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

Prof. Jason Eisner — Fall 2024
Due date: Wednesday 2 October, 11 pm

Probabilistic models are an indispensable part of modern NLP. This homework will try to convince you that even simplistic and linguistically stupid models like n-gram models can be useful, provided their parameters are estimated carefully. See section A in the attached reading handout.

You now know enough about probability to build and use some trigram language models. You will experiment with different types of smoothing, including using PyTorch to train a log-linear model. You will also get some experience in running corpus experiments over training, development, and test sets. This is the only homework in the course to focus on that.

Homework goals: After completing this homework, you should be comfortable with

- estimating conditional probabilities from supervised data
  - direct estimation of probabilities (with simple or backoff smoothing)
  - conditional log-linear modeling (including feature engineering using external information such as lexicons)
  - subtleties of language modeling (tokenization, EOS, OOV, OOL)
  - subtleties of training (logarithms, autodiff, SGD, regularization)
- evaluating language models via sampling, perplexity, and multiple tasks, using a train/dev/test split
- tuning hyperparameters by hand to improve a formal evaluation metric
- implementing these methods cleanly in Python
  - partitioning the work sensibly into different classes, files, and scripts that play nicely together
  - using basic facilities of PyTorch
  - using remote GPUs to speed up your computation (if you choose)

Collaboration: *You may work in teams of up to 2 on this homework.* 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. **Your collaborator may not be** your discussion partner from HW2. Make new friends! :-)

Reading: Read the long handout attached to the end of this homework!

Materials: Python starter code and data are at `http://cs.jhu.edu/~jason/465/hw-lm/`. You've already used some of the data (the lexicons) in the previous homework.[1]

[1] If you lack the bandwidth to download all of the data files to your own machine, just download a few. Once your code is working, you can upload your code to the `ugrad` filesystem and run it on one of the `ugrad` machines, where the same files are available in the directory `/usr/local/data/cs465/hw-lm/`. Please don't make additional copies of the data on the `ugrad` filesystem, as this would waste space; you can use symbolic links instead.

On getting programming help: Use Python, building on the provided starter code. Since this is an upper-level NLP class, not a programming class, I don't want you wasting much time on low-level issues like syntax, type annotations, and I/O. Also, I don't want you to get stuck on understanding the starter code. 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 a CA) helps you with Python issues. Those aren't Interesting™.

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

The ☞ symbol in the left margin of this handout marks items that should be answered in your PDF. In your PDF, you should refer to question numbers like 3(a). (Don't refer to the blue ☞ numbers; they are just for your convenience and may change if this homework handout is updated.)

How to test and hand in your code and models:

✍1
- Your code and your trained models will need to be zipped and uploaded separately on Gradescope. We will post more detailed instructions on Piazza.
- 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 (question 7(d)), an autograder will run your system and score its accuracy on `test` and `grading-test` data (see reading section C.1).
  - 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.
  - The `test` results are intended to help you evaluate your own system. Grades will be based on separate `grading-test` data that you have never seen, simulating what happens when you deploy your tested system in the real world.

1 Perplexities and corpora

Your starting point is the sample programs `build_vocab.py` `build_lm.py` `fileprob.py`, in the `code` directory. The `INSTRUCTIONS` file in the same directory explains how to get the programs running (e.g., exactly what to type). Those instructions will let you automatically compute the $\log_2$-probability of three sample files (`data/speech/{sample1,sample2,sample3}`). Try it!

More precisely, for each file, `fileprob` will give you the total $\log_2$-probability of all token sequences in the file. Each line of the file is considered to be a separate token sequence (sentence or document) that is implicitly preceded by BOS and followed by EOS.

Next, you should spend a little while studying those sample programs yourself, and browsing around the `data/` directory to see what's there. See reading sections B and C for more information.

☞2 If a language model is built from the `data/speech/switchboard-small` corpus, using add-0.01 smoothing and a vocab threshold of 3, what is the model's *perplexity per word* on each of the three sample files?

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

2 Implementing a generic text classifier

Modify `fileprob` to obtain a new program `textcat` that does text categorization via Bayes' Theorem.

The two programs both use the same `probs` module to get the language model probabilities. So when you extend `probs.py` with new smoothing methods in question 5 below, they will immediately be available from both programs.

`textcat` should be run from the command line almost exactly like `fileprob`. However,

- it needs to take *two* language models, one for each category of text
- it needs to specify the prior probability of the first category

Of course, you can imagine extending this to do n-way classification, with n language models each trained on a different corpus, and prior probabilities for each of the n categories. But in this homework, we will stick with binary classification.

Again, consult `INSTRUCTIONS` for some tips on working with the starter code.

You'll train models of genuine emails (`gen`) and spam emails (`spam`). You (and the graders) should then be able to use your program for classification like this:

```
./textcat.py gen.model spam.model 0.7 foo.txt bar.txt baz.txt
```

which uses the two trained models to classify the listed files. Its printed output should label each file with a training corpus name (in this case `gen` or `spam`):

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

In other words, your program classifies each file by printing its maximum *a posteriori* class (the file name of the model that more probably generated it). Then it prints a summary of all the files.

The number 0.7 on the above command line specifies your prior probability that a test file will be `gen`. Thus, 0.3 is your prior probability that it will be `spam`. See reading section C.2.

Please use the exact output formats above. If you would like to print any additional output lines for your own use, please direct it to STDERR, using the logging facility as illustrated in the starter code.

As reading section D.3 explains, both language models that you provide to `textcat` should use the same finite vocabulary. Specifically, please construct this vocabulary to consist of words that appeared ≥ 3 times in the *union* of the `gen` and `spam` training corpora, plus OOV and EOS.[2]

To check your work: When we trained on the smallest training sets with $\lambda = 1$ and then classified all 270 dev files with a prior $p(\texttt{gen}) = 0.7$, just as in question 3(a) below, we classified 23 of them as `spam`.

[2]Of course, OOV and EOS may appear in the training corpora too—in fact, EOS must appear. But they might appear < 3 times. `build_vocab` is careful to include them in the vocabulary anyway. This is important. If OOV weren't in the vocabulary, we wouldn't be able to handle OOV words. And if EOS weren't in the vocabulary (and hence was treated as just part of the OOV category), then we couldn't sample from the language model—we wouldn't know when we had generated EOS and could end the sentence!

3 Evaluating a text classifier

In this question, you will evaluate your `textcat` program on the problem of spam detection. The datasets are under `data/gen_spam`. Look at the `README` file there, and then examine the training data to get a sense for how the genuine emails differ from the spam emails. **(Don't peek at the test data!)**

Using add-1 smoothing, run `textcat` on all the dev data for your chosen task. That is, train your language models on the `gen` and `spam` training sets, and then classify the files `gen_spam/dev/gen/*` and `gen_spam/dev/spam/*`. Use 0.7 as your prior probability of `gen`.

(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? (See reading section E.)

(b) *Extra credit:* We will focus on the spam detection problem in this assignment. But lectures in class focused on the language identification task, using a character-trigram model instead of a word-trigram model. Formally, these settings are very similar. If you're curious, you can try out the language ID setting as well, using the data in `data/english_spanish`. This should be quite fast since the corpora and vocabulary are small. Train your language models on the `en.1K` and `sp.1K` datasets, then classify the files `english_spanish/dev/english/*/*` and `english_spanish/dev/spanish/*/*`. Use 0.7 as your prior probability of English. All of these files have been tokenized into characters for you, so that you can use the same code as before.[3]

What percentage of the dev files were classified incorrectly?

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

(d) 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 4(b) why $\lambda = 0$ probably won't work well.)

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

In principle, you should try a lot of λ values to find the one that does best on dev data.[4] However, for purposes of this assignment, it's okay to try just $\lambda \in \{5, 0.5, 0.05, 0.005, 0.0005\}$.

(e) In principle, you should apply different amounts of smoothing to the `gen` and `spam` models. For example, if `gen`'s dev set has a higher rate of novel words than `spam`'s dev set, then you'd want to smooth `gen` more.

However, for simplicity in this homework, let's smooth both models 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* $\lambda \in \{5, 0.5, 0.05, 0.005, 0.0005\}$?

(As in the previous question, you should be evaluating the `gen` model on `gen` development files only, and the `spam` model on `spam` development files only, to make sure that they are good models of

[3] Properly speaking, these sequences are not complete sentences. They are substrings plucked from the middle of documents, so they don't really have BOS or EOS. Ideally, you would change the iterator over trigrams to reflect that: if you observe the sequence $w_1 w_2 w_3 w_4 w_5$, the only trigram probabilities that you can compute or train are $p(w_3 \mid w_1 w_2)$, $p(w_4 \mid w_2 w_3)$, and $p(w_5 \mid w_3 w_4)$, because you don't know what was before w_1 (not necessarily BOS) or after w_5 (not necessarily EOS).

[4] Perhaps using the charming little golden-section search algorithm, a lovely approach when you only have one hyperparameter (λ). When you have multiple hyperparameters, it is typical to use grid search, Nelder-Mead, or a randomized search algorithm.

their intended categories. To measure the overall cross-entropy per token for a given λ, 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. Add the two results, and then divide by the *total* number of tokens in all of the development files.)

9 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^*.)

(f) Each of the dev files has a length. The length in words is embedded in the filename (as the first number).

Come up with some way to quantify or graph how the error rate of add-λ^* (on dev data) depends on file length. Some tips about graphing are in `http://cs.jhu.edu/~jason/465/hw-lm/graphing.html`. You may also be interested in correlations.

10 Write up your results.

11 (g) *Extra credit:* If you are also experimenting with language ID, similarly report on how error rate (on dev data) depends on file length. The length in words is again embedded in the filename (as the first number), and also appears in the directory name.

(h) 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: `gen` vs. `spam`; `gen-times2` vs. `spam-times2` (twice as much training data); and similarly for `...-times4` and `...-times8`.

12 Graph the error rate (on the y axis) versus the training data size (on the x axis). This is sometimes called a "learning curve." Do you expect error rate to approach 0 as training size $\rightarrow \infty$?

13 (i) *Extra credit:* If you're also experimenting with language ID, you can do the same exercise there if you're still curious. We've provided training corpora of 6 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`.

4 Analysis

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

14 (a) At the end of question 2 and in reading section D.4, the vocabulary size 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 add-λ estimate?

15 (b) What would go wrong with the add-λ estimate if we set $\lambda = 0$? You can even try it! (Remark: This gives an unsmoothed "relative frequency estimate." It is commonly called the *maximum-likelihood estimate*, because it maximizes the probability of the *training* corpus.)

16 (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 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 $c(xyz) = c(xyz') = 1$?

17 (d) In add-λ smoothing with backoff, how does increasing λ affect the probability estimates? (Think about your answer to the previous question.)

5 Backoff smoothing

Implement add-λ smoothing with backoff, as described in reading section F.3. 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! Think back to the Tablish language from recitation.

You can test out the method as you see fit, including experimenting with different λ values.

Note: Feel free to describe your findings in your writeup, but there is nothing to hand in here other than your code. To check your code, the autograder will run it on small training and testing files with some λ.

6 Sampling from language models

So far, we have used our language models to compute the probability of given word sequences. Each language model represents a probability distribution over word sequences.

But because these are well-defined probabilistic models, we can also *sample* from the distributions they represent. As we saw in class, we sample random text by rolling a sequence of weighted dice whose sides are words. This is a good way to see what the language model does and doesn't know about English.

This is just like Homework 1, where your PCFG allowed you both to sample a new sentence (`randsent`) and to compute the probability of a given sentence (`parse`). You can also do both of these things with an n-gram model, which is a different generative model of text.[5]

Implement a generic sampling method that will work with any of our trained language models. When called, your method should condition on BOS and produce new tokens until it reaches EOS. These tokens are drawn from the smoothed model distribution according to their probabilities. This should remind you of `randsent` in Homework 1. Note that OOV will sometimes be drawn, since it will have positive probability.

Your method should not return or yield BOS and EOS, since they are not part of the sampled sentence. They are just part of an n-gram model's particular sampling process (similar to private variables in a class implementation). If instead you were generating sentences with a PCFG model, then you would start with ROOT at the top rather than BOS at the left, and you would stop at terminal symbols at the bottom rather than needing to generate EOS at the right.

Write a separate script based on `fileprob` that will sample k sentences from a given language model, using the sampling method you described above. (Especially because of UNIFORM, impose a maximum length limit M, as in the PCFG homework. Sequences longer than your configurable limit should be truncated with "`...`".)

Since you're using PyTorch, you should probably sample using `torch.multinomial` (rather than the Python `random` module that you probably used in HW1).

We should be able to call your script like this:

```
./trigram_randsent.py model_file 10 --max_length 20
```

18 Choose two trained models that seem to have noticeably different behavior. (They might use different smoothing methods, or different hyperparameters.) Give a sample of 10 sentences from each of the models. Discuss the differences you see and why they arise.

[5] Actually, not so different: an n-gram model turns out to be a special case of a PCFG. Can you show that this is true for $n = 2$?

7 Implementing a log-linear model and training it with backpropagation

(a) Add support for a log-linear trigram model. This is another smoothed trigram model, but the smoothing comes from several feature functions instead of modified or backed-off counts. As usual, see `INSTRUCTIONS` for details about working with the starter code.

Your code will need to compute $\hat{p}(z \mid xy)$ using the specific features in reading section F.4.1. The parameters $\vec{\theta}$ will be stored in X and Y matrices. You can use random or zero parameters at first, just to get the code working.

Remember that you need to look up an embedding for each word, falling back to the OOL embedding if that word is not in the lexicon. In particular, OOV will fall back to OOL.

You can use embeddings of your choice from the `lexicons` directory. (See the `README` file in that directory.) These word embeddings were derived from Wikipedia, a large diverse corpus with lots of useful evidence about the usage of many English words.

Make sure to use character embeddings if you try `english`/`spanish`. For `gen`/`spam`, you should use word embeddings; `words-gs-only-*` is recommended based on our experiments.

(b) Implement a function that uses stochastic gradient descent to find the X and Y matrices in equation (7) that minimize $-F(\vec{\theta})$, which is the L_2-regularized objective function described in reading section H.1. (This is equivalent to maximizing $F(\vec{\theta})$.)

If you initialize the matrices to 0 (see reading section I.5) (so $\vec{\theta} = \vec{0}$), then you are initializing to a uniform distribution. So it is a good sanity check that training for 0 epochs (`--epochs 0`) in fact gives the same results as just using a uniform distribution (`--smoother uniform`).

You may prefer to try log-linear modeling first on language ID (`data/english_spanish`), since training a log-linear model takes significantly more time than add-λ smoothing. For example, here's what you should get if you train a log-linear language model for 10 epochs on `en.1K`, with a vocabulary of size 30 derived from `en.1K` with threshold 3, the features described in reading section F.4, the character embeddings of dimension $d = 10$ (`chars-10.txt`), L_2 regularization with coefficient $C = 1$, and the `torch.optim.SGD` optimizer with its default arguments (see reading section I.7.2) and learning rate $\gamma = 0.01$:

```
Training from corpus en.1K
epoch 1: F = -3.2130978107452393
epoch 2: F = -3.0860249996185303
epoch 3: F = -3.037041425704956
  ...  [you should print these epochs too]
epoch 10: F = -2.946611166000366
Finished training on 1027 tokens
```

You may find it helpful to speed this up by using a GPU as explained in reading section K. Learning how to do that will come in handy in question 7(d) below and again in Homework 6.

(c) 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. Just construct two log-linear models over a shared vocabulary, and then compare the probabilities of a new document (dev or test) under these models.

For the autograder's sake, when `log_linear` is specified on the command line, please train for $E = 10$ epochs, use the exact hyperparameters suggested in reading section I.5, use the `torch.optim.SGD`

optimizer (reading section I.7.2), and print output in the format above (this is printing $F(\vec{\theta})$ rather than $F_i(\vec{\theta})$). Your training numbers should be close to what we got, although we won't expect them to match perfectly.

19 Report cross-entropy and text categorization accuracy on `gen_spam` 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?

20 (d) Now you get to have some fun! Add some new features to the log-linear model and report the effect on its performance. In fact, this may be necessary to beat the simpler add-λ methods. 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.5 and I.7).

A good way to devise features is to try sampling sentences from the basic log-linear model (using your `sample` method from question 6). What's wrong with these sentences? Specifically, what's wrong with the trigrams, since that's all that you can fix within the limits of a trigram model? Are there features that you think are too frequent, or not frequent enough? If so, try adding these features, and then the trained model should get them to occur at the right rate. (Remember from the log-linear visualization that the predictions of a log-linear model, if it was trained without regularization, will have the same features on average as actual words in the training corpus.)

For this question, you can additionally try changing the optimization method (see reading section I.7) and the hyperparameters in search of better results.

Your improved method should be selected with the command-line argument `log_linear_improved` (in place of `add_lambda`, `log_linear`, etc.). You will submit your code and your trained model to Gradescope for autograding.

You are free to submit many versions of your system—with different implementations of `log_linear_improved`. All will show up on the leaderboard, with comments, so that you and your classmates can see what works well. (You should submit *only* `log_linear_improved` models to the leaderboard, not the other kinds of model, even if those do better.) For final grading, the autograder will take the submitted version of your system that worked best on the released `test` data, and then evaluate its performance on `grading-test` data.

8 Speech recognition

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, drawn from the Switchboard corpus (see the `README` file there). As usual, it is divided into training, development, and test sets. Here is a sample file (`dev/easy/easy025`):

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

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. Let's reason about how to choose among the 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?[6] 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; this will be used in question 9 below.[7]

21 We won't actually make you write any code here. But 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.)

[6] Actually, the real estimate was 15 times as large. Noisy-channel 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.

[7] The word error rate of each transcription was computed for you by a scoring program, or "scorer." 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.

9 *Extra credit:* Language modeling for speech recognition

Actually implement the speech recognition selection method in question 8, using one of the language models you've already built. Use the `switchboard` corpus for training. You may experiment on the development set before getting your final results from the test set. When experimenting, you may want to start out with training on `switchboard-small`, just for speed.

(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.

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):

```
./speechrec switchboard_whatever.model 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`.)

(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`?

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?

10 *Extra credit:* Open-vocabulary modeling

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).

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.

$\mathfrak{P}_{24}$ Just giving the formulas for your estimator will get you some extra credit. Implementing and testing them would be even better!

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 $\sum_z p(z \mid xy) = 1$

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

Prof. Jason Eisner — Fall 2024

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

A Are n-gram 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 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 for a long time, you could only do a *little* better—dramatic improvements over trigram models were 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 good trigram model's perplexity per word by 10% (equivalent to reducing the cross-entropy by just 0.152 bits per word).

Machine translation. Statistical machine translation (MT) systems were originally developed in the late 1980's and made use of trigram language models. After a quiet period, this paradigm was resurrected at the end of the century and started getting good practical results. Statistical MT systems often included 5-gram models trained on massive amounts of data.

Why 5-grams? Because an MT system that translates into English 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.

An English speech recognition system can get away without 5-grams because it is not generating a new English sentence. It observes the spoken version of an existing English sentence, and only has to guess what words the speaker *actually* said. 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 ("flower"), baking ("flour"), or cleaning ("floor").

Smoothing. Fancy smoothing techniques developed in the 1990's, applied to trigram models, eventually managed to achieve up to a total 50% reduction in perplexity per English word (equivalent to a cross-entropy reduction of 1 bit per word). A thorough review supported by careful comparative experiments can be found in Goodman (2001).

As they noted, however, improving perplexity didn't reliably improve the error rate of the speech recognizer. In some sense, the speech recognizer only needs the language model to break ties among utterances that sound similar. Many improvements to perplexity didn't happen to help break these ties.

Neural language models. A line of work starting in 2000 used neural networks to produce smoothed probabilities for n-gram language models. These neural networks can be thought of as log-*nonlinear* models—a generalization of the log-linear models considered in reading section F.4 below. The starting point for both is the word embeddings that were introduced on the previous homework.

Next, *recurrent* neural networks (RNNs) became a popular way to get *beyond* n-gram models. We will touch on these methods in this course. They are not limited to a fixed-length history. They can learn to exploit complex patterns in which the choice of next word is affected by the syntax, semantics, topic, style, and format of the left context.

RNN-based language models were originally proposed in 1991 by the inventor of RNNs, but there seem to be no published results on real data until 2010. In general, neural networks played little role in practical NLP until about 2014. Around then, thanks to a series of small innovations over the preceding decade in neural network architectures and parameter optimization, together with larger datasets and faster hardware, neural methods in NLP started to show real gains over traditional non-neural probabilistic methods. In particular, neural language models started to show real gains over n-gram models. However, it was noted that cleverly smoothed 7-gram models could still do about as well as an RNN model by looking at lots of features of the previous 6-gram (Pelemans et al. (2016)).

A new neural architecture called the Transformer, introduced in 2017, showed further empirical gains in language modeling, halving perplexity (i.e., saving 1 bit of cross-entropy) over RNN models of comparable size. Transformers quickly took over language modeling. We'll look at them later.

More training data. Of course, training on larger corpora helps any method! Wikipedia maintains a list of large language models—mostly Transformer-based—including the sizes of their training corpora. A breakthrough model was GPT-3 (Brown et al., 2020), an enormous Transformer with 175 billion parameters, trained on 500 billion tokens[1] of (mostly) English text obtained by crawling the web. It achieved a perplexity of 20.5 on the Penn Treebank test set and is quite remarkably good at generating and extending text passages across a wide range of topics, styles, and formatting. Its creators showed that these abilities can be used to help solve many other NLP tasks, because the language model has to know a lot about language, meaning, and the real world in order to do such a good job of predicting what people are going to say next. GPT-3 formed the basis of ChatGPT.

[1] These tokens are actually word fragments, rather than words: see reading section D.6.

B Boundary symbols

Remember from the previous homework 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 homework.

In other words, $w_N = \text{EOS}$ ("end of sequence"), while for $i < 1$, $w_i = \text{BOS}$ ("beginning of sequence"). Thus, $\vec{w}$ consists of $N - 1$ words plus an EOS symbol. Notice that we do not generate BOS but we do condition on it (it was always there).[2] Conversely, we do generate EOS but never condition on it (nothing follows it). The boundary symbols BOS, EOS are special symbols that do not appear among $w_1 \ldots w_{N-1}$.

In the homework that accompanies this reading, we will consider every *line* in a file to implicitly be preceded by BOS and followed by EOS. A file might be a sentence, or an email message, or a text fragment.

C Datasets for Homework 3

Homework 3 will mention corpora for three tasks: spam detection, language identification, and speech recognition. They are all at `http://cs.jhu.edu/~jason/465/hw-lm/data/`. 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 use collect counts on `train`, tune the "hyperparameters" like λ to maximize performance on `dev`, and then evaluate your performance on `test`.

In principle, you shouldn't look at `test` until you're ready to get the final results for a system, and then you must commit to reporting those results to avoid selective reporting. The danger of experimenting on `test` to improve performance on `test` is that you might "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 test set, not in general.

To be on the safe side, we will actually evaluate your system in a blind test, when we run it on new data you've never seen. Your grade will therefore be determined based on a `grading-test` set that you don't have access to. So overfitting to `test` might give you good results on `test` in your writeup, but it will hurt you on `grading-test`. (Just as if you tell your boss that the system works great and is ready to ship, and then it doesn't work for real users.)

C.2 Class ratios

In the homework, 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.

[2] Just like the ROOT symbol in a PCFG.

We can try to estimate the fraction from training data, or perhaps more appropriately from dev data (which are supposed to be "like the test data"). It happens that in dev data, $\frac{2}{3}$ of the documents are genuine email, and $\frac{1}{2}$ are English. In this case, the prior probability is a parameter or hyperparameter of the model, to be estimated from training or dev data as usual.

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 homework, 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.[3] 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 $= 0$.

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`

[3] This symbol sometimes goes by the alternative name UNK, which stands for "unknown."

Admittedly, this total probability is higher than the probability of the *particular* sentence involving `snuffleupagus`. But in most of this homework, 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 the *comparison* is fair.[4]

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 $p_{\text{model1}}(\texttt{i saw OOV on the tv})$ with $p_{\text{model2}}(\texttt{i saw snuffleupagus on the tv})$, since the former is actually the total probability of many sentences, and so will tend to be larger.

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

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

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

D.4 How to choose the vocabulary

How should you choose the vocabulary? For this homework, 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 or an integerizer.

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 will convert to OOV (if your code is correct).

D.5 Open-vocabulary language modeling

The homework assumes a fixed finite vocabulary. However, an *open-vocabulary* language model does not limit in advance to a finite vocabulary. Homework question 10 (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 character 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 $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$.)

[4] Homework question 10 and reading section D.6 discuss alternative approaches, which may also work better for text categorization, since they look at the spellings of the unfamiliar words rather than treating them all as identical OOVs.

D.6 Alternative tokenizations

A language model is a probability distribution over sequences of tokens. But what are the tokens? We are working with text that has been tokenized into *words*. That's why our finite vocabulary is large and why OOV tokens are nonetheless rather common in test data.

An alternative would be to tokenize into characters, as we did in the English/Spanish data. V is now very small, but we still expect excellent coverage. For instance, with just English letters, digits, whitespace characters, and punctuation marks, we can keep $V < 100$ and still handle almost all English text. Any additional characters such as emojis, math symbols, and characters from non-English alphabets can be treated as OOV.

Of course, we are now in the business of predicting individual characters. And now a trigram language model only looks at the two previous characters, which is not very much context than the two previous words. So in this case, we would want n in our n-gram model to be much larger than 3. (Or better yet, we'd use a neural language model.)

An intermediate option is to tokenize into subwords. For example, we could use a morphological tokenizer that breaks the word `flopping` into two morphemes: the stem `flop` and the suffix `-ing`. The language model then predicts these tokens one at a time. It might assign positive probability to the sequence `flop -ed` in test data even if the word `flopped` had never occurred in training data.

But building a morphological tokenizer is a challenging language-specific problem—and not all unknown words can be decomposed into known morphemes (consider person names, drug names, or misspelled words).

Thus, language models in recent years have often *learned* a subword tokenizer. Uncommon words are automatically split up into common word fragments—i.e., substrings that are common in the training corpus. (See footnote 1.) Thus, the training and test data are preprocessed into sequences of "word pieces" rather than words. In this case, `flopping` might be tokenized into `flop -ping` (or perhaps `flopp -ing`), which approximates the morphological analysis above. If all single-character strings are in the subword vocabulary, then OOV becomes unnecessary: it is possible to represent any string as a sequence of subwords.

E Evaluation metrics (also called "evaluation loss functions")

In this homework, you will measure your performance in two ways—which we discussed early on in class. To measure the intrinsic predictive power of the model, you will use cross-entropy (per token). To measure how well the model does at the extrinsic task of text categorization, 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 hyperparameter 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 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 doesn't merely compute a single output class for each document.[5] 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 $p(\texttt{gen} \mid document_1) = 0.49$, 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 $p(\texttt{gen} \mid document_1)$ 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.

- Another continuous error metric is the *log-loss*, which is the system's expected surprisal about the correct answer on the extrinsic task. The system's surprisal on document 1 is $-\log_2 p(\texttt{gen} \mid document_1) = -\log_2 0.49 = 1.03$ 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 $p(\texttt{gen} \mid document_1) = 0$ is infinitely surprised by the correct answer (namely $-\log_2 0 = \infty$). 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.[6] However, they are very different:

metric	what it evaluates	probability used	units	long docs count more?
log-loss	the whole classification system	$p(\texttt{gen} \mid document_1)$	bits per document	no
cross-entropy	the `gen` model within the system	$p(document_1 \mid \texttt{gen})$	bits per `gen` token	yes

E.2 Generative vs. discriminative

There is an important difference in style between these metrics.

[5] Unlike a decision tree classifier, or a perceptron classifier that chooses the class with the highest score.

[6] 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 cross-entropy metric you'll use on this homework 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!

Our cross-entropy (or perplexity) is a *generative metric* because it measures how likely the system would to randomly *generate* the observed test data. In other words, it evaluates how well the system predicts the test data.[7]

The error rate, expected error rate, and log-loss are all said to be *discriminative metrics* because they only measure how well the system discriminates between correct and incorrect classes. This is more focused on the particular task, which is good; but it considers less information from the test data. In other words, the metric has less bias, in the sense that it is measuring what we actually care about, but it has higher variance from test set to test set, and thus is less reliable on a small test set.

In short, a discriminative setup focuses less on explaining the input data and more on solving a particular task—less science, more engineering. The generative vs. discriminative terminology is widely used across NLP and ML:

evaluation (test data) We compared generative vs. discriminative evaluation methods above.

tuning (dev data) Methods for setting hyperparameters may optimize either a generative or discriminative metric on the development data. (Normally they would use the evaluation metric, to match the actual evaluation condition.)

training (train data) Similarly, methods for setting parameters may optimize either a generative or discriminative metric on the development data. These are called generative or discriminative training methods, respectively.

It is possible to use generative training as we are doing on this assignment (so that training gets to consider more information from the training data) but still use discriminative methods for tuning and evaluation (because ultimately we care about the engineering task).

modeling A generative *model* includes a probability distribution $p(\text{input})$ that accounts for the input data. Thus, this homework uses generative models (namely language models).

A discriminative model only tries to predict output from input, possibly using $p(\text{output} \mid \text{input})$. For example, a conditional log-linear model for text classification would be discriminative. This kind of model does not even define $p(\text{input})$, so it can't be used for generative training or evaluation.

F Smoothing techniques

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

F.1 Uniform distribution

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

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

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

[7] In fact, a fully generative metric would require the system to *fully* predict the test data—not only the documents but also their classes. That metric would be the joint log-likelihood, namely, $\log_2 \prod_i p(document_i, class_i) = \sum_i \log_2 p(document_i \mid class_i) \cdot p(class_i)$. The second factor here is the prior probability of the class (e.g., `gen` or `spam`), which would also have to be specified as part of the model.

F.2 Add-λ

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

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

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

F.3 Add-λ with backoff

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

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

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

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 Conditional log-linear modeling

In the previous homework, you learned how to construct log-linear models. Here's that tutorial reading again, but let's restate the construction in our current notation.[8]

Given a trigram xyz, our model p is defined by

$$p(z \mid xy) \stackrel{\text{def}}{=} \frac{\tilde{p}(xyz)}{Z(xy)} \tag{4}$$

where

$$\tilde{p}(xyz) \stackrel{\text{def}}{=} \exp\left(\sum_k \theta_k \cdot f_k(xyz)\right) = \exp\left(\vec{\theta} \cdot \vec{f}(xyz)\right) \tag{5}$$

$$Z(xy) \stackrel{\text{def}}{=} \sum_z \tilde{p}(xyz) \tag{6}$$

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.

The resulting distribution p depends on the value of $\vec{\theta}$. Training on data (see reading section H below) finds a particular estimate of $\vec{\theta}$ (which we could call $\widehat{\vec{\theta}}$), which yields our smoothed distribution $\hat{p}$. Stronger regularization during training gives stronger smoothing.

[8] Unfortunately, the log-linear tutorial reading also used the variable names x and y, but to mean something different than they mean in this homework. Its notation is pretty standard in machine learning.

F.4.1 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.3 below).

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

$$\tilde{p}(xyz) \stackrel{\text{def}}{=} \exp\left(\vec{x}^\top X \vec{z} + \vec{y}^\top Y \vec{z}\right) \tag{7}$$

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 (6): 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.[9] 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 (7)? 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 (7) really a *log-linear* function? Yes it is! Let's write out those d-dimensional vector-matrix-vector multiplications more explicitly:

$$\tilde{p}(xyz) = \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} x_j X_{jm} z_m + \sum_{j=1}^{d}\sum_{m=1}^{d} y_j Y_{jm} z_m\right) \tag{8}$$

$$= \exp\left(\sum_{j=1}^{d}\sum_{m=1}^{d} X_{jm} \cdot (x_j z_m) + \sum_{j=1}^{d}\sum_{m=1}^{d} Y_{jm} \cdot (y_j z_m)\right) \tag{9}$$

[9] 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 (5). Suppose $d = 2$. Then implicitly, we are using a weight vector $\vec{\theta}$ of length $d^2 + d^2 = 8$, defined by

$$\begin{array}{cccccccccc} \langle & \theta_1, & \theta_2, & \theta_3, & \theta_4, & \theta_5, & \theta_6, & \theta_7, & \theta_8 & \rangle \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ \langle & X_{11} & , X_{12} & , X_{21} & , X_{22} & , Y_{11} & , Y_{12} & , Y_{21} & , Y_{22} & \rangle \end{array} \tag{10}$$

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

$$\begin{array}{cccccccccc} \langle & f_1(xyz), & f_2(xyz), & f_3(xyz), & f_4(xyz), & f_5(xyz), & f_6(xyz), & f_7(xyz), & f_8(xyz) & \rangle \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ \langle & x_1 z_1, & x_1 z_2, & x_2 z_1, & x_2 z_2, & y_1 z_1, & y_1 z_2, & y_2 z_1, & y_2 z_2 & \rangle \end{array} \tag{11}$$

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 (11): if we adopt those feature definitions, then our general log-linear formula (5) will yield up our specific model (9) (= (7)) as a special case.

What keeps (7) log-linear is that the feature functions are pre-specified functions of xyz as shown in equation (11). Specifically, they are determined by the word embeddings $\vec{x}, \vec{y}$, and $\vec{z}$. We are not going to learn the word embeddings or the feature functions f_k—we only have to learn the weights θ_k.

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

Is this a sensible model? The feature definitions in (11) 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.[10] Then the product $x_2 z_1$ will be larger for trigrams like `kiss the baby` and `marry the cop`. 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

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

[10] 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.

$$\tilde{p}(\texttt{ers}) = \exp\left([-.5\ 1]\begin{bmatrix}1 & 0\\ 0 & .5\end{bmatrix}\begin{bmatrix}.5\\ .5\end{bmatrix} + [0\ .5]\begin{bmatrix}2 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}.5\\ .5\end{bmatrix}\right)$$
$$= \exp(-.5 \times 1 \times .5\ +\ 1 \times .5 \times .5\ +\ 0 \times 2 \times .5\ +\ .5 \times 1 \times .5) = \exp 0.25 = 1.284$$

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

$$\hat{p}(\texttt{s} \mid \texttt{er}) \stackrel{\text{def}}{=} \frac{\tilde{p}(\texttt{ers})}{Z(\texttt{er})} = \frac{\tilde{p}(\texttt{ers})}{\tilde{p}(\texttt{era}) + \tilde{p}(\texttt{erb}) + \cdots + \tilde{p}(\texttt{erz})} = \frac{1.284}{1.284 + \cdots} \tag{12}$$

Speedup. The example illustrates that the denominator

$$Z(xy) = \sum_{z'} \tilde{p}(xyz') = \sum_{z'} \exp\left(\vec{x}^\top X \vec{z'} + \vec{y}^\top Y \vec{z'}\right) \tag{13}$$

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'.[11] The results can be found as the elements of the row vector $x^\top XE$, 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 library calls in PyTorch (similarly to another matrix library, numpy). That can be considerably faster than a Python loop over all z'. That is because the library call is highly optimized and exploits hardware support for matrix operations (e.g., parallelism).

F.5 Other smoothing schemes

Numerous other smoothing schemes exist. In past years, for example, our course homeworks have used Witten-Bell backoff smoothing, or Katz backoff with Good–Turing discounting.

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 resulting formulas are 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 throughout a sentence or document. It would also use common sense and factual knowledge about the world. Thus, language modeling remains an active area of research that uses grammars, recurrent neural networks, and other techniques.

Indeed, it is reasonable to say that language modeling is AI-complete. That is, we can't solve language modeling without solving pretty much all of AI. This was essentially the point of Alan Turing's 1950 article "Computing Machinery and Intelligence," in which he considered the possibility of a machine that could sustain a deep, wide-ranging conversation. His "Turing Test" involves making you guess whether you're

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

conversing with a computer or a human. If you can't tell, then maybe you should accept that the computer is intelligent in some sense.[12]

(It follows that progress in language modeling might inadvertently *create* progress in AI. The enormous GPT-3 model (reading section A) is already startlingly good at generating sentences that are appropriate in context. It has no explicit representation of grammar, knowledge, or reasoning—yet by learning how to model its training corpus, it has implicitly picked up a lot of whatever is needed to generate intelligent text. While it is still imperfect in many ways, its creators demonstrated that we can interrogate it to get answers to other AI problems that it was not specifically designed to solve.)

G Safe practices for working with log-probabilities

G.1 Use natural log for internal computations

In this homework, 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.

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 unit conversion *only* when you print: at that point convert your $\log_e$-probability from negative nats to positive bits by dividing by $-\log 2 \approx -0.693$.

G.2 Avoid exponentiating big numbers (crucial for gen/spam!)

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 homework 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, or because of the way you randomly initialized your model's parameters.

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

[12] Perhaps the computer still can't perceive and manipulate physical objects, so there are parts of AI (robotics and vision) that it isn't solving. But it still has to be able to *talk* about the physical world, and Turing's proposal was that that being able to talk sensibly about things should be enough to qualify as intelligent.

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. $\log(p+q) = \log(\exp(lp)+\exp(lq)) = \text{logaddexp}(lp, lq)$. See the discussion of `logaddexp` and `logsumexp` in the future Homework 6 handout. These are available in PyTorch.

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 (18) 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

$$F(\vec{\theta}) \stackrel{\text{def}}{=} \frac{1}{N}\left(\underbrace{\left(\sum_{i=1}^{N} \log \hat{p}(w_i \mid w_{i-2}\ w_{i-1})\right)}_{\text{log likelihood}} - \underbrace{\left(C \cdot \sum_k \theta_k^2\right)}_{\text{L}_2\text{ regularizer}}\right) \tag{14}$$

which is the L_2-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).[13]

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

[13] 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

$$\underbrace{p(\vec{w} \mid \vec{\theta})}_{\text{likelihood}} \cdot \underbrace{p(\vec{\theta})}_{\text{prior}} \tag{15}$$

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 (14) is just maximizing the log of (15). 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 $\tilde{p}(xyz)$.)

H.2 Stochastic gradient descent

Fortunately, concave functions like $F(\vec{\theta})$ in (14) 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 (14) as an average of local objectives $F_i(\vec{\theta})$ that each predict a single word, by moving the regularization term into the summation.

$$F(\vec{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \underbrace{\left(\log \hat{p}(w_i \mid w_{i-2}\ w_{i-1}) - \frac{C}{N} \cdot \sum_k \theta_k^2 \right)}_{\text{call this } F_i(\vec{\theta})} \tag{16}$$

$$= \frac{1}{N} \sum_{i=1}^{N} F_i(\vec{\theta}) \tag{17}$$

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

procedure TRAIN
  $\vec{\theta} \leftarrow \vec{\theta}^{(0)}$
  $t \leftarrow 0$ ▷ *number of updates so far*
  for $e : 1 \rightarrow E$: ▷ *do E passes over the training data, or "epochs"*
    for $i : 1 \rightarrow N$: ▷ *loop over summands of (16)*
      $\gamma \leftarrow \frac{\gamma_0}{1 + \gamma_0 \cdot \frac{2C}{N} \cdot t}$ ▷ *current stepsize—decreases gradually*
      $\vec{\theta} \leftarrow \vec{\theta} + \gamma \cdot \nabla F_i(\vec{\theta})$ ▷ *move $\vec{\theta}$ slightly in a direction that increases $F_i(\vec{\theta})$*
      $t \leftarrow t + 1$
  return $\vec{\theta}$

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,[14] 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

[14] 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.

line 6 for gradually decreasing γ works well with our specific L_2-regularized objective (14).[15]

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.7 for some more thoughts about epochs.

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 (16). As you'll recall from the previous homework, each partial derivative takes a simple and beautiful form[16]

$$\frac{\partial F_i(\vec{\theta})}{\partial \theta_k} = \underbrace{f_k(xyz)}_{\text{observed value of feature } f_k} - \underbrace{\sum_{z'} \hat{p}(z' \mid xy) f_k(xyz')}_{\text{expected value of feature } f_k\text{, according to current } \hat{p}} - \underbrace{\frac{2C}{N}\theta_k}_{\text{pulls } \theta_k \text{ towards 0}} \tag{18}$$

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 (7), the feature weights are the entries of the X and Y matrices, as shown in (9). The partial derivatives with respect to these weights are

$$\frac{\partial F_i(\vec{\theta})}{\partial X_{jm}} = x_j z_m - \sum_{z'} \hat{p}(z' \mid xy) x_j z'_m - \frac{2C}{N} X_{jm} \tag{19}$$

$$\frac{\partial F_i(\vec{\theta})}{\partial Y_{jm}} = y_j z_m - \sum_{z'} \hat{p}(z' \mid xy) y_j z'_m - \frac{2C}{N} Y_{jm} \tag{20}$$

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

$$(\forall j, m = 1, 2, \ldots d) \;\; X_{jm} \leftarrow X_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial X_{jm}} \tag{21}$$

$$(\forall j, m = 1, 2, \ldots d) \;\; Y_{jm} \leftarrow Y_{jm} + \gamma \cdot \frac{\partial F_i(\vec{\theta})}{\partial Y_{jm}} \tag{22}$$

[15] It is based on the discussion in section 5.2 of Bottou (2012), "Stochastic gradient descent tricks," who has this objective as equation (10). You should read that paper in full if you want to use SGD "for real" on your own problems.

[16] This formula shows the partial derivative with respect to θ_k only. If you prefer to think of computing the whole gradient vector, for all k at once, you can equivalently write this using vector computations as

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

.

I Practical hints for stochastic gradient ascent

The magenta hyperlinks in this section may be particularly useful. Most of them link to PyTorch documentation.

I.1 Use automatic differentiation

You don't actually have to implement the gradient computations in reading sections H.3 to H.4! You could, of course. But the `backward` method in PyTorch will automatically compute the vector of partial derivatives for you, using a technique known as automatic differentiation by back-propagation (or simply "backprop"). So, you only have to implement the "forward" computation $F_i(\vec{\theta})$ for a given example i, and PyTorch will be able to "work backward" and find $\nabla F_i(\vec{\theta})$. This requires it to determine how small changes to $\vec{\theta}$ would have affected $F_i(\vec{\theta})$. See reading section I.6 below for details.

If the forward computation is efficient and correct, then the backward computation as performed by `backward` will also be efficient and correct. If instead you wrote your own code to compute the gradient, it would be easy to mess up and miss an algebraic optimization—you could end up taking $O(V^2)$ time when $O(V)$ is possible. You'd also have to put in checks to make sure that you were actually computing the gradient correctly (e.g., the "finite-difference check"). This is not necessary nowadays.

I.2 Don't try to learn all of PyTorch

We won't be learning the whole PyTorch framework in this class, although it's very nicely designed. All we need is the basics:

- rapidly manipulate tensors (multidimensional numeric arrays), just as in NumPy, Matlab, and other scientific computing packages
- automatically compute gradients via `backward`

(If you've built neural nets in PyTorch before, for example in the Deep Learning class, then you may have written models by extending `torch.nn.Module` and instantiating the `forward` method. **However, we're going to avoid using `torch.nn`.** If you're used to relying on it, check out Neural net from scratch to see how to train a network with the PyTorch basics only. That link is the starting point for an explanation of how `torch.nn` can optionally be used to streamline the code, but you don't need that.)

I.3 Make the forward computation efficient

In general, use PyTorch's tensor operations wherever possible. One such operation can do a lot of computation, and is much faster than doing the same work with a Python loop. Avoid Python loops!

When the vocabulary is large, the slowest part of the forward computation $F_i(\vec{\theta})$ is computing the denominator $Z(xy)$ for each xy that you find, because it involves computing $\tilde{p}(xyz)$ for all z and then summing over all of them. You can use fast PyTorch operations for this.[17] With efficient coding, each epoch should take about 1 minute on a CPU.

(*Advanced remark:* There are language models that eliminate the need to sum over the whole vocabulary by predicting each word *one bit at a time* (Mnih and Hinton (2009)). Then you only have to predict a

[17] In principle, it could be helpful to "memoize" $Z(xy)$ in a dictionary, so that if you see the same xy many times, you don't have to recompute $Z(xy)$ again each time: you can just look it up. Unfortunately, you *do* still have to recompute $Z(xy)$ if $\vec{\theta}$ has changed since the last time you computed it. Since $\vec{\theta}$ changes often with SGD, this trick may not give a net win.

sequence of $\log_2 V$ bits. Since each prediction is only over two options, the denominator in the probability only has to sum over 2 choices instead of V choices. This is *computationally* faster, but predicting those bits is a somewhat artificial task that's hard to do *accurately*. Nowadays people seem to just do the V-way prediction, ideally using a GPU to accelerate the PyTorch operations.)

I.4 Make the forward computation correct

Our probability model $p(z \mid xy)$ predicts the next word z from the vocabulary. Thus, the denominator Z in equation (6) should sum over the vocabulary of possible next words (including EOS and OOV). It should not sum over the lexicon, which is quite different! There might be dozens of vocabulary words that are not in the lexicon: but we still have to add them into Z, using the OOL embedding for *each* of them. Conversely, there might be thousands of lexicon words that are not in the vocabulary: the model simply can't generate any of them, although it can generate OOV.

To enable fast computation of Z, you can precompute a matrix of embeddings of the vocabulary words using `torch.stack`:

```
torch.stack([embedding(word) for word in self.vocab])
```

where `embedding(word)` looks up the embedding of a word in the lexicon (returning the OOL embedding if necessary). For this to be correct, `self.vocab` includes EOS and OOV.

For reasons discussed in reading section G.2, you won't want to compute $p(z \mid xy)$ directly using equation (4). Instead, you'll want to compute $\log p(z \mid xy) = \log \tilde{p}(xyz) - \log Z(xy)$, where you'd better use `logsumexp` to help find $\log Z(xy)$. For example, `torch.logsumexp(torch.Tensor([1,2,3]),0)` or equivalently `torch.Tensor([1,2,3]).logsumexp(0)` is mathematically equivalent to `log(exp(1)+exp(2)+exp(3))`, but is faster and more numerically stable.[18]

What is the reason for the `0` argument in these expressions? It specifies which dimension of the tensor to operate on. In the example above, we are applying `logsumexp` to a 1D tensor (vector), so the only possible dimension is `0`. But if `A` is a 2D tensor (matrix), then `A.logsumexp(0)` sums up each *row* separately, whereas `A.logsumexp(1)` sums up each *column* separately. This may actually be useful later in the assignment: if you try the mini-batching technique in reading section I.7.6 below, you might be computing $\log Z(xy)$ for many different contexts xy at once.

I.5 Choose your hyperparameters carefully

The convergence speed of stochastic gradient ascent is sensitive to the initial learning rate γ_0. We recommend trying $\gamma_0 = 10^{-2}$ for language ID and $\gamma_0 = 10^{-5}$ for spam detection, but you can experiment.

The initial guess $\vec{\theta}^{(0)}$ also matters. The easiest choice is $\vec{0}$ (i.e., initialize all entries of X and Y to 0), and that will work fine in this case, although it is not recommended in general.[19]

[18] When you're adding up thousands of terms using `log(exp(1)+exp(2)+exp(3)+···)`, the resulting $\log Z$ may be numerically imprecise—and its *gradient* will be so imprecise that trying to follow it actually won't be able to find good parameters for the model!

[19] To see what could go wrong, imagine that we were not given the word embeddings in a file, but were learning them along with X and Y. Such a model would be symmetric—different dimensions of the embeddings would be governed by the same equations. As a result, if we initialized all parameters to 0, then all parameters of the same kind would have to have identical updates, and therefore, they would continue to have identical values throughout optimization!

The standard solution is to initialize the parameters not to 0, but to random values close to 0—for example, drawn from a normal distribution with mean 0 and small standard deviation. This "breaks the symmetry" and allows the different dimensions of the word embeddings to specialize and play different roles.

(Note that the homework asks you to use the hyperparameters recommended above when `log_linear` is selected (question 7(b)). 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 `log_linear_improved` is selected (see question 7(d)) to get better performance.)

I.6 Compute and apply the gradient properly

To implement line 7 of Algorithm 1, you should do something like this:

```
F_i = log_prob - regularizer  # as defined in (16); depends on θ⃗ and on example i
F_i.backward()                # increases θ⃗.grad by the vector (∂F_i/∂θ_1, ∂F_i/∂θ_2, ...)
```

The first line is the final step of the forward computation, and the second line is the backward computation.

The parameters don't actually have to be named $\vec{\theta}$. For example, the probability model given by equations (4) to (7) has parameter matrices X and Y. Suppose we represent those as 2D tensors `X` and `Y`, and then use them to help compute `F_i`. Then `X.grad` and `Y.grad` will be magically defined to be 2D tensors of the same shapes as `X` and `Y`. The call to `F_i.backward()` serves to modify both of them: for example, it will increase `(X.grad)[5,2]` by $\frac{\partial F_i}{\partial X[5,2]}$. To make this happen, however, you need to specify before doing the forward computation that `X` and `Y` should track their gradients: either create them with the `requires_grad=True` flag, or call their `requires_grad_(true)` method right after you create them.

After doing the backward computation, to complete line 7, you can update `X` and `Y` like this:

```
with torch.no_grad():
    F += F_i                  # keep a running total of F over the epoch, if you like
    X += gamma * X.grad       # update X in the direction that will increase F_i
    Y += gamma * Y.grad       # update Y in the direction that will increase F_i
```

The `no_grad` directive says not to track the computations in this block of code, because we don't need the gradient of *those* computations. It would be very expensive if F, X, and Y had to remember *all* the steps that computed them! After all, F depends on all of the trigrams in this epoch, and the current values of X and Y depend on all of the updates from all of the epochs so far. If you kept track of all these computations at once, you would probably run out of memory during training. Fortunately, we only need to keep track of the computations for the current `F_i`, not the total `F`. Once we call `F_i.backward()`, and take the stochastic gradient step, we are allowed to forget how `F_i` was computed. This happens automatically because on the next iteration of the loop, `F_i` is assigned to a new value. The old value and its attached computation are then no longer accessible (as long as we're not tracking the computation of `F`), so Python can garbage-collect them when it gets low on memory.

Finally, remember that the `backward` algorithm doesn't *set* the `grad` tensors; it *increases* them, adding to whatever is already there. This is because of how the backprop algorithm works internally: it accumulates an answer into `grad` by successive addition. Thus, you must explicitly clean up by resetting these accumulators to 0 before the next call to `backward`.

```
    X.grad.zero_()
    Y.grad.zero_()
```

The method name `zero_` ends in _ because it modifies the tensor in place; this is a PyTorch convention.

The performance of the model is then sensitive to the random initialization. A common technique is "random restarts": train several times, initializing randomly each time, and return the trained parameters that worked best on development data.

I.7 Improve the SGD training loop

The following improvements are not required for the homework, but they might help you run faster or get better results. You should read this section in any case. Many of these techniques will pay off even more strongly in HW6.

I.7.1 Monitor your progress

Question 7(b) says to print the objective function (14) at the end of each epoch of Algorithm 1. You are free to show additional information using `log.info()`. You might want to frequently show the objective function, or perhaps its breakdown into log-likelihood and regularizer terms. Better yet, use the `wandb` Python module to send these numbers to the Weights & Biases website where you can view them on a graph.

The objective should improve reasonably steadily; if not, your stepsize γ may be too large. You can also show the log-likelihood on dev data; if you are improving on training data yet getting worse on dev data, then you are overfitting to the training data, so your regularization coefficient C may be too small.

I.7.2 Manage parameter updates using a PyTorch optimizer

One annoying thing about the approach of reading section I.6 is that you have to explicitly update both `X` and `Y`, and you have to explictly reset both `X.grad` and `Y.grad`. The more complex your model, the more variables there are to worry about. The `torch.nn` module can help with this, if you like:

- keep track of all of the model parameters via `torch.nn.Parameter`
- update those parameters via SGD or another optimizer of your choice

This is illustrated in the starter code we provided you. `EmbeddingLogLinearLanguageModel` inherits from `torch.nn.Module` for this reason. When an instance of this model is created, it registers the tensors `X` and `Y` as parameters of the model. Now all of the code in reading section I.6 can be simplified to

```
F_i = log_prob - regularizer    # as before
(-F_i).backward()               # note change of sign
optimizer.step()                # adjusts the parameters (X and Y)
optimizer.zero_grad()           # resets the gradient tensors (X.grad and Y.grad)
```

The `optimizer` object manages the state of an optimization algorithm such as SGD. The starter code shows how to create a basic SGD optimizer via `torch.optim.SGD`. By convention, PyTorch optimizers *minimize* an objective function by taking a step in the direction that will *decrease* the function. That is, `optimizer` insists on doing stochastic gradient *descent* rather than *ascent*. So instead of maximizing $F = \sum_i F_i$, we will use it to minimize $-F = \sum_i (-F_i)$. That is why we find the gradient of `-F_i` above.

I.7.3 Convergent SGD

`torch.optim.SGD` won't decrease the learning rate on every step. But decreasing the stepsize in an appropriate way (as in Algorithm 1) is necessary to guarantee that the SGD optimizer eventually converges to a local minimum.

In the special case of log-linear models, our objective function (the negative of equation (14)) happens to be convex, so it has a unique local minimum—namely the global minimum. It's best to use Algorithm 1 to be sure that that you converge to that. Any initial stepsize $\gamma_0 > 0$ and decrease rate $\lambda > 0$ will suffice (the choice $\lambda = \frac{2C}{N}$ is the one we used in reading section H.2). We have provided an implementation

as `ConvergentSGD`, which you can try as an improvement on `torch.optim.SGD`. Using its default arguments in the setup of question 7(b), it should achieve $F = -2.9677$ at the end of epoch 10.

(However, in the more general case of deep learning (see Homework 6), the simpler `torch.optim.SGD` may actually be a better choice, and is widely used. The reason is that the deep learning objective function will *not* be convex and will have local minima. Some of these are "bad" local minima that only work well on the particular training dataset you used. There is some evidence that keeping a constant learning rate will actually do better at avoiding these bad local minima (i.e., avoiding overfitting), so that the learned parameters generalize better to test data. That is why `torch.optim.SGD` does *not* decrease the learning rate.)

I.7.4 Deciding when to stop

In reading section H.2, you may have wondered how to choose E, the number of epochs. The homework asks you to use a fixed number of epochs ($E = 10$) only to keep things simple. The more traditional SGD 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.

In machine learning, our ultimate goal is not actually to optimize the training objective, but rather to do well on test data. Thus, a more common approach in machine learning is to compute the evaluation metric (reading section E) *on development data* at the end of each epoch (or after each group of m examples). Stop if that "dev objective" 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. This is known as "early stopping" because SGD may not yet have converged to an optimum on the training objective. Early stopping can be an effective regularizer (especially when C is too small) since it prevents overfitting to the training data. In effect, early stopping treats the number of epochs as a hyperparameter that is tuned on dev data. It's efficient and effective.

I.7.5 Shuffling

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. Again, you would evaluate on dev data after every m examples to decide when to stop. 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 (16)). 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 each epoch (including the first). 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.

I.7.6 Mini-batching

Each step of Algorithm 1 tried to improve $F_i(\vec{\theta})$ for some training example i (in our case, a trigram), by moving the parameters in the direction $\nabla F_i(\vec{\theta})$. But instead, we could choose a "mini-batch" I of several examples (typically the next examples in the shuffled order), and try to improve $\sum_{i \in I} F_i(\vec{\theta})$ by moving the parameters in the direction $\nabla \sum_{i \in I} F_i(\vec{\theta})$.

The advantage of mini-batching is that it breaks the serial dependency of SGD, where each example changes $\vec{\theta}$ for the next example and therefore we can only compute one example at a time. Another way of thinking about it is that a single SGD example now consists of several unrelated trigrams instead of just one. Mini-batching means that the probabilities of all these trigrams are determined using the *same* current parameter vector $\vec{\theta}$. Their various updates to $\vec{\theta}$ are added together (since the update $\nabla \sum_{i \in I} F_i(\vec{\theta})$ can be regarded as $\sum_{i \in I} \nabla F_i(\vec{\theta})$).

Since all of the trigram probabilities are all computed using the same math operations (but on different data), you can compute them "all at once" using tensor operations. This is often much faster than doing them one at a time, because the tensor operations are implemented in fast low-level C++ and may even exploit hardware speedups—vectorization (on a CPU) or parallelism (on a GPU).

Just take the next several trigrams xyz,[20] and put all of the x embeddings into one tensor, all of the y embeddings into another, and all of the z embeddings into a third. Then use PyTorch's tensor operations to compute the features and log-probabilities $\log p(z \mid xy)$ for all of these trigrams at once.

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 (7) with additional features (in the case $d = 2$) gives

$$\tilde{p}(xyz) \stackrel{\text{def}}{=} \exp\left(\vec{x}^\top X \vec{z} + \vec{y}^\top Y \vec{z} + \theta_9 f_9(xyz) + \theta_{10} f_{10}(xyz) + \dots\right) \tag{23}$$

$$= \exp\left(\underbrace{\theta_1 f_1(xyz) + \dots + \theta_8 f_8(xyz)}_{\text{as defined in (10)–(11)}} + \theta_9 f_9(xyz) + \theta_{10} f_{10}(xyz) + \dots\right) \tag{24}$$

J.1 OOV features

Some contexts are reasonably likely to be followed by OOV: "I looked up the word _____." Others are not: "I flew to San _____." It would be useful to do a good job of predicting $p(\text{OOV} \mid xy)$.

Unfortunately, our basic model (7) does not have the freedom to learn specific parameters for OOV. It just uses its general parameters X, Y with the pre-specified embedding of OOV.

Worse yet, the embedding of OOV is inappropriate. Recall from reading section F.4.1 that when $z =$ OOV, equation (7) takes $\vec{z}$ to be the embedding of OOL, since OOV is not in the lexicon. But that means OOV is treated just like any other out-of-lexicon word. That's wrong—typically, OOV should have a much higher

[20] For example, if `iter` is the infinite iterator over shuffled examples returned by `draw_trigrams_forever()`, then `itertools.islice(iter, k)` is a finite iterator that yields the next k trigrams from `iter`.

probability than any specific out-of-lexicon word that *is* in the vocabulary, because it represents a whole class of words: $p(\text{OOV} \mid xy)$ stands for the *total* probability of *all* words that are out-of-vocabulary.

Thus, you might add a simple feature $f_{\text{oov}}(xyz)$ that returns 1 if $z = \text{OOV}$ and returns 0 otherwise. Given any xy, the expression $\vec{x}X\vec{z} + \vec{y}Y\vec{z}$ is still equal for all OOL elements of the vocabulary, since they all have the same $\vec{z}$. But when $z = \text{OOV}$, the extra feature f_{oov} now fires as well, which increases $\tilde{p}(xyz)$ by a factor of $\exp\theta_{\text{oov}}$. As a result, for any xy, $\tilde{p}(xyz)$ is exactly $\exp\theta_{\text{oov}}$ times larger when $z = \text{OOV}$ than when z is some specific OOL word. The trained value of θ_{oov} determines how big this factor is. Roughly speaking, this factor should be large if the original training corpus had a lot of different OOV word types.

You could be fancier and try to learn different OOV parameters for the different dimensions. For example, you could add this to the score of xyz before you exponentiate:

$$\begin{cases} \vec{x} \cdot \vec{x}_{\text{oov}} + \vec{y} \cdot \vec{y}_{\text{oov}} & \text{if } z = \text{OOV} \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

Here $\vec{x}_{\text{oov}}$ and $\vec{y}_{\text{oov}}$ are learned parameter vectors, so in effect we are learning $2d$ additional feature weights. This should do a better job of learning how much to raise (or lower) the probability of $z = \text{OOV}$ according to the various properties of the context words x and y. Convince yourself that this model is still log-linear.[21]

J.2 Unigram log-probability

More generally, a weakness of the model (7) 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, which were learned from some other (larger) corpus.

It's pretty common in NLP to use SGD to adjust the embeddings at the same time as the other parameters. But then we wouldn't have a log-linear model anymore, as discussed below equation (11).

One way to fix the weakness while staying within the log-linear framework 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. We'll do that in reading section J.3 below.

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

$$f_{\text{unigram}}(xyz) = \log \hat{p}_{\text{unigram}}(z) \tag{26}$$

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 (18). 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 $\tilde{p}(xyz)$ (check (5) to see that this is true). This means that in the special case where $\beta = 1$ and $X = Y = 0$, we simply have $\tilde{p}(xyz) = \hat{p}_{\text{unigram}}$, so that the log-linear model gives exactly the same probabilities as the add-1 smoothed unigram model $\hat{p}_{\text{unigram}}$. However, by training

[21] An alternative would be to replace OOV's embedding $\vec{z}$ with a new, learned embedding vector $\vec{z}_{\text{oov}}$. Is *that* still log-linear? Well, if we only use the new embedding in the z position, then we're changing the score of xyz from $\vec{x}^\top X\vec{z} + \vec{y}^\top Y\vec{z}$ to $\vec{x}^\top X\vec{z}_{\text{oov}} + \vec{y}^\top Y\vec{z}_{\text{oov}}$. But we can get exactly the same effect with the scheme above by learning $\vec{x}_{\text{oov}} = X(\vec{z}_{\text{oov}} - \vec{z})$ and $\vec{y}_{\text{oov}} = Y(\vec{z}_{\text{oov}} - \vec{z})$.

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 quick way to implement this scheme is to define

$$f_{\text{unigram}}(xyz) = \log(c(z) + 1) \quad \text{(where } c(z) \text{ is the count of } z \text{ in training data)} \tag{27}$$

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

J.3 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-probability feature from reading section J.2?

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 (18) with $C = 0$,

$$\frac{\partial F(\vec{\theta})}{\partial \theta_w} = \sum_{i=1}^{N} \frac{\partial F_i(\vec{\theta})}{\partial \theta_w} \tag{28}$$

$$= \underbrace{\sum_{i=1}^{N} f_w(xyz)}_{\text{observed count of } w \text{ in corpus}} - \underbrace{\sum_{i=1}^{N} \sum_{z'} \hat{p}(z' \mid xy) f_w(xyz')}_{\text{predicted count of } w \text{ in corpus}} \tag{29}$$

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.4 Embedding-based features on unigrams and trigrams

Oddly, (7) 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 (9) that the features have the form $y_j z_m$ and $x_j z_m$. This seems weaker than add-λ with backoff. Thus, add unigram features of the form z_m and trigram features of the form $x_h y_j z_m$.

J.5 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 (9), you could scale down the feature value $(x_j z_m)$ by the number of word tokens that fall between x and z.[22]

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.6 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 (9). 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.

[22] 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.)

J.7 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.8 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 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 $\{w_{i-20}, w_{i-19}, \ldots w_{i-1}\}$.

Of course, this is no longer a trigram model, but that's ok!

J.9 Ensemble modeling

Recall that equation (26) included the log-probability from another model as a feature within your log-linear model. You could include other log-probabilities in the same way, such as smoothed bigram and trigram probabilities from question 5. The log-linear 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.

K Using Kaggle for GPU Acceleration (optional)

Once your Python code is working, here's a way to make your experiments run faster.

GPUs are well-suited for the highly parallelizable computations required in machine learning, such as when training log-linear models or neural networks. PyTorch encourages programmers to write their computations as operations on tensors, which are generally parallelizable. PyTorch can then carry out these operations more efficiently (and backpropagate through them) if GPU hardware is available.[23]

GPUs adopt a Single-Instruction Multiple-Data (SIMD) architecture: thousands of processors execute the same instruction in parallel, but on different data. This makes it fast to carry out tensor operations: square all elements of a vector, sum all rows of a matrix, multiply two large matrices together, transform all pixels of an image, compute feature vectors and probabilities for all trigrams in a mini-batch, etc.

Warning: The GPU will *not* speed up your code until you have organized the work into a few big tensor operations, rather than many little ones.

K.1 Kaggle Notebooks

While there are compute clusters on campus with GPUs, it is possible to pay a cloud computing provider for the use of their GPUs. There are also some opportunities to use cloud GPUs for free. Kaggle is a popular website (now owned by Google) for data science competitions. It provides its users with some free GPU use, up to a weekly limit.

[23] The same is true for other frameworks such as TensorFlow and Jax.

Kaggle lets you interact with Notebooks, Datasets, and Competitions. A Notebook is actually a Jupyter notebook: it allows you to keep a record of a sequence of computations, their results, and your natural-language notes on them. The computations are executed using available hardware (including GPUs), similar to Google Colab.

A Notebook contains a sequence of gray cells, each of which is either Code (by default, Python) or Markdown (formatted text). You can add, edit, move, collapse (hide), and delete cells.

You can *run* a Code cell. This sends its code to an ongoing interactive Python session that is running invisibly in the background—known as the *kernel*—and shows the output immediately below the cell. As a special case, a line of code that starts with `!` is executed by a `bash` shell. For example, you could type `!ls -l` to see the contents of the current working directory of the Python session.[24]

To get started, see the Using Kaggle section of `INSTRUCTIONS`. This explains how to create an account, create a Notebook, and add this homework's public Dataset to your Notebook's kernel's filesystem.

K.2 Your Code on Kaggle

But where do your code files go? Unfortunately, a notebook is only a single file, and is not even a `.py` file. There are two workarounds.

git repo Create a private code repository, hosted on github or some other git server. Clone it from your notebook so that it appears as a subdirectory of `/kaggle/working`. Whenever you edit the code on your local computer, push the changes to the repo and pull them from the notebook.

Kaggle Dataset Upload the code directory from your local computer to Kaggle as an additional Dataset—a private one—and add that Dataset to your Notebook's filesystem. It's not really data, of course, but it will appear as a subdirectory of `/kaggle/input` (and will be read-only). Whenever you edit the code on your local computer, upload a new version of the Dataset and then tell the Notebook to check for updates.

Either way, your `.py` files will now be in the filesystem that is visible from your notebook, so you can import them into your notebook, or invoke them from your notebook via shell commands.

Details of both methods are given in the `INSTRUCTIONS` file. The first method may be easier if you have a github account and are used to the git workflow.

You should probably get everything running first on your local computer, since it's easier to develop and debug there. Wait to upload your `.py` files to Kaggle until everything seems to be working correctly—with highly vectorized computations—and you need that GPU speedup.

K.3 Hardware Acceleration

Once you have things running in the Notebook, the `INSTRUCTIONS` file explains how to turn on the GPU.

[24] This `!ls -l` is just shorthand for the Python call `os.system("ls -l")`, which starts up a new `bash` process, executes the shell command `ls -l` within it, and ends the process. A line that starts with `%` or `%%` is also treated specially, as a "line magic" or "cell magic" command respectively. You can read more about Jupyter magic commands online.