601.465/665 — Natural Language Processing
Assignment 3: Smoothed Language Modeling

Prof. Jason Eisner — Fall 2018
Due date: Friday 5 October, 2 pm

Probabilistic models are an indispensable part of modern NLP. 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. See reading section A.

You now know enough about probability to build and use some trigram language models. You will experiment with different types of smoothing. You will also 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.

Reading: Read the handout attached to the end of this assignment!

Collaboration: ***You may work in teams 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: if you didn't work on a question, you don't deserve credit for it! Your solutions should emerge from collaborative real-time discussions with both of you present.

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. 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 an upper-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 wasting time on programming issues, 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$^{\text{TM}}$.

How to hand in your written work: Via Gradescope as before. Besides the comments you embed in your source files, put all other notes, documentation, and answers to questions in a `README.pdf` file.

How to test and hand in your code:

✍$_1$

- Your code and your trained parameters will need to be placed into a single submission directory, which you will zip and upload separately on Gradescope. We will post more detailed instructions on Piazza.

[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.

- For the parts where we tell you exactly what to do, an autograder will check that you got it right.
- For the open-ended challenge, an autograder will run your system and score its accuracy on "dev-test" and "final-test" data.
  - You should get a decent grade if you do at least as well as the "baseline" system provided by the TAs. Better systems will get higher grades.
  - Each time you submit a version of your system, you'll be able to see the results on "dev test" data and how they compare to the baseline system. You'll also be able to see what other teams tried and how their accuracies compared to yours. (Teams will use made-up names, not real names.)
  - The "dev-test" results are intended to help you develop your system. Grades will be based on the "final-test" data that you have never seen.

1. **Your starting point is the sample program `fileprob`.**

   It can be found on the ugrad machines (`ugrad{1,2,...,24}.cs.jhu.edu`) in the directory

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

   There are subdirectories corresponding to different programming languages. Choose one to copy as you like. Or, port to a new language.

   You're welcome to develop your code directly on these machines, if you want to avoid transferring the data.

   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 (`speech/{sample1,sample2,sample3}`). Try it!

   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. See reading sections B and C for more information about the datasets.

   ☞2 If a language model is built from the `speech/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.)

   ☞3 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. The two programs should share code for smoothed language models, so that when you write new smoothing methods later, they will immediately be available from both programs. 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 almost exactly like `fileprob`. However, as additional arguments,

- training needs to specify *two* training corpora rather than one: *train1* and *train2*.
- testing needs to specify the prior probability of *train1*.

For example, you could train your system with a line like

```
textcat TRAIN add1 words-10.txt gen spam
```

which saves the trained models in a file but prints no output. In this example, `gen` and `spam` are the training corpora, corresponding to "genuine" and spam emails. `words-10.txt` is a lexicon containing word vectors, which will used by some later smoothing methods (see problem 6) but is ignored for now.

Then you would test like this:

```
textcat TEST add1 words-10.txt gen spam 0.7 foo.txt bar.txt baz.txt
```

which loads the models from before and uses them to classify the remaining files. It should print output that labels each file with a training corpus name (in this case `gen` or `spam`):

```
spam    foo.txt
spam    bar.txt
gen     baz.txt
1 files were more probably gen (33.33%)
2 files were more probably spam (66.67%)
```

In other words, it classifies each file by printing its maximum *a posteriori* class (the file name of the training corpus that probably generated it). Then it prints a summary.

The number `0.7` on the test command line specifies your prior probability that a test file will be `gen`. (See reading section C.3.)

Please use the exact output formats above. If you would like to print any additional output lines for your own use, please begin each additional line with a `#` symbol to indicate that it is a comment. This will tell the autograder to ignore the additional lines.

As reading section D explains, 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 add-λ 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 tasks. 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!)**

   Using add-1 smoothing, run `textcat` on all the dev data for your chosen task:[2]

[2] 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.

- For the language ID task, 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. Use 0.7 as your prior probability of English.
- Or, for the spam detection task, classify the files `gen_spam/dev/gen/*` using the training corpora `gen` and `spam`.
  Then classify `gen_spam/dev/spam/*` similarly. Use 0.7 as your prior probability of `gen`.

To simplify the wording below, I'll assume that you've chosen the spam detection task.

☞4 (a) From the results, you should be able to compute a total error rate for the technique. That is, what percentage of the dev files were classified incorrectly?

☞5 (b) How small do you have to make the prior probability of `gen` before `textcat` classifies *all* the dev files as `spam`?

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

☞6 What is the minimum cross-entropy per token that you can achieve on the `gen` development files (when estimating a model from `gen` training files with add-λ smoothing)? How about for `spam`?

Note: To check that you are smoothing correctly, the autograder will run your code on small training and testing files.

(d) In principle, you could apply different amounts of smoothing to the `gen` and `spam` models, and this might be wise—for example, if their training sets have different rates of novel words.

However, for simplicity, your `textcat` program in this assignment smooths both models
☞7 in exactly the same way. So what is the minimum cross-entropy per token that you can
achieve on all development files together, if both models are smoothed with the *same* λ?

(To measure cross-entropy per token, find the *total* number of bits that it takes to predict all of the development files from their respective models. This means running `fileprob` twice: once for the `gen` data and once for the `spam` data.[3] Add the two results, and then divide by the *total* number of tokens in all of the development files.)

☞8 What value of λ gave you this minimum cross-entropy? Call this λ^*. (See reading section E for why you are using cross-entropy to select λ^*.)

(e) Each of the dev 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).

Come up with some way to quantify or graph the relation between file length and the
classification accuracy of add-λ^* on development data. (Feel free to use Piazza to discuss
☞9 how to do this.) Write up your results. To include graphs, see http://cs.jhu.edu/~jason/465/hw-lm/graphing.html.

[3] This is not quite the right thing to do, actually. Running `fileprob` on `gen` uses only a vocabulary derived from `gen`, whereas `textcat` is going to use a larger vocabulary derived from `gen` $\cup$ `spam`. If this were a research paper, I'd insist on using `textcat`'s vocabulary to tune λ^*. But for this assignment, take the shortcut and just run `fileprob`.

☞10 (f) Now try increasing the amount of *training* data. (Keep using add-λ^*, for simplicity.) Compute the overall error rate on dev data for training sets of different sizes. Graph the training size versus classification accuracy.

- For the language ID task, 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 task, 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`.

4. Reading section F gives an overview of several smoothing techniques beyond add-λ.

☞11 (a) At the end of question 2, V was 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?

☞12 (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.)

☞13 (c) Let's see on paper how backoff behaves with novel trigrams. If $c(xyz) = c(xyz') = 0$, then does it follow that $\hat{p}(z \mid xy) = \hat{p}(z' \mid xy)$ when those probabilities are estimated by BACKOFF_ADDL smoothing? In your answer, work out and state the value of $\hat{p}(z \mid xy)$ in this case. How do these answers change if $c(xyz) = c(xyz') = 1$?

☞14 (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.

Add support for add-λ smoothing with backoff, ADDL_BACKOFF. (This should allow both `fileprob` and `textcat` to use that method.) See the `INSTRUCTIONS` file for language-specific instructions.

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?? Figure it out!

You will submit trained add-λ^* models as in question 3d. For simplicity, just use the same λ^* as in that question, even though some other λ might work better with backoff.

6. (a) Add support for the LOGLIN model. See the `INSTRUCTIONS` file for language-specific instructions. Just as add-0.01 smoothing was selected by passing the model name `add0.01` to `fileprob` and `textcat`, a log-linear model with $C = 1$ should be selected by passing the model name `loglin1`. (C is the regularization coeffcient used during training: see reading section H.1.)

Your code will need to compute $\hat{p}(z \mid xy)$ using the features in reading section F.5.1. It should refer to the current parameters $\vec{\theta}$ (the entries of the X and Y matrices).

You can use embeddings of your choice from the `lexicons` directory. (See the `README` file in that directory. Make sure to use word embeddings for `gen`/`spam`, but character embeddings for `english`/`spanish`.) These word embeddings were derived from Wikipedia, a large diverse corpus with lots of useful evidence about the usage of many English words.

(b) Implement stochastic gradient ascent (Algorithm 1 in reading section H.2) to find the X and Y matrices that maximize $F(\vec{\theta})$. See the `INSTRUCTIONS` file in the code directory for details. For the autograder's sake, when `loglin` is specified on the command line, please train for $E = 10$ epochs, use the exact hyperparameters suggested in reading section I.1, and print output in the following format (this is printing $F(\vec{\theta})$ rather than $F_i(\vec{\theta})$):

```
Training from corpus en.1K
Vocabulary size is 30 types including OOV and EOS
epoch 1: F=-1.83284
epoch 2: F=-1.64174
  ...  [you should print these epochs too]
epoch 10: F=-1.58733
Finished training on 992 tokens
```

(c) Training a log-linear model takes significantly more time than ADDL smoothing. Therefore, we recommend that you experiment with the language ID task for this part.

Try your program out first by training a log-linear language model on `en.1K`, with character embeddings $d = 10$ (`chars-10.txt`) and regularization strength $C = 1$. As a check, your numbers should match those shown in 6b above.

You should now be able to measure cross-entropies and text categorization error rates under your fancy new language model! `textcat` 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.

15 Report cross-entropy and text categorization accuracy with $C = 1$, but 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?

16 (d) Now you get to have some fun! Add some new features to LOGLIN and report the effect on its performance. Some possible features are suggested in reading section J. *You should make at least one non-trivial improvement;* you can do more for *extra credit*, including varying hyperparameters and training protocols (reading sections I.1 and I.5).

Your improved method should be selected by using the command-line argument `improved` (in place of `add1`, `loglin1`, etc.). You will submit your system to Gradescope for autograding, including the trained model.

You are free to submit many versions of your system—with different implementations of `improved`. All will show up on the leaderboard, with comments, so that you and your classmates can see what works well. For final grading, the autograder will take the submitted version of your system that worked best on dev-test data, and then evaluate its performance on final-test data.

(e) *Extra credit:* In your submission, you can also include a trained model for the spam detection task. This doesn't require very much extra work in principle, but training will be much slower because of the larger vocabulary (slowing down $\sum_{z'}$) and the larger training corpus (slowing down $\sum_i$). You may need to adjust C, using development data as usual. See `lexicons/README` for various lexicons of embeddings that you could try. To speed up training, you could try a smaller training set, a smaller vocabulary, or a lower-dimensional embedding. Report what you did.

7. Finally, we turn briefly to speech recognition. In this task, 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?[4] 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.[5]

17 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.)

[4] 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.

[5] 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.

Extra Credit

8. *Extra credit:* Actually implement the speech recognition selection method in question 7, using one of the language models you've already built.

   (a) 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 words-10.txt 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`.)

   ✌18 (b) 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.

   ✌19 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.

9. *Extra credit:* We have been assuming a finite vocabulary by replacing all unknown words with a special OOV symbol. But an alternative is an open-vocabulary language model (reading section D.5).

   ✌$_{20}$ 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 in reading section F.4 is conceptually pretty simple. The tricky part will be finding the right α values to make the probabilities sum to 1. You have to work out a few formulas, and then rearrange the loops in order to reduce a large runtime to $O(N)$.

    ✌$_{21}$ (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.

    ✌$_{22}$ (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?

    ✌$_{23}$ (c) Observe that the set of zero-count words $\{z : c(z) = 0\}$ has size $V - T()$.[6] 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 (7) we see

[6] 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.

that

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

$$= \left(\sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy)\right) + \alpha(xy) \cdot \left(1 - \sum_{z:\, c(xyz)>0} \hat{p}(z \mid y)\right) \tag{2}$$

To make $\sum_z \hat{p}(z \mid xy) = 1$, solving the equation shows that you will need[7]

$$\alpha(xy) = \frac{1 - \sum_{z:\, c(xyz)>0} p_{\text{disc}}(z \mid xy)}{1 - \sum_{z:\, c(xyz)>0} \hat{p}(z \mid y)} \tag{3}$$

✌24 Got that? Now, step (2) 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 . . .

✌25 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.

✌26 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.

✌27 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 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 $\hat{p}(z \mid xy)$ for all possible trigrams xyz. However, since there are V^3 possible trigrams, that would take a lot of time and memory. Instead,

[7] 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.

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.

✌$_{28}$ (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.

✌$_{29}$ (g) If you're curious, implement Witten-Bell backoff using the techniques above. How does this smoothing method (which does not use any λ) affect your error rate compared to add-λ backoff?

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

- Write a loop that checks that $\sum_z \hat{p}(z \mid xy) = 1$ 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.

(h) 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. ✌$_{30}$

601.465/665 — Natural Language Processing
Reading for Assignment 3: Smoothed Language Modeling

Prof. Jason Eisner — Fall 2018

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies assignment 3, which refers to it.

A Are trigram models useful?

Why build n-gram models when we know they are a poor linguistic theory? Answer: A linguistic system without statistics is often fragile, and may break when run on real data. It will also be unable to resolve ambiguities. So our first priority is to get some numbers into the system somehow. An n-gram model is a starting point, and may get reasonable results even though it doesn't have any real linguistics yet.

Speech recognition. Speech recognition systems have made heavy use of trigram models for decades. Alternative approaches that *don't* look at the trigrams do worse. One can do better by building fancy language models that *combine* trigrams with syntax, topic, and so on. But only a little better—dramatic improvements over trigram models are hard to get. In the language modeling community, a rule of thumb was that you had enough for a Ph.D. dissertation if you had managed to reduce a standard trigram model's perplexity per word by 10% (equivalent to a cross-entropy reduction of just 0.152 bits per word).

Machine translation. In the same way, machine translation (MT) systems have often included 5-gram models trained on quite massive amounts of data. 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.

Why doesn't a speech recognition system need 5-grams? Because it is not generating a new sentence. It 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.

Neural language models. In the 2010's, language models based on recurrent neural networks finally started to show dramatic gains over trigram models. These neural language models are much slower to train, but they are not limited to trigrams, and can learn to notice complex ways in which the context affects the next word. We won't quite be covering them in this course, but the starting point is the word embeddings used in this assignment and the previous one.

Neural language models have become increasingly popular since the mid-2010's. However, cleverly smoothed 7-gram models can still do about as well by looking at lots of features of the previous 6-gram, according to Pelemans et al. (2016).

B Boundary symbols

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, $w_N = \textsc{eos}$ ("end of sequence"), while for $i < 1$, $w_i = \textsc{bos}$ ("beginning of sequence"). Thus, $\vec{w}$ consists of $N-1$ words plus an EOS symbol. Notice that we do not generate BOS but condition on it (it was always there). Conversely, we do generate EOS but never condition on it (nothing follows it). The boundary symbols $\textsc{bos}, \textsc{eos}$ are special symbols that do not appear among $w_1 \ldots w_{N-1}$.

In assignment 3, we will consider every *file* to implicitly start with BOS and end with EOS. A file might be a sentence, or an email message, or a fragment of text.

Some files also contain sentence boundary symbols `<s>` and `</s>`. You should consider these to be just ordinary words.

C Datasets for Assignment 3

Assignment 3 will have corpora for three tasks: language identification, spam detection, and speech recognition. Each corpus has a `README` file that you should look at.

C.1 The train/dev/test split

Each corpus has already been divided for you into training, development, and test sets, which are in separate directories.

You will collect counts on the training set, tune the "hyperparameters" like λ to maximize performance on the development set, and then evaluate your performance on the test set.

In this case, we actually have two test sets.

- The "dev test" set is for use as you develop your system. If you experiment on it repeatedly, there is a danger that you will "overfit" to it—that is, you might find your way to a method that seems really good, but is actually only good for that particular dataset, not in general.
- Thus, for fairness, we have to find out whether your system can do well in a blind test, when we run it on data you've never seen. Your grade will therefore be determined based on a new "final test" set.

C.2 Domain mismatch

In this assignment, the training set is unfortunately a bit different from the others. To simplify the command-line syntax for you, I've assumed that the training set consists of a *single* big file. That means that there is only one BOS and one EOS in the whole training set.

By contrast, BOS and EOS are much more common in the dev set and the test sets. 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—because it has seen only one file!

C.3 Class ratios

In the assignment, you'll have to specify a prior probability that a file will be genuine email (rather than spam) or English (rather than Spanish). In other words, how often do you expect the real world to produce genuine email or English in your test data? We will ask you to guess 0.7.

Of course, your system won't know the true fraction on test data, because it doesn't know the true classes—it is trying to predict them.

We can try to estimate the fraction from training data. It happens that $\frac{2}{3}$ of the documents are genuine email, and $\frac{1}{2}$ are English.[1] In this case, the prior probability is a parameter of the model, to be estimated from training data (with smoothing!) like any other parameter.

But if you think that test data might have a different rate of spam or Spanish than training data, then the prior probability is not necessarily something that you should represent within the model and estimate from training data. Instead it can be used to represent your personal guess about what you think test data *will* be like.

Indeed, in the assignment, you'll use training data only to get the smoothed language models, which define the likelihood of the different classes. This leaves you free to specify your prior probability of the classes on the command line. This setup would let you apply the system to different test datasets about which you have different prior beliefs—the spam-infested email account that you abandoned, versus your new private email account that only your family knows about.

Does it seem strange to you that a guess or assumption might have a role in statistics? That is actually central to the Bayesian view of statistics—which says that you can't get something for nothing. Just as you can't get theorems without assuming axioms, you can't get posterior probabilities without assuming prior probabilities.

D The vocabulary

D.1 Choosing a finite vocabulary

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.

[1] These numbers are actually derived from dev data rather than training data, because of the domain mismatch issue above—the training data are not divided into documents.

D.2 Consequences for evaluating a model

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]

D.3 Comparing apples to apples

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 $p_{\text{model1}}(\texttt{i saw } \textsc{oov} \texttt{ on the tv})$ 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 $p(\texttt{the} \mid \textsc{oov}, \texttt{on})$, for example by backing off to $p(\texttt{the} \mid \texttt{on})$.

Similarly, the smoothing method must give a reasonable (non-zero) probability to $p(\textsc{oov} \mid \texttt{i}, \texttt{saw})$. 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.

D.4 How to choose the vocabulary

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.

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

D.5 Open-vocabulary language modeling

In this assignment, we assume a fixed finite vocabulary. However, an *open-vocabulary* language model does not limit in advance to a finite vocabulary. Question 9 (extra credit) explores this possibility.

An open-vocabulary model must be able to assign positive probability to any word—that is, to any string of letters that might ever arise. If the *alphabet* is finite, you could do this with a letter n-gram model!

Such a model is sensitive to the spelling and length of the unknown word. Longer words will generally receive lower probabilities, which is why it is possible for the probabilities of all unknown words to sum to 1, even though there are infinitely many of them. (Just as $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots=1$.)

E Performance metrics

In this assignment, you will measure your performance in two ways. To measure the predictive power of the model, you will use cross-entropy (per token). To measure how well the model does at a task, you will use error rate (per document). In both cases, smaller is better.

Error rate may be what you really care about! However, it doesn't give a lot of information on a small dev set. If your dev set has only 100 documents, then the error rate can only be one of the numbers $\{\frac{0}{100},\frac{1}{100},\ldots,\frac{100}{100}\}$. It can tell you if your changes helped by correcting a wrong answer. But it can't tell you that your changes were "moving in the right direction" by merely increasing the probability of right answers.

In particular, for some of the tasks we are considering here, the error rate is just not very sensitive to the smoothing parameter λ: there are many λ values that will give the same integer number of errors on dev data. That is why you will use cross-entropy to select your smoothing parameter λ on dev data: it will give you clearer guidance.

E.1 Other possible metrics

As an alternative, could you devise a continuously varying version of the error rate? Yes, because our system[3] doesn't merely compute a single output class for each document. It constructs a probability distribution over those classes, using Bayes' Theorem. So we can evaluate whether that distribution puts high probability on the correct answer.

- One option is the *expected error rate*. Suppose document #1 is `gen`. If the system thinks $p(\texttt{gen} \mid \textit{document}_1) = 0.49$, then sadly the system will output `spam`, which ordinary error rate would count as 1 error. But suppose you pretend—just for evaluation purposes—that the system chooses its output randomly from its posterior distribution ("stochastic decoding" rather than "MAP decoding"). In that case, it only has probability 0.51 of choosing `spam`, so the *expected* number of errors on this document is only 0.51. Partial credit!

  Notice that expected error rate gives us a lot of credit for increasing $p(\texttt{gen} \mid \textit{document}_1)$ from 0.01 to 0.49, and little additional credit for increasing it to 0.51. By contrast, the actual error rate *only* gives us credit for the increase from 0.49 to 0.51, since that's where the actual system output would change.

[3] Unlike rule-based classifiers and some other ML classifiers.

- Another continuous error metric is the *log-loss*, which is the system's expected surprisal about the correct answer. The system's surprisal on document 1 is $-\log_2 p(\texttt{gen} \mid \textit{document}_1) = -\log_2 0.49 = 1.03$ bits.

  Both expected error rate and log-loss are averages over the documents that are used to evaluate. So document 1 contributes 0.51 errors to the former average, and contributes 1.03 bits to the latter average.

  In general, a single document contributes a number in $[0, 1]$ to the expected error rate, but a number in $[0, \infty]$ to the log-loss. In particular, a system that thinks that $p(\texttt{gen} \mid \textit{document}_1) = 0$ is infinitely surprised by the correct answer ($-\log_2 0 = \infty$). So optimizing for log-loss would dissuade you infinitely strongly from using this system . . . basically on the grounds that a system that is completely confident in even one wrong answer can't possibly have the correct probability distribution. To put it more precisely, if the dev set has size 100, then changing the system's behavior on a single document can change the error rate or the expected error rate by at most $\frac{1}{100}$—after all, it's just one document!—whereas it can change the log-loss by an *unbounded* amount.

What is the relation between the log-loss and cross-entropy metrics? They are both average surprisals.[4] However, they are very different:

metric	what it evaluates	probability used	units	long docs count more?
log-loss	the whole classification system	$p(\texttt{gen} \mid \textit{document}_1)$	bits per document	no
cross-entropy	the `gen` model within the system	$p(\textit{document}_1 \mid \texttt{gen})$	bits per `gen` token	yes

E.2 Generative vs. discriminative

The error rate, expected error rate, and log-loss are all said to be *discriminative metrics* because they measure how well the system discriminates between correct and incorrect classes. By contrast, the cross-entropy is a *generative metric* because it measures how good the system would be at randomly generating the observed data.

Methods for setting hyperparameters or parameters are said to be generative or discriminative according to whether they optimize a generative or discriminative metric.

A generative model includes a probability distribution $p(\text{input})$ that accounts for the input data. Thus, this assignment uses generative models (language models). A discriminative model only tries to predict output from input, possibly using $p(\text{output} \mid \text{input})$. Thus, a conditional log-linear model for text classification would be discriminative. A discriminative model or training procedure focuses less on explaining the input data and more on solving a particular task—less science, more engineering.

F Smoothing techniques

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

[4] Technically, you could regard the log-loss as a *conditional cross-entropy* . . . to be precise, it's the conditional cross-entropy between empirical and system distributions over the *output* class. By contrast, the metric you'll use on this assignment is the cross-entropy between empirical and system distributions over the *input* text. The output and the input are different random variables, so log-loss is quite different from the cross-entropy we've been using to evaluate a language model!

F.1 Uniform distribution (UNIFORM)

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

$$\hat{p}(z \mid xy) = 1/V \tag{4}$$

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

F.2 Add-λ (ADDL)

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

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda}{c(xy) + \lambda V} \tag{5}$$

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

F.3 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':

$$\hat{p}(z \mid xy) = \frac{c(xyz) + \lambda V \cdot \hat{p}(z \mid y)}{c(xy) + \lambda V} \tag{6}$$

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?? Figure it out!

F.4 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.

$$\hat{p}(z \mid xy) = \begin{cases} p_{\text{disc}}(z \mid xy) & \text{if } c(xyz) > 0 \\ \alpha(xy)\hat{p}(z \mid y) & \text{otherwise} \end{cases} \tag{7}$$

$$\hat{p}(z \mid y) = \begin{cases} p_{\text{disc}}(z \mid y) & \text{if } c(yz) > 0 \\ \alpha(y)\hat{p}(z) & \text{otherwise} \end{cases} \tag{8}$$

$$\hat{p}(z) = \begin{cases} p_{\text{disc}}(z) & \text{if } c(z) > 0 \\ \alpha() & \text{otherwise} \end{cases} \tag{9}$$

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

$$p_{\text{disc}}(z \mid xy) = \frac{c(xyz)}{c(xy) + T(xy)} \tag{10}$$

$$p_{\text{disc}}(z \mid y) = \frac{c(yz)}{c(y) + T(y)} \tag{11}$$

$$p_{\text{disc}}(z) = \frac{c(z)}{c() + T()} \tag{12}$$

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. (You are defining probability distributions over the vocabulary, including OOV, so each of these summations will have V terms.) Question 10 (extra credit) challenges you to work out the details.

F.5 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.[5]

Given a trigram xyz, our model $\hat{p}$ is defined by

$$\hat{p}(z \mid xy) \overset{\text{def}}{=} \frac{u(xyz)}{Z(xy)} \tag{13}$$

where

$$u(xyz) \overset{\text{def}}{=} \exp\left(\sum_k \theta_k \cdot f_k(xyz)\right) = \exp\left(\vec{\theta} \cdot \vec{f}(xyz)\right) \tag{14}$$

$$Z(xy) \overset{\text{def}}{=} \sum_z u(xyz) \tag{15}$$

Here $\vec{f}(xyz)$ is the feature vector extracted from xyz, and $\vec{\theta}$ is the model's weight vector. $\sum_z$ sums over the V words in the vocabulary (including OOV) in order to ensure that you end up with a probability distribution over this chosen vocabulary. That is the goal of all these language models; you saw a similar $\sum_z$ in BACKOFF_WB.

[5] 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.

F.5.1 Bigrams and skip-bigram features from word embeddings

What features should we use in the log-linear model?

A natural idea is to use one binary feature for each specific unigram z, bigram yz, and trigram xyz (see reading section J.2 below).

Instead, however, let's start with the following model based on word embeddings:

$$u(xyz) \stackrel{\text{def}}{=} \exp\left(\vec{x}^\top X \vec{z} + \vec{y}^\top Y \vec{z}\right) \tag{16}$$

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

This model may be a little hard to understand at first, so here's some guidance.

What's the role of the word embeddings? Note that the language model is still defined as a conditional probability distribution over the *vocabulary*. The *lexicon*, which you will specify on the command line, is merely an external resource that lets the model look up some attributes of the vocabulary words. Just like the dictionary on your shelf, it may also list information about some words you don't need, and it may lack information about some words you do need. In short, the existence of a lexicon doesn't affect the interpretation of $\sum_z$ in (15): that formula remains the same regardless of whether the model's features happen to consult a lexicon!

For OOV, or for any other type in your vocabulary that has no embedding listed in the lexicon, your features should back off to the embedding of OOL—a special "out of lexicon" symbol that stands for "all other words." OOL *is* listed in the lexicon, just as OOV is included in the vocabulary.

Note that even if an specific out-of-vocabulary word is listed in the lexicon, you must not use that listing.[6] For an out-of-vocabulary word, you are supposed to be computing probabilities like $p(\text{OOV} \mid xy)$, which is the probability of the whole OOV class—it doesn't even mention the specific word that was replaced by OOV. (See reading section D.2.)

Is this really a log-linear model? Now, what's up with (16)? It's a valid formula: you can always get a probability distribution by defining $\hat{p}(z \mid xy) = \frac{1}{Z(xy)} \exp(\text{any function of } x, y, z \text{ that you like})$! But is (16) really a *log-linear* function? Yes it is! Let's write out those d-dimensional vector-matrix-vector multiplications more explicitly:

$$u(xyz) = \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} x_j X_{jm} z_m + \sum_{j=1}^{d}\sum_{m=1}^{d} y_j Y_{jm} z_m\right) \tag{17}$$

$$= \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} X_{jm} \cdot (x_j z_m) + \sum_{j=1}^{d}\sum_{m=1}^{d} Y_{jm} \cdot (y_j z_m)\right) \tag{18}$$

[6] This issue would not arise if we simply defined the vocabulary to be the set of words that appear in the lexicon. This simple strategy is certainly sensible, but it would slow down normalization because our lexicon is quite large.

This does have the log-linear form of (14). Suppose $d = 2$. Then implicitly, we are using a weight vector $\vec{\theta}$ of length $d^2 + d^2 = 8$, defined by

$$\begin{array}{ccccccccc} \langle\, \theta_1, & \theta_2, & \theta_3, & \theta_4, & \theta_5, & \theta_6, & \theta_7, & \theta_8 & \rangle \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ \langle X_{11}\,, & X_{12}\,, & X_{21}\,, & X_{22}\,, & Y_{11}\,, & Y_{12}\,, & Y_{21}\,, & Y_{22} & \rangle \end{array} \tag{19}$$

for a vector $\vec{f}(xyz)$ of 8 features

$$\begin{array}{ccccccccc} \langle f_1(xyz), & f_2(xyz), & f_3(xyz), & f_4(xyz), & f_5(xyz), & f_6(xyz), & f_7(xyz), & f_8(xyz) & \rangle \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ \langle\ x_1 z_1, & x_1 z_2, & x_2 z_1, & x_2 z_2, & y_1 z_1, & y_1 z_2, & y_2 z_1, & y_2 z_2 & \rangle \end{array} \tag{20}$$

Remember that the optimizer's job is to automatically manipulate some control sliders. This particular model with $d = 2$ has a control panel with 8 sliders, arranged in two $d \times d$ grids (X and Y). The point is that we can also refer to those same 8 sliders as $\theta_1, \ldots \theta_8$ if we like. What features are these sliders (weights) be connected to? The ones in (20): if we adopt those feature definitions, then our general log-linear formula (14) will yield up our specific model (18) (= (16)) as a special case.

As always, $\vec{\theta}$ is incredibly important: it determines all the probabilities in the model.

Is this a sensible model? The feature definitions in (20) are *pairwise products of embedding dimensions*. Why on earth would such features be useful? First imagine that the embedding dimensions were bits (0 or 1). Then $x_2 z_1 = 1$ iff ($x_2 = 1$ **and** $z_1 = 1$), so you could think of multiplication as a kind of *feature conjunction*. Multiplication has a similar conjunctive effect even when the embedding dimensions are in $\mathbb{R}$. For example, suppose $z_1 > 0$ indicates the degree to which z is a human-like noun, while $x_2 > 0$ indicates the degree to which x is a verb whose direct objects are usually human.[7] Then the product $x_2 z_1$ will be larger for trigrams like `kiss the baby` and `marry the policeman`. So by learning a positive weight X_{21} (nicknamed θ_3 above), the optimizer can drive $\hat{p}(\texttt{baby} \mid \texttt{kiss the})$ higher, at the expense of probabilities like $\hat{p}(\texttt{benzene} \mid \texttt{kiss the})$. $\hat{p}(\texttt{bunny} \mid \texttt{kiss the})$ might be somewhere in the middle since bunnies are a bit human-like and thus $\texttt{bunny}_1$ might be numerically somewhere between $\texttt{baby}_1$ and $\texttt{benzene}_1$.

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

$$\vec{\texttt{e}} = \begin{bmatrix} -.5 \\ 1 \end{bmatrix}, \quad \vec{\texttt{r}} = \begin{bmatrix} 0 \\ .5 \end{bmatrix}, \quad \vec{\texttt{s}} = \begin{bmatrix} .5 \\ .5 \end{bmatrix}, \quad X = \begin{bmatrix} 1 & 0 \\ 0 & .5 \end{bmatrix}, \quad Y = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

[7] You might wonder: What if the embedding dimensions don't have such nice interpretations? What if z_1 doesn't represent a single property like humanness, but rather a linear combination of such properties? That actually doesn't make much difference. Suppose z can be regarded as $M\tilde{z}$ where $\tilde{z}$ is a more interpretable vector of properties. (Equivalently: each z_j is a linear combination of the properties in $\tilde{z}$.) Then $x^\top X z$ can be expressed as $(M\tilde{x})^\top X (M\tilde{y}) = \tilde{x}^\top (M^\top X M)\tilde{y}$. So now it's $\tilde{X} = M^\top X M$ that can be regarded as the matrix of weights on the interpretable products. If there exists a good $\tilde{X}$ and M is invertible, then there exists a good X as well, namely $X = (M^\top)^{-1} \tilde{X} M^{-1}$.

First, we compute the unnormalized probability.

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

$$= \exp(-.5 \times 1 \times .5 \ + \ 1 \times .5 \times .5 \ + \ 0 \times 2 \times .5 \ + \ .5 \times 1 \times .5) = \exp 0.25 = 1.284$$

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

$$\hat{p}(\texttt{s} \mid \texttt{er}) \stackrel{\text{def}}{=} \frac{u(\texttt{ers})}{Z(\texttt{er})} = \frac{u(\texttt{ers})}{u(\texttt{era}) + u(\texttt{erb}) + \cdots + u(\texttt{erz})} = \frac{1.284}{1.284 + \cdots} \tag{21}$$

Speedup. The example illustrates that the denominator

$$Z(xy) = \sum_{z'} u(xyz') = \sum_{z'} \exp\left(\vec{x}^\top X \vec{z'} + \vec{y}^\top Y \vec{z'}\right) \tag{22}$$

is expensive to compute because of the summation over all z' in the vocabulary. Fortunately, you can compute $x^\top X z' \in \mathbb{R}$ simultaneously for all z'.[8] The results can be found as the elements of the row vector $x^\top X E$, where E is a $d \times V$ matrix whose columns are the embeddings of the various words z' in the vocabulary. This is easy to see, and computing this vector still requires just as many scalar multiplications and additions as before ... but we have now expressed the computation as a pair of vector-matrix mutiplications, $(x^\top X)E$, which you can perform using a matrix library (such as numpy or jblas). That can be considerably faster than looping over all z'. That is because the library call is highly optimized and exploits hardware support for matrix operations (e.g., parallelism).

F.6 Other smoothing schemes

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.)

In practical settings, the most popular n-gram smoothing scheme is something called modified Kneser-Ney. One can also use a more principled Bayesian method based on the hierarchical Pitman-Yor process—the result is very close to modified Kneser-Ney.

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 as accurately as a human can finish another human's sentence, it would go beyond the whole n-gram family to consider syntax, semantics, and topic. This is an active area of research that uses grammars, recurrent neural networks, and other techniques.

G Safe practices for working with log-probabilities

G.1 Use natural log for internal computations

In this assignment, as in most of mathematics, log means $\log_e$ (the log to base e, or natural log, sometimes written ln). This is also the standard behavior of the `log` function in most programming languages.

[8] The same trick works for $y^\top Y z'$, of course.

With natural log, the calculus comes out nicely, thanks to the fact that $\frac{\mathrm{d}}{\mathrm{d}Z} \log Z = \frac{1}{Z}$. It's only with natural log that the gradient of the log-likelihood of a log-linear model can be directly expressed as observed features minus expected features.

On the other hand, information theory conventionally talks about bits, and quantities like entropy and cross-entropy are conventionally measured in bits. Bits are the unit of $-\log_2$ probability. A probability of 0.25 is reported "in negative-log space" as $-\log_2 0.25 = 2$ bits. Some people do report that value more simply as $-\log_e 0.25 = 1.386$ nats. But it is more conventional to use bits as the unit of measurement. (The term "bits" was coined in 1948 by Claude Shannon to refer to "binary digits," and "nats" was later defined by analogy to refer to the use of natural log instead of log base 2. The unit conversion factor is $\frac{1}{\log 2} \approx 1.443$ bits/nat.)

Even if you are planning to *print* bit values, it's still wise to standardize on $\log_e$-probabilities for all of your *formulas, variables, and internal computations.* Why? They're just easier! If you tried to use negative $\log_2$-probabilities throughout your computation, then whenever you called the `log` function or took a derivative, you'd have to remember to convert the result. It's too easy to make a mistake by omitting this step or by getting it wrong. So the best practice is to do this conversion *only* when you print: at that point convert your $\log_e$-probability to bits by dividing by $-\log 2 \approx -0.693$. This is a unit conversion.

G.2 Avoid exponentiating big numbers (not necessary for this assignment)

Log-linear models require calling the exp function. Unfortunately, $\exp(710)$ is already too large for a 64-bit floating-point number to represent, and will generate a runtime error ("overflow"). Conversely, $\exp(-746)$ is too close to 0 to represent, and will simply return 0 ("underflow").

That shouldn't be a problem for this assignment if you stick to the language ID task. If you are experiencing an overflow issue there, then your parameters probably became too positive or too negative as you ran stochastic gradient descent. That could mean that your stepsize was too large, or more likely, that you have a bug in your computation of the gradient.

But to avoid these problems elsewhere—including with the spam detection task—standard trick is to represent all values "in log-space." In other words, simply store 710 and -746 rather than attempting to exponentiate them.

But how can you do arithmetic in log-space? Suppose you have two numbers p, q, which you are representing in memory by their logs, lp and lq.

- *Multiplication:* You can represent pq by its log, $\log(pq) = \log(p) + \log(q) = lp + lq$. That is, multiplication corresponds to log-space addition.

- *Division*: You can represent p/q by its log, $\log(p/q) = \log(p) - \log(q) = lp - lq$. That is, division corresponds to log-space subtraction.

- *Addition:* You can represent $p + q$ by its log. $\log(p + q) = \log(\exp(lp) + \exp(lq)) = \text{logsumexp}(lp, lq)$. See the discussion of logsumexp in the future Assignment 6 handout.

For training a log-linear model, you can work almost entirely in log space, representing u and Z in memory by their logs, $\log u$ and $\log Z$. In order to compute the expected feature vector in (27) below, you will need to come out of log space and find $p(z' \mid xy) = u'/Z$ for each word z'. But computing u' and Z separately is dangerous: they might be too large or too small. Instead, rewrite $p(z' \mid xy)$ as $\exp(\log u' - \log Z)$. Since $u' \leq Z$, this is exp of a negative number, so it will never *overflow*. It might *underflow* to 0 for some words z', but that's ok: it just means that $p(z' \mid xy)$ really *is* extremely close to 0, and so $\vec{f}(xyz')$ should make only a negligible contribution to the expected feature vector.

H Training a log-linear model

H.1 The training objective

To implement the conditional log-linear model, the main work is to train $\vec{\theta}$ (given some training data and a regularization coefficient C). As usual, you'll set $\vec{\theta}$ to maximize

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \frac{1}{N}\left(\underbrace{\left(\sum_{i=1}^{N} \log \hat{p}(w_i \mid w_{i-2}\ w_{i-1})\right)}_{\text{log likelihood}} - \underbrace{\left(C \cdot \sum_k \theta_k^2\right)}_{\text{L}_2 \text{ regularizer}}\right) \tag{23}$$

which is the regularized log-likelihood per word token. (There are N word tokens.)

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 on average 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).[9]

The regularization coefficient $C \geq 0$ can be selected based on dev data.

H.2 Stochastic gradient descent

Fortunately, concave functions like $F(\vec{\theta})$ in (23) are "easy" to maximize. You can 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 (23) as an average of local objectives $F_i(\vec{\theta})$ that each

[9] 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

$$\underbrace{p(\vec{w} \mid \vec{\theta})}_{\text{likelihood}} \cdot \underbrace{p(\vec{\theta})}_{\text{prior}} \tag{24}$$

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 (23) is just maximizing the log of (24). 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 $u(xyz)$.)

predict a single word, by moving the regularization term into the summation.

$$F(\vec{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \underbrace{\left(\log \hat{p}(w_i \mid w_{i-2}\ w_{i-1}) - \frac{C}{N}\cdot\sum_k \theta_k^2\right)}_{\text{call this } F_i(\vec{\theta})} \tag{25}$$

$$= \frac{1}{N}\sum_{i=1}^{N} F_i(\vec{\theta}) \tag{26}$$

The gradient of this average, $\nabla F(\vec{\theta})$, is therefore the *average* value of $\nabla F_i(\vec{\theta})$.

Algorithm 1 Stochastic gradient ascent

```
Input: Initial stepsize $\gamma_0$, initial parameter values $\vec{\theta}^{(0)}$, training corpus $\mathcal{D} = (w_1, w_2, \cdots, w_N)$,
    regularization coefficient $C$, number of epochs $E$
1: procedure SGATrain
2:   $\vec{\theta} \leftarrow \vec{\theta}^{(0)}$
3:   $t \leftarrow 0$                                   ▷ number of updates so far
4:   for $e : 1 \to E$ :                              ▷ do $E$ passes over the training data, or "epochs"
5:     for $i : 1 \to N$ :                            ▷ loop over summands of (25)
6:       $\gamma \leftarrow \frac{\gamma_0}{1 + \gamma_0 \cdot \frac{2C}{N} \cdot t}$      ▷ current stepsize—decreases gradually
7:       $\vec{\theta} \leftarrow \vec{\theta} + \gamma \cdot \nabla F_i(\vec{\theta})$           ▷ move $\vec{\theta}$ slightly in a direction that increases $F_i(\vec{\theta})$
8:       $t \leftarrow t + 1$
9:   return $\vec{\theta}$
```

Discussion. On each iteration, the algorithm picks some word i and pushes $\vec{\theta}$ in the direction $\nabla F_i(\vec{\theta})$, which is the direction that gets the fastest increase in $F_i(\vec{\theta})$. The updates from different i will partly cancel one another out,[10] but their *average* direction is $\nabla F(\vec{\theta})$, so 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; a theorem guarantees that the algorithm will converge to that maximum if allowed to run forever ($E = \infty$).

How far the algorithm pushes $\vec{\theta}$ is controlled by γ, known as the "step size" or "learning rate." This starts at γ_0, but needs to decrease over time in order to guarantee convergence of the algorithm. The rule in line 6 for gradually decreasing γ is the one recommended by Bottou (2012), "Stochastic gradient descent tricks," which you should read in full if you want to use this method "for real" on your own problems.

Note that t increases and the stepsize decreases on every pass through the inner loop. This is important because N might be extremely large in general. Suppose you are training on the whole web—then the stochastic gradient ascent algorithm should have essentially converged even before you finish the first epoch! See reading section I.5 for some more thoughts about epochs.

[10] For example, in the training sentence `eat your dinner but first eat your words`, $\nabla F_3(\vec{\theta})$ is trying to raise the probability of `dinner`, while $\nabla F_8(\vec{\theta})$ is trying to raise the probability of `words` (at the expense of `dinner`!) in the same context.

H.3 The gradient vector

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 (25). As you'll recall from the previous assignment, each partial derivative takes a simple and beautiful form[11]

$$\frac{\partial F_i(\vec{\theta})}{\partial \theta_k} = \underbrace{f_k(xyz)}_{\text{observed value of feature } f_k} - \underbrace{\sum_{z'} \hat{p}(z' \mid xy) f_k(xyz')}_{\text{expected value of feature } f_k, \text{ according to current } \hat{p}} - \underbrace{\frac{2C}{N}\theta_k}_{\text{pulls } \theta_k \text{ towards } 0} \tag{27}$$

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 ccording 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.

H.4 The gradient for the embedding-based model

When we use the specific model in (16), the feature weights are the entries of the X and Y matrices, as shown in (18). The partial derivatives with respect to these weights are

$$\frac{\partial F_i(\vec{\theta})}{\partial X_{jm}} = x_j z_m - \sum_{z'} \hat{p}(z' \mid xy) x_j z'_m - \frac{2C}{N} X_{jm} \tag{28}$$

$$\frac{\partial F_i(\vec{\theta})}{\partial Y_{jm}} = y_j z_m - \sum_{z'} \hat{p}(z' \mid xy) y_j z'_m - \frac{2C}{N} Y_{jm} \tag{29}$$

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

$$(\forall j, m = 1, 2, \ldots d) \;\; X_{jm} \leftarrow X_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial X_{jm}} \tag{30}$$

$$(\forall j, m = 1, 2, \ldots d) \;\; Y_{jm} \leftarrow Y_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial Y_{jm}} \tag{31}$$

I Practical hints for stochastic gradient ascent

I.1 Choose your hyperparameters carefully

In practice, the convergence rate of stochastic gradient ascent is sensitive to the initial guess $\vec{\theta}^{(0)}$ and the learning rate γ_0. It's common to take $\vec{\theta}^{(0)} = \vec{0}$ (e.g., initialize all entries of X and V to 0), and we recommend trying $\gamma_0 = 0.1$ for spam detection or $\gamma_0 = 0.01$ for language ID.

[11] If you prefer to think of computing the whole gradient vector at once, using vector computations, you can equivalently write this as

$$\nabla F_i(\vec{\theta}) = \vec{f}(xyz) - \sum_{z'} \hat{p}(z' \mid xy) \vec{f}(xyz') - \frac{2C}{N}\vec{\theta}$$

.

(Note that the assignment asks you to use the hyperparameters recommended above when `loglin` is selected (question 6b). This will let you and us check that your implementation is correct. However, you may want to experiment with different settings, and you are free to use those other settings when `improved` is selected, to get better performance.)

I.2 Don't modify the parameters as you compute the gradient

Make sure that at line 7 of Algorithm 1, you compute the entire gradient before modifying $\vec{\theta}$. If you were to update each parameter immediately after computing its partial derivative, then the subsequent partial derivatives would be incorrect.

I.3 Compute the gradient efficiently

Optimization should be reasonably fast (seconds per epoch).

Be careful that you compute the second term of (27) in time $O(V)$. If you're not careful, you might use $O(V^2)$ time, because each of the V summands has a factor $\hat{p}(z' \mid xy)$ whose denominator $Z(xy)$ itself takes $O(V)$ time to compute. The salvation is that xy remains the same throughout example i. So when you're computing a gradient vector $\nabla F_i(\vec{\theta})$, you only have to compute $Z(xy)$ *once* and reuse it for all z' and all k.[12] *Warning:* However, as $Z(xy)$ depends on $\vec{\theta}$, it will become out-of-date once you update $\vec{\theta}$. You can't save it to reuse for some future example j where $w_{j-2}w_{j-1}$ also equals xy (not even if $j = i$ so that you're processing the very same example again).

Constant factors can also play a big role in the speed. For the particular log-linear model used here, reading section F.5.1 gave a way to speed up the gradient computation. In general, replacing for-loops with library vector/matrix operations (via numpy or jblas) will probably speed you up a lot.

If you really want to get fancy, you could completely eliminate the need to sum over the whole vocabulary by predicting each word *one bit at a time*, as in the hierarchical log-bilinear language model of Mnih and Hinton (2009); then you are making $\log_2 V$ binary predictions, instead of a V-way prediction.

I.4 Compute the gradient correctly: The finite-difference check

It's easy to make a mistake when computing the gradient, which will mess everything up. One way to test your gradient computation code is to use the finite-difference approximation (see Bouttou (2012), section 4.2, or Vieira (2017)). We won't require this test, but it can help you debug your code; it's generally wise to test that what you're computing really is the gradient!

Suppose you've just computed $F_i(\vec{\theta})$ for your current i and θ. Remember that F_i is differentiable at $\vec{\theta}$ (and everywhere), which means it can be approximated locally by a linear function. So try adding a small number δ to one of the weights θ_k to obtain a new weight vector $\vec{\theta}'$. By the definition of partial derivatives, you'd predict that $F_i(\vec{\theta}') \approx F_i(\vec{\theta}) + \delta \cdot \frac{\partial F(\vec{\theta})}{\partial \theta_k}$.

You can check this prediction by actually evaluating $F_i(\vec{\theta}')$. By making δ small enough (say 1e-6), you should be able to make the predictions match the truth arbitrarily well (say to within

[12] Alternatively, note that you can rewrite the second term from the previous footnote as $\frac{\sum_{z'} u(xyz')\vec{f}(xyz')}{\sum_{z'} u(xyz')}$. Then a *single* loop over z' serves to compute the numerator (a vector) and the denominator (the scalar $Z(xy)$) in parallel. You can then increment $\vec{\theta}$ by $\gamma/Z(xy)$ times the numerator vector.

1e-10). If you can't, then either there's a bug in your gradient computation or (unlikely) you need to be using higher-precision floating-point arithmetic.

(Your program doesn't have to check every single partial derivative if that's too slow: each time you compute a partial derivative, you can flip a weighted coin to decide whether or not to check it. Once you're sure that the gradient computation is correct, you can turn the checks off.)

More generally, you can check that $F_i(\vec{\theta} + \vec{\delta}) \approx F_i(\vec{\theta}) + \vec{\delta} \cdot \nabla F_i(\vec{\theta})$ where $\vec{\delta}$ is any small vector, perhaps a vector of small random numbers, or a vector that is mostly 0's, or a small multiple of $\nabla F_i(\vec{\theta})$ itself. (Again, you don't have to do this check every time you compute a gradient.) A slightly better version is the symmetric finite-difference approximation, $\frac{1}{2}\left(F_i(\vec{\theta} + \vec{\delta}) - F_i(\vec{\theta} - \vec{\delta})\right) \approx \vec{\delta} \cdot \nabla F_i(\vec{\theta})$, which should be even more accurate, although it requires two extra computations of F_i instead of just one.

Thinking about why all these approximations are valid may help you remember how partial derivatives work.

I.5 How much training is enough?

In reading H.2, you may have wondered how to choose E, the number of epochs. The following improvements are not required for the assignment, but they might help. You should read this section in any case.

We are using a fixed number of epochs only to keep things simple. A better approach is to continue running until the function appears to have converged "well enough." For example, you could stop if the average gradient over the past epoch (or the past m examples) was very small. Or you could evaluate the accuracy or cross-entropy *on development data* at the end of each epoch (or after each group of m examples), and stop if it has failed to improve (say) 3 times in a row; then you can use the parameter vector $\vec{\theta}$ that performed best on development data.

In theory, stochastic gradient descent shouldn't even use epochs. There should only be one loop, not two nested loops. At each iteration, you pick a random example from the training corpus, and update $\vec{\theta}$ based on that example. That's why it is called "stochastic" (i.e., random). The insight here is that the regularized log-likelihood per token, namely $F(\vec{\theta})$, is actually just the average value of $F_i(\vec{\theta})$ over all of the examples (see (25)). So if you compute the gradient on one example, it is the correct gradient on average (since the gradient of an average is the average gradient). So line 7 is going in the correct direction on average if you choose a random example at each step.

In practice, a common approach to randomization is to still use epochs, so that each example is visited once per epoch, but to *shuffle the examples into a random order* at the start of training or at the start of each epoch. To see why shuffling can help, imagine that the first half of your corpus consists of Democratic talking points and the second half consists of Republican talking points. If you shuffle, your stochastic gradients will roughly alternate between the two, like alternating between left and right strokes when you paddle a canoe; thus, your average direction over any short time period will be roughly centrist. By contrast, since Algorithm 1 doesn't shuffle, it will paddle left for the half of each epoch and then right for the other half, which will make significantly slower progress in the desired centrist direction.

J Ideas for log-linear features

Here are some ideas for extending your log-linear model. Most of them are not very hard, although training may be slow. Or you could come up with your own!

Adding features means throwing some more parameters into the definition of the unnormalized probability. For example, extending the definition (16) with additional features (in the case $d = 2$) gives

$$u(xyz) \overset{\text{def}}{=} \exp\left(\vec{x}^\top X \vec{z} + \vec{y}^\top Y \vec{z} + \theta_9 f_9(xyz) + \theta_{10} f_{10}(xyz) + \dots\right) \tag{32}$$

$$= \exp\left(\underbrace{\theta_1 f_1(xyz) + \cdots + \theta_8 f_8(xyz)}_{\text{as defined in (19)–(20)}} + \theta_9 f_9(xyz) + \theta_{10} f_{10}(xyz) + \dots\right) \tag{33}$$

J.1 Unigram log-probability

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 section J.2 below) 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.

But first, here's a simpler method: just add a single *non-binary* feature defined by

$$f_{\text{unigram}}(xyz) = \log \hat{p}_{\text{unigram}}(z) \tag{34}$$

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 (27). 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 (14) to see that this is true). This means that in the special case where $\beta = 1$ and $X = Y = 0$, 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 ($X, Y \neq 0$) to judge which words z are likely in the context xy.

A simpler way to implement this scheme is to define

$$f_{\text{unigram}}(xyz) = \log(c(z) + 1) \quad \text{(where } c(z) \text{ is the count of } z \text{ in training data)} \tag{35}$$

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.

J.2 Unigram, bigram, and trigram indicator features

Try adding a unigram feature f_w for each word w in the vocabulary. That is, $f_w(xyz)$ is 1 if $z = w$ and 0 otherwise. Does this work better than the log-unigram feature from section J.1?

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 (27) with $C = 0$,

$$\frac{\partial F(\vec{\theta})}{\partial \theta_w} = \sum_{i=1}^{N} \frac{\partial F_i(\vec{\theta})}{\partial \theta_w} \tag{36}$$

$$= \underbrace{\sum_{i=1}^{N} f_w(xyz)}_{\text{observed count of } w \text{ in corpus}} - \underbrace{\sum_{i=1}^{N} \sum_{z'} \hat{p}(z' \mid xy) f_w(xyz')}_{\text{predicted count of } w \text{ in corpus}} \tag{37}$$

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.

J.3 Embedding-based features on unigrams and trigrams

Oddly, (16) only includes features that evaluate the *bigram* yz (via weights in the Y matrix) and the *skip-bigram* xz (via weights in the X matrix). After all, you can see in (18) 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 $x_h y_j z_m$.

J.4 Embedding-based features based on more distant skip-bigrams

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 X matrix at all positions $i-10, i-9, \ldots, i-2$ (while still using a separate Y 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 (18), you could scale down the feature value $(x_j z_m)$ by the number of word tokens that fall between x and z.[13]

Note: The provided code has separate methods for 3-grams, 2-grams, and 1-grams. To support general n-grams, you'll want to replace these with a single method that takes a list of n words. It's probably easiest to streamline the provided code so that it does this for all smoothing methods.

J.5 Spelling-based features

The word 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 $\sum_{j=1}^{d}\sum_{m=1}^{d} Y_{jm} \cdot (y_j z_m)$ in (18). 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 Y_{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, Y can be represented as a hash map whose keys are pairs of properties, such as pairs of suffixes.

J.6 Meaning-based features

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.

J.7 Repetition

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

[13] 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 $\lambda_6 X$. Now X is shared across all positions, but the various multipliers such as λ_6 are learned by SGD along with the entries of X and Y. 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 X and Y.)

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 $\{w_{i-20}, w_{i-19}, \ldots w_{i-1}\}$.

J.8 Ensemble modeling

Recall that (34) 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.