UMass CS690N: Advanced Natural Language Processing, Spring 2018

Assignment 1

Due: Feb 7

Introduction: This homework includes a few mathematical derivations and an implementation of one simple language model—Saul and Pereira's "aggregate" latent-class bigram model. Pereira (2000) uses the model to try to disprove one of the most famous foundational examples in 20th century linguistics; you will replicate this experiment and assess the evidence. (Pereira 2000, "Formal grammar and information theory: together again?," Phil. Trans. R. Soc. Lond. A, 358, 1239-1253.)

You can implement code with whatever language you like; we recommend Python.

- Write your final answers in a PDF document. All equations and text must be typed (for example, with LaTeX or MS Word). Submit this to Gradescope.
- Upload a zip file of your code to Moodle. Do NOT include data files.
- For the writeup:
  - Put each full question (Q1, Q2, etc.) on a new page, and tell Gradescope where they are. We may not be able to grade you correctly if you don't do this!
  - Clearly mark the numbers of each question you're answering.
  - Always use the natural logarithm, please.

The end of this document has a number of implementation tips.

Data: Please use the Brown corpus as disributed by NLTK at `http://www.nltk.org/nltk_data/`. It defines tokens and sentences, and includes POS tags. Remove POS tags and lowercase all words. When we do this, we get 57340 sentences, 1161169 tokens (not including START or END symbols) and vocabulary size 49740 (not including END, though it may be convenient to throw in START and END into the vocabulary). Note the END symbols are NOT the same thing as sentence-final punctuation; punctuation symbols are themselves tokens. (Not all sentences end with punctuation!) Note that you might get slightly different answers if your preprocessing is slightly different; that's fine, just note what it is.

Take a look at some of the files so you have a sense of what's in it. The Brown Corpus was a pioneering dataset when it was released in the 1960s. It's tiny by modern standards (1M tokens), but is large enough to do a little latent-variable language learning.

To help verify it's loaded correctly, try printing out for yourself the most and least common words; for example, with python's collections.Counter and its most_common() (or just use a dict then sorted(dict.items(), key=lambda (w,c): -c)[:10]). You will use these little counting/ranking tricks all the time in NLP.

Q1: Models

Q1.1 (Unigram model; 2 points): Calculate and report $\log p$("colorless green ideas sleep furiously END") under an MLE unigram model.

Technically speaking, when evaluating a sentence's probability, you should always include the probability of generating the END token; this yields a proper, sums-to-1 probability distribution over all possible sentences.

Q1.2 (Bigram model; 1 point): What is the probability of this sentence under an MLE bigram model (a.k.a. first order Markov model)? Why?

Q1.3 (2 points): Why is it necessary to model the END symbol?
Technically speaking, when we define the probability of a sentence $(w_1, \ldots, w_T)$ under a Markov LM, we use:

$$\log p(w_1, w_2, \ldots, w_{T+1} = \text{END} \mid w_0 = \text{START})$$

Here we're trying to be careful with START/END symbols. Tokens 1 through T are actual observed word tokens. The zeroth word START is a dummy symbol for the model to condition on for generating the first word. And the word at {T+1} isn't really a word, but a dummy symbol that indicates the sentence is ending. The generative story says: you keep drawing the next word and emitting it, unless you draw END; then you stop.

But you could imagine not using END. For example, if you were trying to use a language model for rescoring a proposed translation $(w_1, \ldots, w_T)$, you could instead calculate:

$$p(w_1, w_2, \ldots, w_T \mid w_0 = \text{START})$$

Please give a reason why modeling END is important for linguistic quality. Use an example to back up your argument.

Q1.4 (5 points): Consider a different reason to model END: it is necessary for a well-formed probability distribution over all strings. (Formally, a string is a sequence of any integer length of symbols from a fixed set of symbols (a.k.a. vocabulary) $\mathcal{V}$.) A well-formed probability distribution is one that obeys the usual axioms of probability. Show that if you use the bad equation in Q1.3 to calculate a string's probability, the probability distribution you define is not well-formed.
(Note "show" means "prove"—that is, argue mathematically, though we don't require a formal proof. A correct answer can be very short.)

Q2: Sentence lengths

Q2.1 (5 points): Assume sentences are generated with a unigram language model. Show that the length distribution is a geometric distribution. Write the $P(\text{length} = n)$ probability in terms of LM parameters. You don't need all the LM parameters to derive this—which ones do you need, and why?

Q2.2 (8 points): Graph a histogram of the actual empirical sentence lengths in the data, next to the theoretical sentence length distribution you would expect to see, given the parameters you learned in your MLE unigram LM. Make sure to label the axes appropriately and use titles on the graphs.
Discuss whether these distributions are similar or different, and possible reasons why.

Q3: S+P basics

Q3.1 (4 points): List the parameters for Saul and Pereira's "aggregate bigram" model. Give them variable names and say, in short English descriptions, what the variables mean. How many parameters total does the full model have (the total dimensionality)? Notes: 1) the z variables are not considered parameters for this purpose, and 2) You can have fun thinking about whether to count the last dimension in a multinomial parameter vector, because technically, it's extraneous. We don't care whether you include it or not for this problem.

Q3.2 (2 points): Assume the model is learned and you have access to the parameters. In terms of the parameters you have defined, write out the equation that can be calculated to evaluate the log-probability of a new sentence $(w_1, \ldots, w_T)$:

$$\log p(w_1, w_2, \ldots, w_{T+1} = \text{END} \mid w_0 = \text{START})$$

Q3.3 (2 points): Write the E-step update equation for Saul and Pereira's model, in pseudocode that is mathematically precise, uses the notation you've defined, but is easier to read than computer code. You will have to define new mathematical variables; explain in English what they mean. Explain in English what the E-step update equations are doing.

Q3.4 (2 points): Write the M-step update equations for the model in a similar manner.

Q4: Implementing EM

Implement Saul and Pereira's latent-class transition model with EM training on the Brown corpus. EM can be tricky to debug; a set of implementation tips are included at the end of this document. Read over the questions below before starting; they ask about pieces of your implementation which you may be able to implement incrementally, instead of all at once.

Algorithm: We suggest using the following order for the algorithm. You can combine the E-step and the M-step 1 if desired (because you don't need to store the token-level posteriors). Also note we're taking a slightly more token-level view of the problem than the original paper, which describes a type-level implementation.

- Randomly initialize the parameters.
- For each iteration:
  - (E step): Calculate posteriors at each token level $p(z_t \mid w_{t-1}, w_t)$.
  - (M-step phase 1): Calculate expected counts of transitions and emissions.

– (M-step phase 2): Normalize these counts so they correctly sum to 1 to be the new parameters (note normalization is different for each parameter matrix).

Verification of M-step counts: You have to calculate summary statistics at the start of the M-step: the expected counts of word-to-state transitions (a matrix $V \times K$), and the expected counts of the state-to-word emissions (a matrix $V \times K$). One way to debug the E-step is to confirm that, if you go by the matrix shapes above, the row sums equal the raw word counts in the data. (You of course don't need to implement them with matrices exactly like this; this is for illustration.)

Q4.1 (1 point): Explain why this should be the case.

Q4.2 (4 points): Check whether this is the case empirically in your code. If the numbers are way different, there may be a bug in your code. Once you're reasonably certain you're debugged, report a little bit whether the numbers are the same or some of the numbers are the same. Give a useful quantitative and/or qualitative summary to compare these vectors.

Q4.3 (Verification of M-step probabilities; 2 points): Verify that your parameters sum to 1 the way you think they should. Explain.

Q4.4 (Verification of EM as an optimizer; 8 points): It can be proved that, as EM is running, the marginal log-likelihood $\log p(w) = \log \sum_z p(w, z)$ increases at each EM iteration.

In this model, the marginal log-likelihood is actually simple to evaluate since you can integrate out each token-level z in turn as you did in the per-sentence log-likelihood question. Please use the marginal probability equation to calculate the corpus log-likelihood and report it at each iteration. This will help verify your algorithm is working correctly. (But MLL calculation may be a bit slow, so you may want to sometimes turn it off during development.)

When reporting marginal log-likelihood, please give the average log-likelihood per token:

$$TokLL = \frac{1}{N_{tok}} \sum_{sentence} \log p(\text{words in sentence})$$

where N_{tok} is the number of probabilistic word generation events (number of tokens in the corpus plus number of END symbols). Note: perplexity is $\exp(-TokLL) = \frac{1}{\exp(TokLL)}$.

Q4.5 (2 point): Our dataset has approximately $V = 50000$. Therefore we should expect per-token average log probability to be higher than -10.8. Why?

Q4.6 (20 points): Once you feel relatively confident that training is working correctly, run EM for 20 iterations with $K = 3$ latent classes. Run this for three different random initializations, and save the per-iteration log-likelihoods each time. Plot all three curves on the same graph. Discuss the convergence behavior of EM.

Tip: You could save log-likelihood trajectories in global variables or something, or, you could simply print out the likelihoods at each iteration, save the output of your script, then string-munge them back out for the plot.

Q4.7 *Optional* extra credit. Prove the statement in Q4.4 that marginal likelihood increases in every EM iteration. You will want to analyze how EM optimizes the variational objective $J(Q,\theta) = \mathbb{E}_Q[\log p_\theta(w,z) - \log Q(z)]$.

Q5: Analyzing the model.

20 EM iterations with $K = 3$ should be enough to get the "colorless" example to work. We've noticed that it doesn't always, which is fine—you're investigating Pereira's claim. Once you have a trained model you feel confident is working as it is supposed to, conduct more fine-grained analysis as follows. (Jupyter notebook may be a convenient approach for this.)

Q5.1 (4 points): Let's compare to Pereira 2000, page 1245. Calculate and show the following two quantities. (We'll use the American English spelling of "colorless" to match our corpus.)

$$\log p(\text{colorless green ideas sleep furiously})$$
$$\log p(\text{furiously sleep ideas green colorless})$$

What is the probability ratio inferred by your model? How does it compare to Pereira's?

Q5.2 (1 point): Why can this model infer meaningful probabilities while the classic bigram Markov model cannot?

Q5.3 (5 points): Try at least 3 other example pairs of your choice, where each of a well-formed sentence and a nonsensical reordering. Which ones does the model get right or wrong? (Use sentences where all words are in-vocabulary).

Q5.4 (5 points): Find an example of a well-formed sentence and a reordering of it where you think the model gets the directionality wrong (or use one of the above). Explain what's going on. Develop a falsifiable hypothesis about your model and its linguistic properties, then develop a few more examples to test it on. Criticize and suggest a reformulation or refinement of your hypothesis for future work. How could the model be changed to improve performance on these examples?

Q5.5 (5 points): Is Pereira's claim sensitive to randomness in EM? First, explain how randomness in EM could affect results. Then try multiple random restarts of the learning algorithm to investigate this and report your results.

Q6: Discussion.

Q6 (10 points): This model is capable of evaluating sentence probability. What is the relationship between this or similar models' sentence probability, versus sentence grammaticality? In what ways do they track each other, and do not, and why?

We expect at least one paragraph of thoughtful discussion as the answer to this question. Back up your arguments with examples. Beyond the Pereira (2000) reading, for some ideas see potentially:

- Joe Pater's blog post:
  `http://blogs.umass.edu/comphon/2015/06/02/wellformedness-probability/`
- Peter Norvig's comments on Noam Chomsky's 2011 remarks (way too long, but somewhat interesting): `http://norvig.com/chomsky.html`

Implementation tricks and tips:

EM:

- First focus on defining the data structures and randomly initializing the parameters.
- Then get EM to run on just 1 iteration on a subset of sentences.
- Then get it to run for 1 iteration on all the sentences.
- Then try more iterations.
- When debugging, use a low K, since it's faster.
- Try a number of random restarts: EM is initialization-dependent so they will give different results.
- To tell whether it's working: at each iteration try printing out
  - The probabilities for a few sample sentences, like the "colorless" examples.
  - The corpus per-token log likelihood, though this is slow.
  - Print out top-10 most probable output words per class, excluding globally common words. The problem with outputting the most probable words in this model is that it doesn't learn to put function words in their own class; instead, they sit in all the classes at the top of each distribution. But if you dig a little deeper you can see this better; only rank among words that are outside the set of 1000 most frequent words in the corpus. You should be able to see different syntactic categories emerge during learning.
    * Code tip: numpy.argsort(-probs)[:10] is an easy way to get the indexes of the 10 highest entries, in descending rank order, from a vector (1d array) of probabilities.
    * Another method is to rank words by the ratio $\frac{p(w|z=k)}{p(w)}$, instead of just $p(w \mid z = k)$, a.k.a. "exponentiated pointwise mutual information". This automatically focuses on words especially unique to the class. Unfortunately, PMI overly aggressively prefers rare words and the words at the top of this ratio-ranked list will be extremely rare ones. If you try this, you'll have to add a filter to only include words with a count of at least 50 or 100 or something.
- Other notes:
  - Speed: Our naive Python implementation, which iterates over each token position (this is slow since Python is slow!), takes less than a minute per iteration.

– Data structures: It's typically easiest to use matrices to store the parameters. You will need to track the vocabulary as a two-way map between wordstrings and a unique integer index for each word type. We usually do this with two objects: a Python list (for integer → wordstring) and a Python dict (for wordstring → integer). If you don't like this, you could use Python dicts for everything instead, but it will be slower.

Other tips:

- Python 2.7 is still often used in NLP/ML. If you're using it, don't forget "from __future__ import division".

- We recommend writing functions in a Python file, then running them from a Jupyter notebook located in the same directory. You can write test code to call specific functions. If the file is called "lm.py", then just add this little line in the top of a cell:
  import lm;reload(lm)
  then call the various functions from the module "lm" that you want. Every time you re-execute the cell, you'll get the current code.

- One issue with module reloading is that object methods don't work well. It's a good idea to make a model class with all the data structures for the model. And maybe even to put initialization code as methods on that class. But if you put training or inference code as methods on that class, when you call reload(lm), your already-existing model objects will be stuck with the old methods, not the new ones. This will be confusing. But you don't want to re-train your model every time you change your code; maybe you want to debug your log likelihood function, for example. A better solution is to put important code as standalone functions in the file which simply take a model object as one of their arguments. This is the "dumb objects that are just a pile of data structures" approach to programming, as opposed to "smart objects that have functionality inside of them".

- If you are bored waiting for one iteration to finish, try printing out a progress indicator with e.g.:

```
for si,sentence in enumerate(sentences):
    if si % 1000 == 0: sys.stderr.write(".")
    [...rest of code...]
```