Project 2: Language Modeling

Due: Tuesday Feb. 15th, 11:00am (Eastern Time)

Deliverables

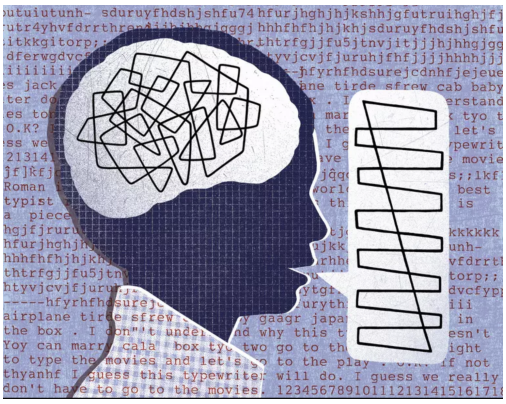
- Two python files: `unigram.py` and `bigram.py` (plus a third file, `crypto.py`, if you chose to complete the bonus section)
- `writeup.tex` and a corresponding `writeup.pdf` with answers to the Lab questions and which briefly lists the relevant output generated by the program.

A note on external libraries

In general you are free to use external python libraries. For instance, you will probably find **NumPy**[1] to be helpful for dealing with the **math associated with probabilities**. However you cannot use **NLTK** for this assignment, since the whole point is to implement these models yourself in order to understand how they work.

1 Unigram Models

In this assignment you'll write a program that can, at least to some degree, distinguish between real English and the output of a (very bad) French-to-English machine translation system.



The basic idea is that you create a language model (LM) for English, and then run it on both the good and bad versions of sentences. The one that is assigned the higher probability is declared to be the real sentence.

[1] After 10 years I still think it should rhyme with "lumpy" rather than "plum pie."

All the data you'll need for this is in `/data/cs65-S22/langmod/hansard/`. This data comes from the Canadian **Hansard's**, which are parliamentary proceedings and are **legally mandated** to appear in both English and French. Here we just use the English.

These files have one sentence per line and have already been tokenized for you. As you saw during the first Lab, there are always design decisions and tradeoffs to be made when doing text normalization (including tokenization), but here you can treat the provided data as **Gold standard** and not worry about it. If you do find any potential errors or interesting tid-bits about the tokenization, feel free to post about it under the "Social/Miscellaneous" tab on — there might be an interesting reason why it's been done that way, or there might not!

The main training data is `english-senate-0.txt`. You'll also need held-out development data to set the smoothing parameters, that's in `english-senate-1.txt`. You will need to implement a search algorithm to optimize α and β. I recommend **Golden-section search** as it's quite elegant, although you are not required to use that particular algorithm, e.g. you may find **Backtracking line search** to be simpler. Any implementation that properly finds the optimum smoothing parameters within a reasonable time is fair-game. A note about online resources here: since the Wikipedia page I link to above includes code, and this is a standard algorithm not specific to NLP, it is considered acceptable here to adapt the code for that function for your needs on the Lab. As always though, be sure to attribute any sources you find in a comment in the submitted code.

Your language model model will assign a probability to each sentence which you will evaluate over the test data in `english-senate-2.txt`.

Lastly, you will use your trained language model to differentiate good from bad English. In `good-bad-split.txt` we have pairs of sentences, first the good one, then the bad. Each sentence is one its own line, and each pair is separated from the next with a blank line.

1.1 Deriving the unigram model for document probabilities

You may have noticed that in class I defined a unigram language model as a distribution over documents as follows in Eq.1:

$$P(W) = P(N) \prod_{i=1}^{N} P(W_i) \tag{1}$$

where $P(N)$ is a distribution over document lengths and $P(W_i)$ is the probability of word W_i. You can read this formula as an instruction for generating a document W: first pick its length N from the distribution $P(N)$ and then independently generate each of its words W_i from the distribution $P(W_i)$. Two important factors simplify this equation when we instead want to model the probability of an existing document with a unigram model rather than generate it.

1. A unigram model assumes that the probability of a word $P(W_i)$ does not depend on its position i in the document, i.e., $P(W_i = w) = P(W_j = w)$ for all i, j in 1,...,N. This lets us model only a single distribution over word generation. This is: $P(W_i = w) = \theta_w$
2. For our purposes we can assume that $P(N)$ is the same for all languages, so we can safely ignore it. (A note for the reader: Why can we ignore it if it's the same for all languages?)

This leaves us with the likelihood function for our unigram model in Eq. 2, where $n_w(d)$ is the number of times word w appears in d.

$$L_d(\theta) = \prod_{w \in W} \theta_w^{n_w(d)} \tag{2}$$

We are using add-α smoothing for our unigram model. This means that we're adding a 'pseudo-count' of α to each word's empirical frequency from our training corpus. We then just need to readjust the denominator so that our estimates of θ still sum to 1. Thus our smoothed maximum likelihood estimator for unigram probabilities is given in Eq. 3.

$$\tilde{\theta}_w = \frac{n_w(d) + \alpha}{n_\circ(d) + \alpha|W|} \tag{3}$$

The $\circ$ notation means *'any word'*, so thus $n_\circ(d)$ is just the total token count of d. And $|W|$ is the cardinality of our vocabulary, which is just the number of types appears in d.

The estimate of α that maximizes the likelihood of the held-out development data is defined by the Argmax in Eq. 4. Where the likelihood of the some corpus 'x' given a particular α is defined in Eq. 5. This simply says that the likelihood of the data is the product of the probability of each word token in that data. For the development set we take 'x' to be the held-out corpus 'h' (`english-senate-1.txt`) whereas for the test data we take 'x' to be the test corpus 't' (`english-senate-2.txt`)

$$\hat{\alpha} = argmax_\alpha L_h(\alpha) \tag{4}$$

$$L_x(\alpha) = \prod_{w \in W} \theta_w^{n_w(x)} \tag{5}$$

1.2 Separating the good from the bad: Unigrams

Questions

1. Create a smoothed unigram language model with the smoothing parameter $\alpha = 1$. Compute the log probability of your language-model test data `english-senate-2.txt`.

2. Now set the unigram smoothing parameter α to optimize the likelihood of the held-out data. What value of α do you find? Repeat the evaluation described in the previous step using your new unigram model. (The log probability of the language-specific test data should increase.)

3. Try accurately differentiating the good and bad sentences in `good-bad-split.txt` using your language model from Q.2 and report your accuracy. This should work only moderately well (ballpark of about 70%); pick out a couple pairs of sentences that get categorized incorrectly by the model and provide your intuition as to why this is happening.

1.3 Implementation notes

You do not need particularly high precision when optimizing for α, a single decimal place will suffice.

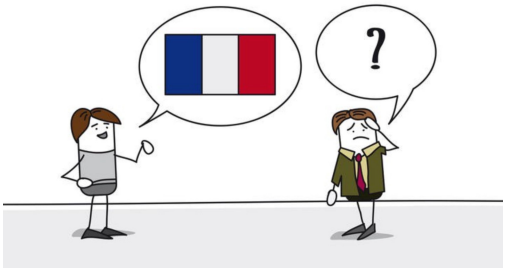
Several of your CS65 predecessors have asked me what to set as your upper and lower bounds when optimizing α. The lower bound for any smoothing parameter is of course that it needs to be positive. As for what a good upper-bound might be, I will leave to you to figure out yourself: in real-world NLP work parameter selection is often a big aspect of performance! A tell-tale sign that your bounds are too small would be if the 'best' value found is exactly your upper or lower bound.

Lastly, a note about run-time. You are not being graded on quickly your program runs[2]. However, a reasonable implementation of the unigram model here should complete this whole section on a modern machine in less than 10 seconds; if your program is taking substantially longer then that, you likely have a bug somewhere or poor choice of data structures.

[2]within reason that is. If your program runs hundreds of times slower than it should then I will be forced to deduct points

2 Bigrams

Now you'll solve exactly the same problem as from Section 1 but with a much more effective *bigram* language model.



2.1 Deriving the bigram model for document probabilities

When working with any N-gram model larger than unigrams we need to apply *padding tokens* to each sentence to indicate where it starts and stops. We can write this out as `*START*` and `*STOP*` but any unique symbols you like will do fine. Treat the padding tokens just like any other normal tokens even though they never appear in the sequences we can generate (or you can think that they *always* appear).

A bigram language model is defined in Eq. 6

$$P(W) = \prod_{i=1}^{n+1} P(W_i|W_{i-1}) \tag{6}$$

where $P(W_i = w'|W_{i-1} = w) = \Theta_{w,w'}$ for all indices i in 1,2,...,$n+1$ (and n is the number of words in the sentence).

Θ is a matrix of values specifying the conditional probability of w' given that it is immediately preceded by w. Just as in the unigram case, we assume that these conditional probabilities are time-invariant, i.e., they do not depend on i directly. Because probabilities must sum to one, it is necessary that $\sum_{w' \in W} \Theta_{w,w'} = 1$ for all $w \in W$.

At its core we want our estimates of Θ to reflect the relative frequency of a word conditioned on the preceding context as in Eq. 7.

$$\hat{\Theta}_{w,w'} = \frac{n_{w,w'}(d)}{n_{w,\circ}(d)} \tag{7}$$

where $n_{w,w'}(d)$ is the number of times that the bigram w, w' appears anywhere in our training corpus d (including the padding tokens), and $n_{w,\circ}(d)$ is the number of bigram tokens that begin with w. But note that this is equivalent to simply the number of times w appears in d.

Like in the unigram case though, we need to apply some smoothing to the Θ's in order to handle bigrams that we never saw in our training corpus. We again accomplish this by adding a 'pseudo-count' to each bigram's empirical frequency, but here indicated with the bigram-smoothing

parameter β. We keep this simple by setting the β's to be the same regardless of the bigram under consideration (i.e. we set $\beta_{w,w'} = \beta\tilde{\theta}_{w'}$). This gives us a single adjustable constant β with can be applied to Eq. 7 to produce our smoothed maximum likelihood estimator for bigram probabilities in Eq. 8.

$$\tilde{\Theta}_{w,w'} = \frac{n_{w,w'}(d) + \beta\tilde{\theta}_{w'}}{n_{w,\circ}(d) + \beta} \tag{8}$$

where $\theta_{w'}$ is computed exactly the same as it was for our unigram language model.

The likelihood of our held-out data under the bigram model is given in Eq. 9

$$L_h(\beta) = \prod_{w,w' \in W} \tilde{\Theta}_{w,w'}^{n_{w,w'}(h)} \tag{9}$$

where $\tilde{\Theta}_{w,w'}$ is defined in Eq. 8 and the product we multiply together need only range over the bigrams (w, w') that actually occur in h. (Note: make sure you understand why this is!) Just like in the unigram case, a simple line search can be used to find the value $\hat{\beta}$ of β that optimizes the likelihood of our held-out data.

2.2 Separating the good from the bad: Bigrams

Questions

4. Write a smoothed bigram model, setting $\beta = 1$. Keep the same α value that you found for your Unigram LM. Compute the log probability of your test data `english-senate-2.txt` according to the bigram LM.

5. Now optimize the bigram smoothing parameter β over the development data. What value of β maximizes the likelihood of the held-out data?

6. Lastly, use the smoothed bigram model to determine good/bad sentences and report your accuracy. You'll find this works far better than the unigram model (ballpark of about 90%). Nonetheless pick out a couple pairs of sentences that the model still gets wrong and provide your intuition as to why.

A few implementation notes. Remember: you need to pad every sentence with START and STOP tokens, whether reading in training data or estimating the likelihood of our held-out and test data.

The precision when optimizing β does not need to be any greater than when optimizing α in your unigram model; a single decimal place is sufficient.

To keep things simple for optimization you can just plug in the best value of α that you computed in the first part, and just optimize for the best β assuming that (In real-world applications you would need to optimize both simultaneously since they are not actually independent. But you don't need to worry about that here.)

Lastly the same note about run-time applies here as before. The bigram model necessarily runs more slowly than the unigram model, but the program should still complete within a couple minutes. If your implementation is taking substantially longer then that, you likely have a bug or a poor choice of implementation.

3 Feedback

Please provide answers to these in `writeup.tex` just like the rest of the questions. There are no right or wrong answers here of course :)

Questions

7. Approximately how long did you spend on this assignment?

8. Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

9. Which aspects of this assignment did you like? Is there anything you would have changed?

4 Extra Credit: Language modeling for cryptography

We can combine our language model from Parts 1 and 2 with a new LM over letter sequences to automatically break a simple encryption scheme, the **substitution cipher**. This is optional (but quite fun I think!) and you'll receive *up to* 15% extra credit on the assignment if you complete it.

Up to now we built n-gram language models over sequences of *words*. But the approach to learning probabilistic transitions between linguistic units is very general, and could be applied to morphemes, sentences or individual letters.

The trick for breaking a substitution cipher is that, since the replacement of original letters by replacement codes is a *fixed* system, the probability distribution of their occurrence remains the same — just the labels are different. This means that since `'t'` is the most frequent letter in English, then in our encrypted data whatever the most frequent letter is must correspond to `'t'`. Similarly, since only some letters are likely to occur in repeated sequence (`'ll'` or `'ss'` we can use that fact to map encrypted double-letters back to their original counter-parts. The same holds for both unigram and bigram frequencies at the word level as well.

If you learned the key that mapped from encrypted letter to original English letters then you can use that to translate the text back to the original.

We have also learned a method for evaluating our potential translations automatically: A language model that's trained on actual English will assign extremely low probability to encrypted text, but high probability to properly decoded text.

Your task in this bonus section (if you chose to do it) is to break a substitution cipher using language models. This involves figuring out the cipher-key, and outputting the unencrypted text. The data for this section is in `/data/cs65-S22/langmod/crypto/`. The file `train.txt` is text from the Gutenberg corpus that I have lower-cased for you (this way there are only 26 letter pairs to learn rather than 52). You can use this file to train LMs over both word and letter occurrence. The test files `test1shuffled.txt` and `test2shuffled.txt` are also files from the Gutenberg corpus which have been encrypted with different substitution ciphers. The cipher applied is straightforward: it only translates the 26 English letters `[a-z]`, leaving all other characters (*space*, numbers, punctuation, etc.) unchanged.

You are free to implement this however you want, but I'd suggest something like the following design with three sets of functions:

- proposing a decryption key (and storing partial answers)
- applying a decryption key to translate from input to output text
- evaluating how good the output text looks

You can of course look at the output by hand while you're debugging, and to test individual functions one-by-one, but the final program needs to run automatically without hard-coding in the cipher key.

This is open-ended and there are *many* possible strategies that will work, so have fun!