600.465 — Natural Language Processing
Assignment 3: Smoothed Language Modeling

Prof. J. Eisner — Spring 2016
Due date: Friday 4 March, 2 pm

Statistical models are an indispensable part of modern NLP.[1] This assignment will try to convince you that even simplistic and linguistically stupid models like n-gram models can be very useful, provided their parameters are estimated carefully.

Most systems for speech recognition continue to use trigram models.[2] 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. This is the only assignment in the course to focus on that.

Collaboration: ***You may work in groups of up to 2 on this assignment.*** That is, if you choose, you may collaborate with 1 partner from the class, handing in a single homework with multiple names on it. You are expected to do the work *together*, not divide it up: your solutions should emerge from collaborative real-time discussions with both of you present.

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.[3] This code is currently provided in Java and Python. 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 find yourself doing so, then by all means seek help from someone who knows the language

[1] A statistical system without real linguistics may get reasonable results. By contrast, a linguistic system without statistics is often fragile, and may break when run on real data. (There are exceptions in both directions, though.)

[2] Similarly, today's machine translation (MT) systems often include 5-gram models trained on massive amounts of data. Why 5-grams rather than 3-grams? An MT system has to generate a *new* fluent sentence of English, and 5-grams do a better job than 3-grams of memorizing common phrases and local grammatical phenomena. By contrast, a speech recognition system only has to determine what words have *already* been said by an English speaker. A 3-gram model helps to choose between "flower," "flour," and "floor" by using one word of context on either side. That already provides most of the value that we can get out of *local* context. Going to a 5-gram model wouldn't help too much with this choice, because it still wouldn't look at enough of the sentence to determine whether we're talking about gardening, baking, or cleaning.

[3] 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.

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^{TM}.

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 word sequence* $\vec{w}$. 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 boundary symbols handled as in the previous assignment. In other words, w_N = EOS ("end of sentence"), while for $i < 1$, w_i = BOS ("beginning of sentence"). Thus, $|\vec{w}| = N - 1$, not counting the extra EOS symbol. The boundary symbols are special symbols that never appear in $\vec{w}$ proper.

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. You should regard these other words as nothing more than variant spellings of the OOV symbol. Note that OOV stands for "out of vocabulary," not for "out of corpus," so OOV words may have token count > 0 and in-vocabulary words may have count 0.

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.[4]

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 $p_{\text{model1}}(\texttt{i saw OOV on the tv})$ with $p_{\text{model2}}(\texttt{i saw snuffleupagus on the tv})$, 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.

Even if 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 word types that appeared ≥ 3 times anywhere in *training* data. Then augment this set with a special OOV symbol. Let V be the size of the resulting set (including OOV). Whenever you read a training or test word, you should immediately convert it to OOV if it's not in the vocabulary. This is fast to check if you store the vocabulary in a hash set.

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, where the training data is assumed to be `train/switchboard`). This lets you identify such words at a glance. In this dataset, for example, we convert `uncertain` to `[uncertain]`—this doesn't change its count, but does indicate that this is one of the words that your code ought to convert to OOV. You can experiment on that dataset, starting with the following problem.

---

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![5]

[4] Problem 9 explores a more elegant approach that may also work better for text categorization.

[5] There is one little problem with our setup. To simplify the command-line syntax, I've assumed that the training corpus consists of a *single* big file. That means that there is only one BOS and one EOS in the whole training corpus—

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, and `words-10.txt` is a lexicon containing word vectors):

   ```
   textcat add1 words-10.txt 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 all words that appeared ≥ 3 times in the *union* of the two training corpora, 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.

whereas BOS and EOS are much more common in the collection of short test files. 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.

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^*: for simplicity, you will use $\lambda = \lambda^*$ 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^* 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. To include graphs, see `http://cs.jhu.edu/~jason/465/hw-lm/graphing.html`.

   (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,

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

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

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

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda}{c(xy) + \lambda V} \tag{2}$$

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':

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda V \cdot \hat{p}(z \mid y)}{c(xy) + \lambda V} \tag{3}$$

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.

$$\hat{p}(z \mid xy) = \begin{cases} p_{\text{disc}}(z \mid xy) & \text{if } c(xyz) > 0 \\ \alpha(xy)\hat{p}(z \mid y) & \text{otherwise} \end{cases} \tag{4}$$

$$\hat{p}(z \mid y) = \begin{cases} p_{\text{disc}}(z \mid y) & \text{if } c(yz) > 0 \\ \alpha(y)\hat{p}(z) & \text{otherwise} \end{cases} \tag{5}$$

$$\hat{p}(z) = \begin{cases} p_{\text{disc}}(z) & \text{if } c(z) > 0 \\ \alpha() & \text{otherwise} \end{cases} \tag{6}$$

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

$$p_{\text{disc}}(z \mid xy) = \frac{c(xyz)}{c(xy) + T(xy)} \tag{7}$$

$$p_{\text{disc}}(z \mid y) = \frac{c(yz)}{c(y) + T(y)} \tag{8}$$

$$p_{\text{disc}}(z) = \frac{c(z)}{c() + T()} \tag{9}$$

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. This count includes the EOS token at the end of each sequence.

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) In the previous assignment, you learned how to construct log-linear models. Let's restate that construction in our current notation.[6] Given a trigram xyz, our model $\hat{p}$ is defined by

$$\hat{p}(z \mid xy) \stackrel{\text{def}}{=} \frac{u(xyz)}{Z(xy)} \tag{10}$$

where

$$u(xyz) \stackrel{\text{def}}{=} \exp\left(\sum_k \theta_k \cdot f_k(xyz)\right) = \exp\left(\vec{\theta} \cdot \vec{f}(xyz)\right) \tag{11}$$

$$Z(xy) \stackrel{\text{def}}{=} \sum_z u(xyz) \tag{12}$$

Here $\vec{f}(xyz)$ is the feature vector extracted from xyz, and $\vec{\theta}$ is the model's weight vector.

What features should we use? It would be possible to use one binary feature for each specific unigram z, bigram yz, and trigram xyz (see problem 6(g)i). Instead, however, let's start with the following model based on word embeddings:

$$u(xyz) \stackrel{\text{def}}{=} \exp\left(\vec{x}^T U \vec{z} + \vec{y}^T V \vec{z}\right) \tag{13}$$

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

where the vectors $\vec{x}, \vec{y}, \vec{z}$ are specific d-dimensional embeddings of the word types x, y, z, while U, V are $d \times d$ matrices. The ^{T} superscript is the matrix transposition operator, used here to transpose a column vector to get a row vector.

You can use embeddings of your choice from the `lexicons` directory. (See the `README` file in that directory.) In the last assignment, we provided word embeddings that were derived from Wikipedia, a far larger corpus that contains lots of useful evidence about the usage of many English words. For OOV, or for any word type that is in our gen/spam vocabulary but has no embedding listed in the lexicon, you should back off to the embedding of OOL (the special "out of lexicon" symbol).

But is (13) really a log-linear function? Yes it is! Let's write out those d-dimensional vector-matrix-vector multiplications:

$$u(xyz) = \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} x_j U_{jm} z_m + \sum_{j=1}^{d}\sum_{m=1}^{d} y_j V_{jm} z_m\right) \tag{14}$$

$$= \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} U_{jm} \cdot (x_j z_m) + \sum_{j=1}^{d}\sum_{m=1}^{d} V_{jm} \cdot (y_j z_m)\right) \tag{15}$$

This does have the log-linear form of (11). Suppose $d = 2$. Then implicitly, we are using a weight vector $\vec{\theta}$ of length 8 defined by

$$\begin{array}{cccccccc} \langle\, \theta_1, & \theta_2, & \theta_3, & \theta_4, & \theta_5, & \theta_6, & \theta_7, & \theta_8 \;\rangle \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \langle U_{11}\,, & U_{12}\,, & U_{21}\,, & U_{22}\,, & V_{11}\,, & V_{12}\,, & V_{21}\,, & V_{22}\,\rangle \end{array} \tag{16}$$

for a vector $\vec{f}(xyz)$ of 8 features defined by pairwise products of embedding dimensions,

$$\begin{array}{cccccccc} \langle f_1(xyz), & f_2(xyz), & f_3(xyz), & f_4(xyz), & f_5(xyz), & f_6(xyz), & f_7(xyz), & f_8(xyz)\,\rangle \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \langle\; x_1 z_1, & x_1 z_2, & x_2 z_1, & x_2 z_2, & y_1 z_1, & y_1 z_2, & y_2 z_1, & y_2 z_2 \;\rangle \end{array} \tag{17}$$

As an example, let's calculate the letter trigram probability $\hat{p}(\mathtt{s} \mid \mathtt{er})$. Suppose the relevant letter embeddings and the feature weights are given by

$$\vec{\mathtt{e}} = \begin{bmatrix} -.5 \\ 1 \end{bmatrix}, \vec{\mathtt{r}} = \begin{bmatrix} 0 \\ .5 \end{bmatrix}, \vec{\mathtt{s}} = \begin{bmatrix} .5 \\ .5 \end{bmatrix}, U = \begin{bmatrix} 1 & 0 \\ 0 & .5 \end{bmatrix}, V = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

First, we compute the unnormalized probability.

$$u(\mathtt{ers}) = \exp\left([-.5\ 1] \begin{bmatrix} 1 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} .5 \\ .5 \end{bmatrix} + [0\ .5] \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} .5 \\ .5 \end{bmatrix}\right)$$
$$= \exp(-.5 \times 1 \times .5 \;+\; 1 \times .5 \times .5 \;+\; 0 \times 2 \times .5 \;+\; .5 \times 1 \times .5) = 0.25$$

We then normalize $u(\mathtt{ers})$.

$$\hat{p}(\mathtt{s} \mid \mathtt{er}) \stackrel{\text{def}}{=} \frac{u(\mathtt{ers})}{Z(\mathtt{er})} = \frac{u(\mathtt{ers})}{u(\mathtt{era}) + u(\mathtt{erb}) + \cdots + u(\mathtt{erz})} = \frac{0.25}{0.75} \approx 0.67 \tag{18}$$

As always, $\vec{\theta}$ is incredibly important: the way we pick it will determine all the probabilities in the model. As usual, we'll select some $C \geq 0$ and set $\vec{\theta}$ to maximize

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \underbrace{\left(\sum_{i=1}^{N} \log \hat{p}(w_i \mid w_{i-2}\ w_{i-1})\right)}_{\text{log likelihood}} - \underbrace{\left(C \cdot \sum_k \theta_k^2\right)}_{\text{L}_2 \text{ regularizer}} \tag{19}$$

So we want $\vec{\theta}$ to make our training corpus probable, or equivalently, to make the N events in the corpus (including the final EOS) probable given their bigram contexts. At the same time, we also want the weights in $\vec{\theta}$ to be close to 0, other things equal (regularization).[7]

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) Let's see how backoff behaves with novel trigrams versus almost-novel trigrams. 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.)

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!

[7] As explained on the previous homework, this can also be interpreted as maximizing $p(\vec{\theta} \mid \vec{w})$—that is, choosing the most probable $\vec{\theta}$ given the training corpus. By Bayes' Theorem, $p(\vec{\theta} \mid \vec{w})$ is proportional to

$$\underbrace{p(\vec{w} \mid \vec{\theta})}_{\text{likelihood}} \cdot \underbrace{p(\vec{\theta})}_{\text{prior}} \tag{20}$$

Let's assume an independent Gaussian prior over each θ_k, with variance σ^2. Then if we take $C = 1/2\sigma^2$, maximizing (19) is just maximizing the log of (20). The reason we maximize the log is to avoid underflow, and because the derivatives of the log happen to have a simple "observed − expected" form (since the log sort of cancels out the exp in the definition of $u(xyz)$.)

(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 from ADDL to BACKOFF_ADDL affect your performance (with $\lambda = \lambda^*$)? Measure cross-entropies for the switchboard corpora as in question 1, and text categorization error rates for english/spanish or gen/spam as in question 3c.

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

6. To implement the conditional log-linear model, the main work is to train $\vec{\theta}$ (given some training data and a regularization coefficient C). Fortunately, concave functions like $F(\vec{\theta})$ in (19) are "easy" to maximize. We ask you to implement a simple *stochastic gradient descent (SGD)* method to do this optimization.

   More properly, this should be called stochastic gradient *ascent*, since we are maximizing rather than minimizing, but that's just a simple change of sign. The pseudocode is given by Algorithm 1. We rewrite the objective $F(\vec{\theta})$ given in (19) as a sum of local objectives $F_i(\vec{\theta})$ that each predict a single word, by moving the regularization term into the summation.

$$F(\vec{\theta}) = \sum_{i=1}^{N} \underbrace{\left(\log \hat{p}(w_i \mid w_{i-2}\ w_{i-1}) - \frac{C}{N} \cdot \sum_k \theta_k^2 \right)}_{\text{call this } F_i(\vec{\theta})} \tag{21}$$

$$= \sum_{i=1}^{N} F_i(\vec{\theta}) \tag{22}$$

Algorithm 1 Stochastic gradient ascent

Input: Initial learning rate γ_0, initial parameter values $\vec{\theta}^{(0)}$, training corpus $\mathcal{D} = (w_1, w_2, \cdots, w_N)$, regularization coefficient C, number of epochs E

procedure SGATrain
  $\vec{\theta} \leftarrow \vec{\theta}^{(0)}$
  $t \leftarrow 0$ # number of updates so far
  for $e : 1 \rightarrow E$ **do**
    for $i : 1 \rightarrow N$ **do** # loop over summands of (21)
      $\gamma \leftarrow \frac{\gamma_0}{1 + \gamma_0 \cdot \frac{C}{N} \cdot t}$ # current learning rate[8]—decreases gradually
      $\vec{\theta} \leftarrow \vec{\theta} + \gamma \cdot \nabla F_i(\vec{\theta})$ # move $\vec{\theta}$ slightly in a direction that increases $F_i(\vec{\theta})$
      $t \leftarrow t + 1$
  return $\vec{\theta}$

The gradient vector $\nabla F_i(\vec{\theta})$ describes the direction in which you can push $\vec{\theta}$ to get the fastest increase in $F_i(\vec{\theta})$. Since the algorithm above tries to improve each summand $F_i(\vec{\theta})$ in turn, some of these updates will partly cancel one another out,[9] but their average effect will be to improve the overall objective $F(\vec{\theta})$. Since we are training a log-linear model, our $F(\vec{\theta})$ is a concave function with a single global maximum; the algorithm will converge to that maximum if allowed to run forever ($E = \infty$).

The gradient vector $\nabla F_i(\vec{\theta})$ is merely the vector of partial derivatives $\left(\frac{\partial F_i(\vec{\theta})}{\partial \theta_1}, \frac{\partial F_i(\vec{\theta})}{\partial \theta_2}, \ldots\right)$. where $F_i(\vec{\theta})$ was defined in (21). As you'll recall from the previous assignment, each partial derivative takes a simple and beautiful form

$$\frac{\partial F_i(\vec{\theta})}{\partial \theta_k} = \underbrace{f_k(xyz)}_{\text{observed value of feature } f_k} - \underbrace{\sum_{z'} \hat{p}(z' \mid xy) f_k(xyz')}_{\text{expected value of feature } f_k, \text{ according to current } \hat{p}} - \underbrace{\frac{2C}{N}\theta_k}_{\text{pulls } \theta_k \text{ towards 0}} \tag{23}$$

where x, y, z respectively denote w_{i-2}, w_{i-1}, w_i, and the summation variable z' in the second term ranges over all V words in the vocabulary, including OOV. This obtains the partial derivative by summing multiples of three values: the *observed* feature count in the training data, the *expected* feature counts according to the current $\hat{p}$ (which is based on the *entire* current $\vec{\theta}$, not just θ_k), and the current weight θ_k itself.

When we use the specific model in (13), the feature weights are the entries of the U and V matrices, as shown in (15). The partial derivatives with respect to these weights are

$$\frac{\partial F_i(\vec{\theta})}{\partial U_{jm}} = x_j z_m - \sum_{z'} \hat{p}(z' \mid xy) x_j z'_m - \frac{2C}{N} U_{jm} \tag{24}$$

$$\frac{\partial F_i(\vec{\theta})}{\partial V_{jm}} = y_j z_m - \sum_{z'} \hat{p}(z' \mid xy) y_j z'_m - \frac{2C}{N} V_{jm} \tag{25}$$

where as before, we use $\vec{x}, \vec{y}, \vec{z}, \vec{z}'$ to denote the embeddings of the words x, y, z, z'. Thus, the update to $\vec{\theta}$ (Algorithm 1, line 7) is

$$(\forall j, m = 1, 2, \ldots d) \quad U_{jm} \leftarrow U_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial U_{jm}} \tag{26}$$

$$(\forall j, m = 1, 2, \ldots d) \quad V_{jm} \leftarrow V_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial V_{jm}} \tag{27}$$

In practice, the convergence rate of stochastic gradient ascent is sensitive to the learning rate γ_0 and initial guess $\vec{\theta}^{(0)}$. For starters, we recommend using $\gamma_0 = 0.1$ and $\vec{\theta}^{(0)} = \vec{0}$ (e.g., initialize all entries of U and V to 0). Of course, you're welcome to experiment with different settings.

[8] Recommended by Bottou (2012), "Stochastic gradient descent tricks," which you should read if you want to use this method "for real" on your own problems.

[9] For example, in the training sentence `eat your dinner and eat your words`, $\nabla F_3(\vec{\theta})$ is trying to raise the probability of `dinner`, while $\nabla F_7(\vec{\theta})$ is trying to raise the probability of `words` (at the expense of `dinner`!) in the same context.

(a) Add support for the LOGLIN 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 $\vec{\theta}$ (the entries of the U and V matrices, along with the weights of any other features we add later).

(b) Implement stochastic gradient ascent (Algorithm 1) to find the U and V matrices that maximize $F(\vec{\theta})$. See the `INSTRUCTIONS` file in the code directory for details.

(c) Try training a language model on `en.1K`, with character embeddings $d = 10$ (`ch-10.txt`). Train for 10 epochs, and place the output in your `README` in the following format (the objective function values are just placeholders):

```
epoch 1: F=-65535
epoch 2: F=-1023
```

Notice that we are asking for $F(\vec{\theta})$ rather than $F_i(\vec{\theta})$. We'll post some tips on Piazza for checking that the code works correctly.

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

Training a log-linear model takes significantly more time than ADDL smoothing. Therefore, we recommend that you experiment with the language identification task for this part. Try your program out. Report cross-entropy and categorization accuracy with $C = 1$, and also experiment with other values of $C > 0$, including a small value such as $C = 0.05$. Let C^* be the best value you find. Using $C = C^*$, play with different embedding dimensions and report the results. 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?

(e) *Extra credit:* Try out your program on the spam detection task. This doesn't require very much work, but training will be much slower because of the larger vocabulary (slowing down $\sum_{z'}$) and the larger training corpus (slowing down $\sum_i$). See `lexicons/README` for various lexicons of embeddings that you could try. You may need to adjust C, using development data as usual. Report what you did.

(f) Now let's add some more features. A possible problem with the model so far is that it doesn't have any parameters that keep track of how frequent specific words are in the training corpus. Rather, it backs off from the words to their embeddings: its probability estimates are based *only* on the embeddings. One way to fix that (see problem 6(g)i) would be to have a binary feature f_w for each word w in the vocabulary, such that $f_w(xyz)$ is 1 if $z = w$ and 0 otherwise. Here's a simpler method: just add a single *non-binary* feature defined by

$$f_{\text{unigram}}(xyz) = \log \hat{p}_{\text{unigram}}(z) \tag{28}$$

where $\hat{p}_{\text{unigram}}(z)$ is estimated by add-1 smoothing. Surely we have enough training data to learn an appropriate weight for this single feature. In fact, because *every* training

token w_i provides evidence about this single feature, its weight will tend to converge quickly to a reasonable value during SGD.

This is not the only feature in the model—as usual, you will use SGD to train the weights of *all* features to work together, computing the gradient via (23). Let $\beta = \theta_{\text{unigram}}$ denote the weight that we learn for the new feature. By including this feature in our definition of $\hat{p}_{\text{unigram}}(z)$, we are basically multiplying a factor of $(\hat{p}_{\text{unigram}}(z))^{\beta}$ into the numerator $u(xyz)$ (check (11) to see that this is true). This means that in the special case where $\beta = 1$ and $U = V = 0$, we simply have $u(xyz) = \hat{p}_{\text{unigram}}$, so that the LOGLIN model gives exactly the same probabilities as the add-1 smoothed unigram model $\hat{p}_{\text{unigram}}$. However, by training the parameters, we might learn to trust the unigram model less ($0 < \beta < 1$) and rely more on the word embeddings ($U, V \neq 0$) to judge which words z are likely in the context xy.

A simpler way to implement this scheme is to define

$$f_{\text{unigram}}(xyz) = \log(c(z) + 1) \quad \text{(where } c(z) \text{ is the count of } z \text{ in training data)} \tag{29}$$

This gives the same model, since $\hat{p}_{\text{unigram}}(z)$ is just $c(z) + 1$ divided by a constant, and our model renormalizes $u(xyz)$ by a constant anyway.

Try throwing this feature into your LOGLIN model. What weight $\beta = \theta_{\text{unigram}}$ do you learn from `en.1K`? Does this new feature help cross-entropy? Does it help text categorization performance?

(g) Now you get to have some fun! Make some improvements to LOGLIN and report the effect on its performance. *You should make at least one non-trivial improvement;* you can do more for *extra credit.*

Measure cross-entropy and text categorization accuracy on development data (at least language ID, and optionally gen/spam). There will be (edible) prizes for the best-performing or most interesting systems.

Here are some ideas—most of them are not very hard, although training may be slow. Or you could come up with your own!

i. As suggested in the previous part, try adding a binary feature f_w for each word in the vocabulary, such that $f_w(xyz)$ is 1 if $z = w$ and 0 otherwise. Does this work better than the log-unigram feature from question 6f? Now try also adding a binary feature for each bigram and trigram that appears at least 3 times in training data. How good is the resulting model?

In all cases, you will want to tune C on development data to prevent overfitting. This is important—the original model had only $2d^2 + 1$ parameters where d is the dimensionality of the embeddings, but your new model has enough parameters that it can easily overfit the training data. In fact, if $C = 0$, the new model will *exactly* predict the unsmoothed probabilities, as if you were not smoothing at all (add-0)! The reason is that the maximum of the concave function $F(\vec{\theta}) = \sum_{i=1}^{N} F_i(\vec{\theta})$ is achieved when its partial derivatives are 0. So for *each* unigram feature f_w defined

in the previous paragraph, we have, from equation (23) with $C = 0$,

$$\frac{\partial F(\vec{\theta})}{\partial \theta_w} = \sum_{i=1}^{N} \frac{\partial F_i(\vec{\theta})}{\partial \theta_w} \tag{30}$$

$$= \underbrace{\sum_{i=1}^{N} f_w(xyz)}_{\text{observed count of } w \text{ in corpus}} \underbrace{- \sum_{i=1}^{N} \sum_{z'} \hat{p}(z' \mid xy) f_w(xyz')}_{\text{predicted count of } w \text{ in corpus}} \tag{31}$$

Hence SGD will adjust $\vec{\theta}$ until this is 0, that is, until the predicted count of w *exactly* matches the observed count $c(w)$. For example, if $c(w) = 0$, then SGD will try to allocate 0 probability to word w in all contexts (no smoothing), by driving $\theta_w \to -\infty$. Taking $C > 0$ prevents this by encouraging θ_w to stay close to 0.

ii. The embeddings were automatically computed based on which words tend to appear near one another. They don't consider how the words are *spelled*! So, augment each word's embedding with additional dimensions that describe properties of the spelling. For example, you could have dimensions that ask whether the word ends in `-ing`, `-ed`, etc. Each dimension will be 1 or 0 according to whether the word has the relevant property.

Just throw in a dimension for each suffix that is common in the data. You could also include properties relating to word length, capitalization patterns, vowel/consonant patterns, etc.—anything that you think might help!

You could easily come up with thousands of properties in this way. Fortunately, a given word such as `burgeoning` will have only a few properties, so the new embeddings will be *sparse*. That is, they consist mostly of 0's with a few nonzero elements (usually 1's). This situation is very common in NLP. As a result, you don't need to store all the new dimensions: you can compute them on demand when you are computing summations like $\sum_{j=1}^{d} \sum_{m=1}^{d} V_{jm} \cdot (y_j z_m)$ in (15). In such a summation, j ranges over possible suffixes of y and m ranges over possible suffixes of z (among other properties, including the original dimensions). To compute the summation, you only have to loop over the few dimensions j for which $y_j \neq 0$ and the few dimensions m for which $z_m \neq 0$. (All other summands are 0 and can be skipped.)

It is easy to identify these few dimensions. For example, `burgeoning` has the `-ing` property but not any of the other 3-letter-suffix properties. In the trigram $xyz =$ `demand was burgeoning`, the summation would include a feature weight V_{jm} for $j =$ `-was` and $m =$ `-ing`, which is included because yz has that particular pair of suffixes and so $y_j v_m = 1$. In practice, V can be represented as a hash map whose keys are pairs of properties, such as pairs of suffixes.

iii. If you can find online dictionaries or other resources, you may be able to obtain other, linguistically interesting properties of words. You can then proceed as with the spelling features above.

iv. Oddly, (13) only includes features that evaluate the *bigram* yz (via weights in the V matrix) and the *skip-bigram* xz (via weights in the U matrix). After all, you can see in (15) that the features have the form $y_j z_m$ and $x_j z_m$. This seems weaker than ADDL_BACKOFF. Thus, add unigram features of the form z_m and trigram features of the form $x_h y_j z_m$.

v. For a log-linear model, there's no reason to limit yourself to trigram context. Why not look at 10 previous words rather than 2 previous words? In other words, your language model can use the estimate $p(w_i \mid w_{i-10}, w_{i-9}, \ldots w_{i-1})$.
There are various ways to accomplish this. You may want to reuse the U matrix at all positions $i-10, i-9, \ldots, i-2$ (while still using a separate V matrix at position $i-1$). This means that having the word "bread" anywhere in the recent history (except at position w_{i-1}) will have the same effect on w_i. Such a design is called "tying" the feature weights: if you think of different positions having different features associated with them, you are insisting that certain related features have weights that are "tied together" (i.e., they share a weight).
You could further improve the design by saying that "bread" has weaker influence when it is in the more distant past. This could be done by redefining the features: for example, in your version of (15), you could scale down the feature value $(x_j z_m)$ by the number of word tokens that fall between x and z.[10]

vi. Since words tend to repeat, you could have a feature that asks whether w_i appeared in the set $\{w_{i-10}, w_{i-9}, \ldots w_{i-1}\}$. This feature will typically get a positive weight, meaning that recently seen words are likely to appear again. Since 10 is arbitrary, you should actually include similar features for several different history sizes: for example, another feature asks whether w_i appeared in $\{w_{i-20}, w_{i-19}, \ldots w_{i-1}\}$.

vii. Recall that (28) included the log-probability of another model as a feature within your LOGLIN model. You could include other log-probabilities in the same way, such as smoothed bigram or trigram probabilities. The LOGLIN model then becomes an "ensemble model" that combines the probabilities of several other models, learning how strongly to weight each of these other models.
If you want to be fancy, your log-linear model can include various trigram-model features, each of which returns $\log \hat{p}_{\text{trigram}}(z \mid xy)$ but only when $c(xy)$ falls into a particular range, and returns 0 otherwise. Training might learn different weights for these features. That is, it might learn that the trigram model is trustworthy when the context xy is well-observed, but not when it is rarely observed.

7. 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)

[10] A fancier approach is to *learn* how much to scale down this influence. For example, you could keep the feature value defined as $(x_j z_m)$, but say that the feature weights for position $i-6$ (for example) are given by the matrix $\lambda_6 U$. Now U is shared across all positions, but the various multipliers such as λ_6 are learned by SGD along with the entries of U and V. If you learn that λ_6 is close to 0, then you have learned that w_{i-6} has little influence on w_i. (In this case, the model is technically log-quadratic rather than log-linear, and the objective function is no longer concave, but SGD will probably find good parameters anyway. You will have to work out the partial derivatives with respect to the entries of λ as well as U and V.)

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)?

8. 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. Note that these documents contain special "words" `<s>` and `</s>` (actually XML tags) that enclose each utterance. These should be treated as actual words, distinct from the BOS and EOS symbols that implicitly enclose each sequence $\vec{w}$.

   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 sentence delimiters `<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?[11] Finally, $0.500 = \frac{4}{8}$ is the **word error rate** of this transcription, which had 4 errors against the 8-word true transcription on the first line of the file.[12]

   (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?

[11] Actually, the real estimate was 15 times as large. Speech recognizers are really rather bad at estimating $\log p(U \mid \vec{w})$, 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.

[12] 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.

(*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 (not counting `<s>` and `</s>`).

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 (plus 2 for `<s>` and `</s>`). Alternatively, you could read the whole line at once and break it up into an array of whitespace-delimited strings.
- 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. Thus, just throw away the rest of the line! (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!

---

9. *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 $\sum_z p(z \mid xy) = 1$

10. *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 6 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 $V - T()$.[13] What is the simple formula for $\alpha()$?

(d) Now let's consider $\alpha(xy)$, which is to be set so that $\sum_z p(z \mid xy) = 1$. From (4) we see that

$$\sum_z \hat{p}(z \mid xy) = \left(\sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy) \right) + \left(\alpha(xy) \cdot \sum_{z:\, c(xyz)=0} \hat{p}(z \mid y) \right) \tag{32}$$

$$= \left(\sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy) \right) + \alpha(xy) \cdot \left(1 - \sum_{z:\, c(xyz)>0} \hat{p}(z \mid y) \right) \tag{33}$$

To make $\sum_z \hat{p}(z \mid xy) = 1$, solving the equation shows that you will need[14]

$$\alpha(xy) = \frac{1 - \sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy)}{1 - \sum_{z:\, c(xyz)>0} \hat{p}(z \mid y)} \tag{34}$$

Got that? Now, step (33) above assumed that $\sum_z \hat{p}(z \mid y) = 1$. Give a formula for $\alpha(y)$ that ensures this. The formula will be analogous to the one we just derived for $\alpha(xy)$. (*Hint:* You'll want to sum over z such that $c(yz) > 0$.)

(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 $\sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy)$ 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 any z such that $c(xyz) > 0$. That allows the following simplification: $\sum_{z:\, c(xyz)>0} p(z \mid y) = \sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid y) = \frac{\sum_{z:\, c(xyz)>0} c(yz)}{c(y)+T(y)}$.
(*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 $\sum_{z:\, c(xyz)>0} c(yz)$ 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

[13] 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 certainly possible for $c(\text{OOV}) > 0$, since our vocabulary omits rarely observed words, treating them as OOV when they appear in training.

[14] 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.

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 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.

11. *Extra credit:* Problem 10a 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.
