COMS W4705 - Natural Language Processing - Homework 1

Due: 4:00pm on Tuesday, February 19th
Total Points: 100

You are welcome to discuss the problems with other students but you must turn in your own work. Please review the academic honesty policy for this course (at the end of the syllabus page).

Create a .zip or .tgz archive containing any of the files you submit. Upload that single file to Courseworks.

The file you submit should use the following naming convention:

YOURUNI_homework1.[zip | tgz]. For example, my uni is db2711, so my file should be named db2711_homework1.zip or db2711_whomework1.tgz.

As a reminder, any assignments submitted late will incur a 20 point penalty. No submissions will be accepted later than 4 days after the submission deadline.

Analytical Component (30 pts)

Write up your solution in a single .pdf or plain (ASCII or UTF-8 encoded) .txt document. Image files and Microsoft Word documents will not be accepted. If you must upload a scan or photo converted into .pdf, make sure that the file size does not exceed 1MB. Name your file written.txt or written.pdf and include it in the zip file you upload to Courseworks.

Problem 1 (15 pts) - Text Classification with Naive Bayes
Consider the following training corpus of emails with the class labels *ham* and *spam*. The content of each email has already been processed and is provided as a bag of words.

Email1 (spam): *buy car Nigeria profit*
Email2 (spam): *money profit home bank*
Email3 (spam): *Nigeria bank check wire*
Email4 (ham): *money bank home car*
Email5 (ham): *home Nigeria fly*

- Based on this data, estimate the prior probability for a random email to be spam or ham if we don't know anything about its content, i.e. P(Class)?
- Based on this data, estimate the conditional probability distributions for each word given the class, i.e. P(Word | Class). You can write down these distributions in a table.
- Using the Naive Bayes' approach and your probability estimates, what is the predicted class label for each of the following emails? Show your calculation.

- *Nigeria*
- *Nigeria home*
- *home bank money*

Problem 2 (15 pts) - Bigram Models
Show that, if you sum up the probabilities of all sentence of length *n* under a bigram language model, this sum is exactly 1 (i.e. the model defines a proper probability distribution). Assume a vocabulary size of *V*.

$$\sum_{w_1, w2, ..w_n} P(w_1, w_2, \ldots, w_n) = \sum_{w_1, w2, ..w_n} P(w_1|start) \cdot P(w_2|w1) \cdots P(w_n|w_{n-1}) = 1$$

Hint: Use induction over the the sentence length.
Comment: This property actually holds for any m-gram model, but you only have to show it for bigrams.

Programming Component - Building a Language Model (70 pts)

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Please make sure you are developing and running your code using Python 3.

Introduction

In this assignment you will be exploring a language model with different orders and smoothing techniques. We will start with a Python script that helps build the models and implement one of the smoothing techniques. You will be asked to run this model and report the results and then compare those to when you add a new smoothing technique. Our *intrinsic* evaluation measure for the model will be 'perplexity', we will use a language identification problem for *extrinsic* evaluation.

(10 pts) Part 1- Getting to know the data and the initial code:

In the homework folder, you can find lang_model.py (the previously mentioned Python script) and the langID_data.zip (the data we're going to use). Let's get to know them.

langID_data.zip:

All the data we'll be using comes from three languages: English, French and German. We want to use to explore perplexity for Language Models (LMs) and text classification.

When you unzip this file you'll find the following 5 files:

HW1english.txt, HW1french.txt and HW2german.txt: for English, French and German text, respectively.

LangID.test.txt: this file has a sentence from a different language in each line.

LangID.gold.txt: this files has the language ID for each of the sentences in LangID.test.txt

lang_model.py:

This code has different parts that when put together can be used to build an n-gram model with and without One-add smoothing and compute the perplexity over a test corpus. Here are its different components:

`get_ngrams`:

is a function that takes a list of strings as input and returns the list of n-grams for an 'n' of choice passed as an argument as well. It also pads the string by adding a 'START' token at the beginning and a 'STOP' at the end. For example:

```
>>> get_ngrams(["natural","language","processing"],1)
[('START',), ('natural',), ('language',), ('processing',), ('STOP',)]
>>> get_ngrams(["natural","language","processing"],2)
('START', 'natural'), ('natural', 'language'), ('language', 'processing'), ('processi
ng', 'STOP')]
>>> get_ngrams(["natural","language","processing"],3)
[('START', 'START', 'natural'), ('START', 'natural', 'language'), ('natural', 'langua
ge', 'processing'), ('language', 'processing', 'STOP')]
```

`corpus_reader`:

This function takes the name of a text file as a parameter and returns a Python generator object. Generators allow you to iterate over a collection, one item at a time without ever having to represent the entire data set in a data structure (such as a list). This is a form of *lazy evaluation*. You could use this function as follows:

```
>>> generator = corpus_reader("")
>>> for sentence in generator:
             print(sentence)

['the', 'fulton', 'county', 'grand', 'jury', 'said', 'friday', 'an', 'investigation',
'of', 'atlanta', "'s", 'recent', 'primary', 'election', 'produced', '``', 'no', 'evid
ence', "''", 'that', 'any', 'irregularities', 'took', 'place', '.']
['the', 'jury', 'further', 'said', 'in', 'term-end', 'presentments', 'that', 'the', '
city', 'executive', 'committee', ',', 'which', 'had', 'over-all', 'charge', 'of', 'th
e', 'election', ',', '``', 'deserves', 'the', 'praise', 'and', 'thanks', 'of', 'the',
'city', 'of', 'atlanta', "''", 'for', 'the', 'manner', 'in', 'which', 'the', 'electio
n', 'was', 'conducted', '.']
['the', 'september-october', 'term', 'jury', 'had', 'been', 'charged', 'by', 'fulton'
, 'superior', 'court', 'judge', 'durwood', 'pye', 'to', 'investigate', 'reports', 'of
', 'possible', '``', 'irregularities', "''", 'in', 'the', 'hard-fought', 'primary', '
which', 'was', 'won', 'by', 'mayor-nominate', 'ivan', 'allen', 'jr', '&', '.']
...
```

Note that iterating over this generator object works only once. After you are done, you need to create a new generator to do it again.

As discussed in class, there are two sources of data sparseness when working with language models: Completely unseen words and unseen contexts. One way to deal with unseen words is to use a pre-defined lexicon before we extract n-grams. The function `corpus_reader` has an optional parameter lexicon, which should be a Python set containing a list of tokens in the lexicon. All tokens that are not in the lexicon will be replaced with a special "UNK" token.

Instead of pre-defining a lexicon, we collect one from the training corpus. This is the purpose of the function `get_lexicon(corpus)`. This function takes a corpus iterator (as returned by `corpus_reader`) as a parameter and returns a set of all words that appear in the corpus more than once. The idea is that words that appear only once are so rare that they are a good stand-in for words that have not been seen at all in unseen text. You do not have to modify this function.

Now, take a look at the `__init__` method of LangModel (the constructor). When a new LangModel is created, we pass in the filename of a corpus file. We then iterate through the corpus *twice:* once to collect the lexicon, and once to count n-grams. You will implement the method to count n-grams in the next step.

`count_ngrams`:

counts the occurrence frequencies for n-grams in the corpus. The method already creates three instance variables of LangModel, which store the unigram, bigram, and trigram counts in the corpus. Each variable is a dictionary (a hash map) that maps the n-gram to its count in the corpus. For example, after populating these dictionaries, we want to be able to query

```
>>> model.trigramcounts[('START','START','the')]

5478

>>> model.bigramcounts[('START','the')]

5478

>>> model.unigramcounts[('the',)]

61428
```

Where *model* is an instance of LangModel that has been trained on a corpus. Note that the unigrams are represented as one-element tuples (indicated by the , in the end). Note that the actual numbers might be slightly different depending on how you set things up.

`smoothed_trigram_probability(self, trigram)` :

which uses linear interpolation between the raw trigram, unigram, and bigram probabilities (see lecture for how to compute this). Set the interpolation parameters to lambda1 = lambda2 = lambda3 = 1/3. Use the raw probability methods defined before.

`sentence_logprob(sentence)`**:**

which returns the log probability of an entire sequence (see lecture how to compute this). You can test it by using the `get_ngrams` function to compute trigrams and the `smoothed_trigram_probability` method to obtain probabilities. Notice how it converts each probability into logspace using `math.log2`. For example:

```
>>> math.log2(0.8)
-0.3219280948873623
```

Then, instead of multiplying probabilities, it adds the log probabilities. Regular probabilities would quickly become too small, leading to numeric issues, so we typically work with log probabilities instead.

`perplexity(corpus)`:

which should compute the perplexity of the model on an entire corpus. Corpus is a corpus iterator (as returned by the corpus_reader method).
Recall that the perplexity is defined as 2^{-l}, where l is defined as:

$$l = \frac{1}{M} \sum_{i=1}^{m} \log p(s_i)$$

Here **M** is the total number of words. So to compute the perplexity, sum the log probability for each sentence, and then divide by the total number of words in the corpus.

Now it's your turn:

The first thing to do is to go through these different components and make sure you understand the implementation. Make sure you get all the details and try out the different functions separately.

Now, using the language ID data, please train a language model for each language. Then do the following:

1/ use *langID.test.txt* to compute the perplexity for n=1,2 and 3 for each language. Report the results in a table similar to this one:

Perpl	Unigram (no smoothing)	Bigram (no smoothing)	Trigram (no smoothing)
English			
French			
German			

***** make sure you modify `sentence_logprob(sentence)` to get the ngrams and probas for each order correctly.**

2/ use *langID.test.txt* together with *langID.gold.txt* together with the trained language models to see how accurately we can classify a sentence into the right language. The trick is to compute, for a sentence 's', the perplexity using each of the trained LMs. The LM with the lower perplexity determines the class of the essay. Report the results in a similar table, i.e. something like this:

Acc	Unigram (no smoothing)	Bigram (no smoothing)	Trigram (no smoothing)
English			
French			
German			

At this point we are *not* going to use the unigram and the trigram models. Starting from this point we'll talk *only* about bigram LMs.

3/ Implement function `smoothed_bigram_probability` which should do the same as `smoothed_trigram_probability` method but for bigrams instead. Once you do that, we want to explore results for smoothed LMs. Use this new method to reproduce the same results as in 1/ and 2/ using linear interpolation smoothing for the ***bigram*** model. Add one column to both perplexity and accuracy tables to report the results.

(15 pts) Part 2- Implementing Add-One smoothing

Now it is your turn to make this script richer. One of the smoothing techniques we talked about in class is the add-one smoothing. Go back to the slides to see how add-one smoothing works. Then implement `add_one_smoothed_bigram_probability` and use it to get a new perplexity and accuracy that you will add to your results table.

(35 pts) Part 3- Implementing Katz back off smoothing

This is same as part 2, the only difference is that we are implementing `katz_backoff_bigram_probability` method that should return bigram probability using the Katz back off technique we visited in class. Again, results for perplexity and accuracy should be added to the result table.

Optional:

`generate_sentence`:

is a function that uses a LM to generate text. Explore this function to see how you can use an LM to generate text, you can also take it one step further and generate many sentences and compute the perplexity to see what it'll look like. It should be very low, but what if you compare it to the perplexity when you compute it on the same training set you used to build the LM? Which one is lower?

(10 pts) Part 4- Quantitative error analysis

What you have produced so far is a wealth of information about this corpus. You've looked it through different perspectives by varying the smoothing technique, the n-gram order and you have used both an extrinsic and an intrinsic measure to compare the results.

Now it's time to put it all together. Given what we have seen in class and the numbers that you have produced here, please write two or three paragraphs in which you can describe how the explored numbers are related and why is it that from one language to another the numbers can be so different.