it affects your results.