Homework 3: N-gram Language Models

Due Sept. 25th

1 Exploring text classification with n-grams

In class, we used an n-gram language model to generate text. In this assignment, you will explore the use of n-grams to analyze textual similarity.

You should put your code for this part in **ngram_classify.py**.

1.1 Reading Data with SpaCy

This week, we'll use the spaCy Python library. SpaCy is designed to make it easy to use pre-trained models to analyze very large sets of data.

Install SpaCy and import it to verify that it is working.

We will use SpaCy to tokenize our text. Pause now and read about space language processing pipelines.

For this part, we only want SpaCy to tokenize our text, so we will set the **pipeline** to **[]**. Since some of our documents will be long, but since we're not doing any memory-intensive processing, we will tell SpaCy that it's okay to load large documents all at once instead of a little bit at a time. To specify these instructions, add these lines to the top of your file:

```
from spacy.lang.en import English

nlp = English(pipeline=[], max_length=5000000)
```

1.2 Extracting data from XML files (15 points)

We will look at a set of XML files containing news articles. Each article has a **hyperpartisan** attribute that indicates whether is it an example of **hyperpartisan news**: "true" or "false". The full data files are in **semeval_full**, but some are quite large. We will mostly use a subsample of this data that is stored in **semeval-sample.xml**. However, it is good practice to write code that could work on the larger dataset.

Loading the entire dataset at once is a recipe for trouble. Fortunately, the lxml library gives us a way to iteratively parse through an xml file, dealing with one node at a time. Here's sample code that opens a file called **myfile.xml** and call a function called **my_func** on every **article** node:

```
from lxml import etree

fp = open("myfile.xml", "br")
```

```
for event, element in etree.iterparse(fp, tag=("article",)):
    my_func(element)
        element.clear()
```

The starter code has a generator function called **do_xml_parse()** that uses **lxml.etree** to yield one node at a time. Look at that code and make sure you can explain how each line of it works before you move on.

For all of the analysis in this lab, you **should** lowercase the tokens unless told otherwise.

Next, write a function called **get_articles(args, attribute, value)** that returns a Counter. **get_articles** will use **do_xml_parse** to iterate through all of the articles in **args.articles**. Here is an example of how to use **do_xml_parse**:

```
for article in do_xml_parse(args.articles, "article"):
        this_value = article.get(attribute)
```

get_articles should then gather unigram counts for each article whose attribute **attribute** has the value **value**. You can use the **get_unigrams** function we wrote in class to do this.

For example, **get_articles(args, 'hyperpartisan', 'true')** will call **get_unigrams** once for every article whose **hyperpartisan** attribute is 'true'.

Hint: If you have an article element called **article**, you can access all of the text children of that element with **article.itertext()**.

Hint: Some of the articles contain HTML entities that are "escaped", i.e., marked with special characters to avoid interfering with the XML parsing. You can unescape those by importing the **html** module and calling **html.unescape** on the articles' text before creating your SpaCy docs.

1.3 Check in

Stop now and confirm that if you call **get_articles** on the **semeval-sample.xml** file for the case where **hyperpartisan=true**, you get the following counts:

```
the: 10631
california: 34
zero: 11
```

1.4 Comparing Texts (15 points)

We are interested in knowing how many of the unigrams in one category of text (e.g., **hyperpartisan='true'**) are not in another category of text (e.g., **hyperpartisan='false'**). Over the course of this assignment, you will explore several ways of grouping the text, so we'll want to carefully organize our code for reusability. In the rest of this writeup, we'll refer to the set of data we generate counts from as the **training set**, and the set of data that we check for zeros using those counts as the **test set**.

Write a function called **compare(train_counter, test_counter, unique=False)**. The three arguments to **compare** should be:

- **train_counter**: A Counter object representing counts from the training set
- **test_counter**: A Counter object representing counts from the test set
- **unique**: A boolean indicating whether to count zeros for tokens (**unique=False**) or types (**unique=True**)

compare should return two numbers:

- The count of (tokens/types) in the test set that have a zero count in the training set *
- The total number of (tokens/types) in the test set

Confirm that if you call **compare(Counter([1,2,3]), Counter([3,4,4]), unique=True)** you get **(1,2)**, and if you call **compare(Counter([1,2,3]), Counter([3,4,4]), unique=False)** you get **(2,3)**.

The given code has a function called **do_experiment** that calls **get_articles** twice (once for the training data, once for the test data), and then prints the results from **compare** as a markdown table. Read through that function now and make sure that you understand it, since you will add it it later in the lab.

1.5 Questions (10 points)

1. What percentage of the tokens that appear in the **hyperpartisan (True)** articles don't appear in the **neutral (False)** articles?
2. What percentage of tokens that appear in the **neutral (False)** articles don't appear in the **hyperpartisan (True)** articles?
3. What if you look at types instead of tokens?
4. Are you surprised by these results? Why or why not?

1.6 Bigram analysis (5 points)

What happens when you move to higher order n-gram models like bigrams and trigrams?

You can use the **get_bigrams** and **get_trigrams** functions that we wrote in class.

Modify your **get_articles** function so that it returns a tuple with 3 items: a Counter of unigrams, a Counter of bigrams, and a Counter of trigrams. Then modify **do_experiment** so that it generates three table rows with statistics for not only unigram zeros, but also bigram and trigram zeros.

1.7 Questions (4 points)

Using the table generated by **do_experiment**, share the following percentages and analyses:

1. What percentage of the bigrams (tokens, not types) that appear in the hyperpartisan articles don't appear in the neutral articles? What percentage of the bigrams that appear in the neutral articles don't appear in the hyperpartisan articles? Are you surprised by these results? Why or why not?

2. What percentage of the trigrams (tokens, not types) that appear in the hyperpartisan articles don't appear in the neutral articles? What percentage of the trigrams that appear in the neutral articles don't appear in the hyperpartisan articles? Are you surprised by these results? Why or why not?

1.8 Balancing the Data

Instead of training on one category of articles and testing on another, suppose we break each of the categories in half. Then we could train on half of the hyperpartisan articles and half of the neutral articles, and test on the other half.

In the sample data you used above, each of the articles has an attribute **randomchunk** that assigns it to either chunk A or chunk B.

You shouldn't need to write much (any!) code here. Instead of calling **do_experiment** on the hyperpartisan attribute, you can now call it on the randomchunk attribute.

The results here can help us calibrate our sense of how much of the effect we saw in the last part is actually connected to the labels (instead of just a natural property of having lots of text). This relates to a concept called a *permutation test* from statistics, a convenient way to reason about what a statistically significant result is when you don't have a clear indication of which probability distribution describes the variable you're interested in.

1.9 Questions (6 points)

Try training on randomchunk A and testing on randomchunk B. Then train on randomchunk B and test on randomchunk A. How are your results different from the previous question? Why?

Your writeup should include a table of your results, which you can generate with your expanded **do_experiment** function from above. Report percentages, not raw counts.

Evaluating a system in this way is called **cross-validation**: instead of having a specific "held out" test set, you split your training data into k equal-sized parts. Each piece then has one turn being the test set, while the other pieces are assembled together as the training set. In this case, since you are breaking your data into $k = 2$ distinct test sets, you are performing 2-fold cross-validation.

2 N-gram language models, revisited

We learned this week about the problem of **sparsity**, or how to handle zeros in language modeling. If you were to train a unigram language model on the **fiction** category of the Brown corpus and then try to calculate the probability of generating the **editorial** category, you would end up with 0 probability because there are words in the **editorial** text that don't appear in **fiction**.

In this part of the assignment, you'll grapple with this problem in the context of using n-gram language models to analyze textual similarity.

Place your code for this part of the assignment in the **ngram_generate.py** file. It already contains the code for building a trigram language model that we wrote in class.

2.1 Calculating text likelihood

We can use our n-gram language model to quantify the similarity between texts in another way. Given a language model trained on one dataset, we can calculate the **perplexity** of another dataset. (We also use perplexity to evaluate language models.)

Perplexity is the inverse probability of the dataset, normalized by the number of words:

$$PP(W) = \left(\prod_{n=1}^{N} p(w_n | w_{n-2} w_{n-1}) \right)^{-\frac{1}{N}}$$

I have given you a function called **calc_text_perplexity** to compute perplexity. If you read the code, you will notice that I am doing a little trick to avoid numerical stability issues: normalizing by the length of the document before exponentiating the sum of log probabilities.

This function takes a list of file names, a Collection of bigrams, and a Collection of trigrams, and calculates and return the perplexity of the document according to the trigram model.

Check in

Run your **calc_text_perplexity** on the training text. Your perplexity should be around 7.

Run your **calc_text_perplexity** on the austen-sense.txt file. What do you observe?

2.2 Handling out-of-vocabulary words (12 points)

The issue with calculating perplexity on texts outside of our training data is that they may contain trigrams that are not observed during training. The resulting perplexity score doesn't make any sense— your script may terminate in an error, or calculate an extremely low perplexity, because dividing 1 by negative infinity returns zero.

We will add support for out-of-vocabulary words using a very simple technique: add-1 smoothing. For every unseen trigram in our test data, we will pretend that we saw it once during training. We will also pretend that we saw each observed trigram once more during trigram, so that we don't skew things too heavily towards unseen data. (We will use the same policy with bigrams.)

Write a function called **add1_smoothing**. Your function should take a list of texts, a bigram Collection, and a trigram Collection. It should process the document and extend the bigram and trigram Collections with a count of 1 for all n-grams that occur in the test doc, but were not already in the n-gram Collection. It should also add 1 to the counts of each existing n-gram in the Collection.

Use your function to smooth the bigram and trigram counts from our training text with "austen-sense.txt."

Check in

If you smooth the n-grams from our original training doc with "austen-sense.txt" and compare the counts for the first 5 bigrams for the same text before and after smoothing, you should observe counts like the following:

ORIGINAL

```
("[", "emma") 1
("emma", "by") 1
("by", "jane") 3
("jane", "austen") 2
("austen", "1816") 1
```

SMOOTHED

```
("[", "emma") 2
("emma", "by") 2
("by", "jane") 4
("jane", "austen") 3
("austen", "1816") 2
```

Note that these counts all increased by 1 because they were previously observed in the training data. If you check the last 4 bigrams, you will see new bigrams, which were not observed in the training data:

```
("or", "producing") 1
("producing", "coolness") 1
("coolness", "between") 1
("husbands", ".") 1
```

These counts were set to 1, because they were not previously observed in the training data.

2.3 Questions (6 points)

Now that we have added smoothing, you can do some analysis of the Gutenberg data.

1. Of the texts that are not written by Austen, which is the most similar to the Austen training text?
2. There are three authors with three texts apiece in the dataset: Austen, Shakespeare, and Chesterton. Retrain your n-gram model using two of the texts per author, and test on the held-out third text. Which author's writing is the most consistent? Your writeup should include details about which texts you used to train and which to test.

2.4 Generation (7 points)

Your n-gram models can be used to generate novel text, as well as to compare the similarities between texts. You can do this using the **generate_text** function, which takes a string as context, a number of words to generate, a set of bigrams, and a set of trigrams.

If you call **generate_text** with "a woman" and a generation length of 4, your smoothed Austen model should generate: "the woman he had been."

Notice that if you repeat this function call, you will generate the same string every time. That's boring! Instead, we would like to **sample** from the n-gram model while favoring higher probability completions.

Implement a probability-based sampler where next words are sampled according to their probability (rather than always taking the most probable next word.) Call this function **sampler**.

Next, modify your **generate_text** function so that it takes an optional argument, **mode**. Your function should call **predict_next_word** when the mode "best" is passed, and **sampler** when the mode "random" is passed.

2.5 Better Sampling (10 points)

It turns out naive sampling isn't a very good approach for natural language, because of the large number of improbable next words. Each of these has a tiny chance of getting selected, but the sum of those small probabilities can still be significant.

Instead, contemporary language models often use more advanced sampling techniques. Two common techniques are *top-k* sampling, and *top-p* sampling.

Read about each of these methods in Jurafsky & Martin Chapter 8.6: https://web.stanford.edu/ jurafsky/slp3/8.pdf

Choose one of these sampling methods and implement it. Name the function **advanced_sampler**. Call the threshold parameter (k or p) **threshold** in the function signature.

Hint: you might find it easier to work with "normal" probabilities rather than log probabilities. Feel free to exponentiate them using ***math.exp****.*

Modify your **generate_text** function so that supplying the mode argument "advanced" runs this sampler.

3 Curiosity Points (10 points)

As usual, you will receive 90 points for implementing everything described above. To increase your score further, you can extend your investigation of text classification or n-gram language models in some way. You may also focus on extending your analysis of either system.

Please include a section in your report detailing what (if anything) you are submitting for this section.

3.1 Submission

On Gradescope, you should submit your **ngram_classify.py** and **ngram_generate.py** files (5 points for working, non-trivial files) and your report PDF.