15-110 Hw6 - Language Modeling

Hw6 and its check-ins are organized differently from the other assignments. If you haven't already done so, you should read the **Hw6 General Guide** to understand how this assignment works.

Project Description

For this project, you will be building your own language model! Given the works of two famous folklore and fairytale authors, Andersen and the Grimm Brothers, you will create a model of the language used in these mystical works. The files are already cleaned and imported in the starter file for you, so you can dive right into the analysis!

Background

Language modeling plays a key role in many successful natural language processing applications, including but not limited to, machine translation, automatic speech recognition, handwriting recognition, spelling corrections, etc. Many of the cool applications you use on a daily basis actually have a language modeling component working at the backend. Amazon Alexa, Apple Siri, and Google Translate are all examples of these applications, which makes this project extremely relevant to today's technology!

Click on the following links to read the instructions for each week's assignment:

Hw6 Check-in 1 - due Monday 11/18

Hw6 Check-in 2 - due Monday 11/25

Hw6 - due Wednesday 12/04

Hw6 Check-in 1 - due Monday 11/18

In this task, you will implement three different language models - Uniform, Unigram, and Bigram. Below are the descriptions of each model:

- Uniform model: a model where all words get the probability 1/V, where V is the size of the vocabulary.

- Unigram model: a model where each word gets the probability C1/N, where C1 is the number of times that word appears in the data and N is the total number of words in the data. In other words, each word's probability depends on how often that word shows up in the data.

- Bigram model: a model where each pair of words `x y` gets the probability C2/N1, where C2 is the number of times that `x y` appears in the data (in that specific order) and N1 is the number of times that `x` appears in the data.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Load Words From Text **[5pts]**

Write a function load_book(filename) that takes a filename, opens it, and creates a 2D list where each row is a sentence in the book, and each column is one word or symbol in that sentence in order. The 2D list is called a corpus of text. The file you are given is already cleaned - each line is a single sentence, all words are lowercases, and all punctuation is separated out by spaces. You should read in the file, and for each line, split it by spaces, and append that list to your list of lists. For example,

"hello and welcome to 15-110 .
we're happy to have you ."

would return a 2D list

[["hello","and","welcome","to","15-110","."],
["we're","happy","to","have","you","."]]

To test your function, run test_load_book().

Step 2: Find Corpus Length **[5pts]**

Write a function get_corpus_length(corpus) which is given a 2D list of words and symbols in a text and returns the total number of single words or punctuation symbols (called unigrams), e.g., "hello", ",", or "world".

corpus = [["hello", "world"],
["hello", "world", "again"]]
get_corpus_length(corpus) -> 5

In the example from load_book, get_corpus_length would return 12.

To test your function, run test_get_corpus_length().

Step 3: Build Unigram Vocabulary **[10pts]**

Given a 2D list corpus, write a function build_vocabulary that creates a new list and iterates through the 2D list adding only words that are not present in the new list already. In other words, it should return a list of all unique unigrams.

corpus = [["hello", "world"],
["hello", "world", "again"]]
build_vocabulary(corpus) -> ["hello", "world", "again"]

corpus = [["hello","and","welcome","to","15-110","."],
["we're","happy","to","have","you","."]]
build_vocabulary(corpus) -> ["hello","and","welcome","to","15-110",".","we're",
"happy","have","you"]

To test your function, run test_build_vocabulary().

Step 4: Count Unigrams **[10pts]**

Given a 2D list corpus, write a function count_unigrams(corpus) which creates a new dictionary with keys as each item in build_vocabulary(corpus) and values initialized or set to 0. Then, iterate through the 2D list and for each unigram, add one to the count of the number of times it was seen. Return the dictionary with the counts of each unique

unigram in the corpus. Don't forget: a unigram is a single word or punctuation symbol, e.g., "hello", ",", or "world".

corpus = [["hello", "world"],
["hello", "world", "again"]]
count_unigrams(corpus) -> { "hello": 2, "world": 2, "again": 1 }

To test this function, run test_count_unigrams().

Step 5: Count the Start Words **[10pts]**

We'll need to keep track of 'start' words; that is, words that start sentences.

Given a 2D list corpus, write a function get_start_words that creates a new list and iterates through the 2D list, adding only the first words in the sentence that it has not seen before. In other words, it should return a list of all unique first words.

corpus = [["hello", "world"],
["hello", "world", "again"]]
get_start_words(corpus) -> ["hello"]

corpus = [["hello","and","welcome","to","15-110","."],
["we're","happy","to","have","you","."]]
get_start_words(corpus) -> ["hello","we're"]

To test this function, run test_get_start_words().

Then, given a 2D list corpus, write a function count_start_words(corpus) which creates a new dictionary with keys as each item in get_start_words(corpus) and values initialized or set to 0. Then, iterate through the 2D list and for each start word, add one to the count of the number of times it was seen. Return the dictionary with the counts of each unique start word in the corpus.

corpus = [["hello", "world"],
["hello", "world", "again"]]
count_start_words(corpus) -> { "hello": 2 }

To test this function, run test_count_start_words().

Step 6: Count Bigrams **[15pts]**

A bigram is a pair of words that appear next to each other. Often, looking at the frequency of pairs of words helps us know more about what word goes next in a sentence instead of only looking at single words (unigrams).

Write a function count_bigrams(corpus) that returns a 2D dictionary with the counts of each unique bigram in the corpus. In order to do this, you will:

- Step 1: create a new dictionary
- Step 2: iterate through each sentence of the corpus.
- Step 3: For each sentence, loop through the indexes of the words in each sentence from 0 to n-1 (don't include the last word). With a word at index i, each pair of words will be sentence[i] and sentence[i+1].
- Step 4: If sentence[i] is not in your main dictionary, add it with the value as an empty dictionary.
- Step 5: Check if sentence[i+1] is in sentence[i]'s dictionary. If it is not, add sentence[i+1] as the key and value 1. If it is in the dictionary, add 1 to the count.

This function should return the dictionary of dictionaries (2D dictionary) representing the counts of all bigrams of words.

corpus = [["hello", "world"],
["hello", "world", "again"]]
bigram_count = count_bigrams(corpus)
bigram_count -> { "hello": { "world": 2 },
"world": { "again": 1 } }
bigram_count["hello"]["world"] -> 2

To test this function, run test_count_bigrams().

Once you've finished all six steps, you can test your functions on larger files! Try running do_week1() and printing out the values held in the variables to see what you get.

Hw6 Check-in 2 - due Monday 11/25

Using the different models, you will now write functions that allow users to analyze the model by looking at the frequency and probability of each word. Then you will write functions that let the user generate new text based on the language models of the books.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Compute Uniform Probabilities **[5pts]**

Since we can compute the list of all of the words in the corpus using build_vocabulary(corpus), we could think of predicting the next word in the text as just a random choice of these words.

Write a function build_uniform_probs(unigrams) which takes a list of unique unigrams and returns a new list of the same length where the value is 1/len(unigrams) at each index. Each index of this new list represents the probability that the corresponding index in unigrams would be chosen at random. Return the list of probabilities.

To test this function, run test_build_uniform_probs().

Step 2: Compute Unigram Probabilities **[10pts]**

Another way to predict the words in the text is by how frequently they occur. You computed the counts of all the unigrams last week. This week we will use them to compute the probability of that word being randomly chosen from the whole book.

Write a function build_unigram_probs(unigrams, unigram_counts, total_count) which takes a list of all of the unique words in the book, a dictionary mapping unique unigrams to counts, and the total count of words in the book, and returns a new list of the probabilities of each word.

In order to do this, you should:

1. make a new empty list
2. iterate through the indexes of the unigram list

a. look up the count of the corresponding unigram in the unigram_counts,
b. divide count/total_count, and
c. append that probability to the list.

The probability at index i will be the probability that the word at the same index in unigrams would be chosen at random from the book. Return the list of probabilities.

To test this function, run test_build_unigram_probs().

Step 3: Compute Bigram Probabilities **[10pts]**

Finally, we can also think about how the probability of one word changes based on the word before it. Think about the word "Happy". The word "birthday" and "Halloween" probably occur at a much higher rate after "Happy" than after other words.

Write a function build_bigram_probs(unigram_counts, bigram_counts) that takes the frequencies of single words (unigram_counts) and of pairs of words (bigram_counts) and returns a new 2D dictionary of the probabilities of a second word being chosen after the first.

1. Make a new dictionary
2. Iterate through each key prev_word in bigram_count
    a. bigram_count[prev_word] is a dictionary of all the words that occurred after prev_word in the book.
    b. make two new lists, one for the words (keys) in bigram_count[prev_word], and one for the probabilities of those words
    c. iterate through all of the keys in bigram_count[prev_word], appending the key to the key list and the key's value count divided by unigram_count[prev_word] to the probability list.
        i. Note 1: unigram_count[key] represents the total # of times the prev_word occurred. That total is divided by all the words that were seen after it. One of those words is key, so the probability of that word is the count of how many times it was seen after prev_word / count of prev_word.
        ii. Note 2: this isn't 100% accurate, because we might have fewer total word occurences in bigram_count than unigram_count if a prev_word occurred at the end of a sentence. But this usually only happens to punctuation, so we'll say this is good enough for now.
    d. make a dictionary mapping the word "words" to the key list and the word "probs" to the list of probabilities.

    e. add to the new dictionary the key prev_word; the value is the dictionary from part d
3. Return the new dictionary.

To test this function, run test_build_bigram_probs().

Step 4: Get Most Likely Words **[10pts]**

Write a function get_top(count, words, probs, ignorelist) which takes an int count, a list of words, and the list of the corresponding probabilities. It finds the top count highest probabilities, then returns a dictionary mapping those highest probability words to their probabilities as values. Only include words that aren't in the list of words ignorelist.

One way to do this is to create an empty dictionary which will hold the known highest probability words, then repeatedly search for the highest probability word that is not already a key in that dictionary and not in the ignorelist. When you finish iterating to find the highest probability word, add it to the dictionary. Continue the process until the length of the dictionary == count.

To test this function, run test_get_top().

Step 5: Generate Sentences with Unigrams **[10pts]**

Write a function generate_text_unigrams(count, words, probs) which takes in a count of the number of words to generate, the word list, and the corresponding probabilities, and returns a string created by concatenating probabilistically-chosen words together.

Use the function choices(words, weights=probs) from the random library to generate a random word sampled according to the given probability distribution. choices returns a list containing the word it picked.

Don't forget to add spaces between words! And return the text once you've added count words.

To test this function, run test_generate_text_unigrams().

Step 6: Generate Sentences with Bigrams **[10pts]**

Write a function generate_text_bigrams(count, start_words, start_word_probs, bigram_probs) which takes in a count of the number of words to generate, the start

word list, the start word probabilities, and the bigram probabilities dictionary, and outputs a string of count words generated probabilistically.

In order to generate words, we follow one of two cases:

- First, if nothing has been seen yet or a period has been seen indicating the end of a sentence, use the choices function with start_words and start_word_probs to generate a new word.

- Otherwise, look up in the bigram_probs dictionary the last word generated. Use the "words" and "probs" lists in the dictionary as parameters to choices, then append the word onto the text you've generated so far.

Don't forget to include spaces between words! And return the text once you've added count words.

To test this function, run test_generate_text_bigrams().

Once you've finished all six steps, you can test your functions on larger files! Try running do_week2(), and observe the most frequent words, and the text produced by your functions (which is based on text from the Grimm and Andersen files combined).

Hw6 - due Wednesday 12/04

In the final stage, you will use matplotlib and the functions you wrote for the earlier assignments to generate graphs that compare and contrast different books.

Step 0-A: Complete Check-in 1 **[20pts]**

If you got a perfect score on Hw6 Check-in 1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 1 and use it to update your Check-in 1 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 0-B: Complete Check-in 2 **[20pts]**

If you got a perfect score on Hw6 Check-in 2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 2 and use it to update your Check-in 2 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 1: Install matplotlib and numpy **[0pts]**

In order to use the matplotlib library, you will need to install it on your machine. If you do not have a personal computer, note that the cluster machines on Gates 5 should have matplotlib installed already.

To install matplotlib and one of its dependencies, numpy, we recommend that you use the pip tool included in your Python installation. This tool will manage the installation process for you, which is much easier than trying to install a module manually. To use pip, open Terminal on a Mac/Linux or PowerShell on Windows. Then run the following lines of code:

pip install numpy
pip install matplotlib

If an error message occurs, try googling it to find a solution. TAs can also help debug installation errors via Piazza or in office hours.

Note that pip is associated with the default version of python on your computer. If you have multiple versions installed (which is often true of Macs, as they come with Python 2.7 installed by default), you will have to run a different command to install the libraries into the version of Python you use. These commands might work:

python3 -m pip install numpy
python3 -m pip install matplotlib

You can test whether the modules have installed into the correct version of Python by running the following commands in your interpreter. If they do not give you an error, you're good to go!

import numpy
import matplotlib

Step 2: Review Provided Code **[0pts]**

Drawing graphs with matplotlib requires a lot of setup code, and we want you to draw quite a few graphs, so we've provided a few functions for you to use, to simplify things. You will write some functions that call the functions we've provided.

You should definitely know what each of the following functions do, and we recommend that you look over their code at the bottom of the file quickly, to get a sense of how they work.

- **bar_plot:** generates a bar chart. The x values are the keys in the dictionary; the y values are the key's values.
- **sidebyside_bar_plots:** generates two bar charts side-by-side, for easy comparison. The x values are the values in the names list, and the y values for the two plots are the values in the values1 and values2 lists.
- **scatter_plot:** generates a scatter plot. The x values are the values in xs, and the y values are the values in ys.

Step 3: Graph the Top 50 Words **[10pts]**

Write a function graph_top_50_words(corpus) which takes as input a corpus, uses the functions from previous weeks to compute the unigrams and the unigram probabilities, and then computes the top 50 most frequent words according to the probabilities (using

get_top and the global ignore list defined above to eliminate common words with no meaning).

Then, using the most common 50 words and their probabilities, calls our bar_plot function with a good title to display the words.

To test this function, run run_week3() and check the first graph that is generated.

Step 4: Graph the Top Starting Words **[10pts]**

Write a function graph_top_start_words(corpus) which takes as input a corpus and uses the functions from previous weeks to compute the start words and the start word probabilities, then computes the top 50 most frequent start words according to the probabilities (using get_top and the global ignore list defined above to eliminate common words with no meaning).

Then, using the most common start words and their probabilities, call our bar_plot function with a good title to display the words.

To test this function, run run_week3() and check the second graph that is generated.

Step 5: Graph the Top Bigrams **[15pts]**

Write a function graph_bigram_words(corpus, word) which takes a corpus and word and graphs the top 10 words that appear after that word in the corpus, not counting words in the global ignore list.

Hint: what does the build_bigram_probs compute? What does the get_top function compute? Can you use those in conjunction with each other to get the top 10 words that appear after a word?

To test this function, run run_week3() and check the third, fourth, fifth, and sixth graphs that are generated. The third and fifth graphs represent bigrams collected from the Grimm text; the fourth and sixth represent bigrams from Andersen. Try comparing them to find similarities and differences!

Step 6: Compare Probabilities Across Two Books **[25pts]**

As the final step of the project, we'll compare the probabilities of the most common words in two books with two approaches - a side-by-side bar chart, and a scatterplot.

First, write a function graph_sidebyside(corpus1, name1, corpus2, name2, title) which takes two corpuses (corpora) and their names. Then it should find the top 50 unigrams in corpus1 (ignoring words in the ignore list), the corresponding probabilities in corpus1, and also a list of the corresponding probabilities in corpus2. It should do the same with corpus2 for the top 50 words in corpus2 (again ignoring words in the ignore list), again finding the probabilities in corpus2 and in corpus1.

The function should then combine these two sets of lists to make one result with the union of the top 50 words in corpus1 and the top 50 words in corpus2 (noting that some words might appear in both, and should not be repeated). Finally, it should use sidebyside_bar_plots to show the differences in probabilities across those top elements. Use name1 and name2 as the category names, and don't forget to include the title- this time, it's provided for you.

Then, write a function graph_scatterplot(corpus1, corpus2, title) which takes two corpuses (corpora). It should find the top 30 unigrams in corpus1 and the corresponding probabilities in corpus1, and also a list of the corresponding probabilities in corpus2. It should do the same with corpus2 for top 30 words in that corpus, again finding probabilities for both corpora. Finally, it should use the scatter_plot function to show the probabilities of corpus1 on the x axis and corpus2 on the y. Don't forget to include the title- this time, it's provided for you.

Note: when calling scatter_plot, the parameters x, y, and labels must be lists.

You can test these final two functions by running run_week3() and checking the final two charts that are generated.

Now you're done! Enjoy looking through the charts and models you've made to find interesting features of the books you've analyzed.

If you want to analyze more books beyond what we've provided, you can do that too! Search for the text of the book you want online, then download it into a .txt file with _raw at the end of the name, and put that file in the data/ folder. Then run clean_files.py. This will generate a new version of the file with _clean at the end, which you can use with your own code!