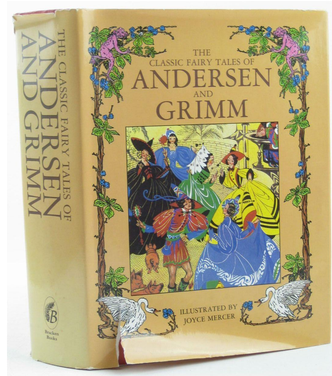
15-110 Hw6 - Language Modeling

Hw6 and its checks 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



In this project, you will build and test your own language model based on the works of two famous folklore and fairytale authors, Andersen and the Grimm Brothers. A **language model** is a machine learning model that holds information about how common certain words are in a specific author's books. You'll use the language model you create to generate text that is similar to the texts written by these two authors, and compare how often they used different words and phrases.

In the first week, you will parse the text of the provided corpuses into new formats, to make generating language models possible. In the second week, you'll implement three simple language models: a uniform model, a unigram model, and a bigram model. You'll compute probabilities and generate new text using these models. In the third week, you'll create visualizations that compare the two texts based on these models.

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

Check6-1 - due Monday 04/10 at noon EST
Check6-2 - due Friday 04/21 at noon EST
Hw6 - due Friday 04/28 at noon EST

Check6-1 - due Monday 04/10 at noon EST

In the first stage, you will reformat two text corpuses (from Andersen and Grimm) to build data that you'll use to generate language models. To do this, you will need to write functions that build the vocabulary of a text, count the number of **unigrams** (individual words) that appear, and count the number of **bigrams** (pairs of words) that appear.

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 the course website.

Step 1: Load Words From Text **[10pts]**

Write a function `loadBook(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; then append that list to your list of lists (unless it's empty, in which case it should be discarded). For example,

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

would produce a 2D list

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

To test your function, run `testLoadBook()`.

Note: to keep the starter file from becoming too crowded, all the tests have been moved to the file `hw6_language_tests.py`, which you can open and read while debugging. The functions in this file are called at the bottom of `hw6_language.py`. **If you change the filename of `hw6_language.py`, you will need to change the filename imported by `hw6_language_tests.py` in the first line of the file too**.

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

Write a function `getCorpusLength(corpus)` which is given a 2D list of words and symbols in a text and returns the total number of **unigrams** (single words or symbols) that occur in that list. Examples of unigrams include `"hello"`, `","`, and `"world"`.

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

In the example from `loadBook`, `getCorpusLength` would return `12`.

To test your function, run `testGetCorpusLength()`.

Step 3: Build Unigram Vocabulary **[5pts]**

Given a 2D list `corpus`, write a function `buildVocabulary(corpus)` that creates a new list and iterates through the corpus, 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"] ]
buildVocabulary(corpus) -> ["hello", "world", "again"]

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

To test your function, run `testBuildVocabulary()`.

Step 4: Track the Start Words **[5pts]**

For the next model, we'll need to keep track of **start** words; that is, words that start sentences.

Given a 2D list `corpus`, write a function `makeStartCorpus(corpus)` that returns a new corpus only containing the starting word of each sentence. This corpus should still be a 2D list, but will only have one word per sentence - the first word.

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

corpus = [ ["hello","and","welcome","to","the","program","."],
           ["we're","happy","to","have","you","."] ]
makeStartCorpus(corpus) -> [ [ "hello" ], [ "we're" ] ]
```

Step 5: Count Unigrams **[5pts]**

Now we'll organize the data to later support a unigram model. A **unigram model** is a model where each word gets the probability X/N, where X 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.

Given a 2D list `corpus`, write a function `countUnigrams(corpus)` which returns a dictionary where the keys are the unigrams in the corpus's vocabulary and the values are how often each key occurs in the actual corpus (the 2D list).

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

To test this function, run `testCountUnigrams()`.

Step 6: Count Bigrams **[10pts]**

Our final model will use bigrams instead of unigrams. A **bigram** is a pair of words that appear next to each other. Often, looking at the frequency of pairs of words helps us more accurately predict which word goes next in a sentence, as opposed to only considering the general frequencies of words.

A **bigram model** is a model where each pair of words `"x y"` gets the probability X/N, where X is the number of times that `"x y"` appears in the data (in that specific order) and N is the number of times that `"x"` appears in the data.

Write a function `countBigrams(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 that sentence from `0` to `len(sentence)-1` (don't include the last word). Each pair of words will be `sentence[index]` and `sentence[index+1]`.
- Step 4: If `sentence[index]` is not in your main dictionary, add it with an empty dictionary as the value.
- Step 5: Check if `sentence[index+1]` is in `sentence[index]`'s dictionary. If it is not, add `sentence[index+1]` as a key with the value `1`. If it is in the dictionary, add `1` to its associated value.

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

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

To test this function, run `testCountBigrams()`.

Step 7: Clean Data - Separate Words **[10pts]**

You may want to run your code on your own datasets (instead of the datasets we've provided). To make that possible, you'll first need to **clean** the data you want to analyze, so it can fit the needed format.

First, write the function `separateWords(line)` which takes a line of text (a string) and returns a list of all the words in the text with punctuation separated out. However, you do *not* want to separate the **'** punctuation marks, as they are usually used as apostrophes and count as part of the words they are in.

```
line = "Hello and welcome to the program, we're happy to have you!"
separateWords(line) -> ['Hello', 'and', 'welcome', 'to', 'the',
'program', ',', "we're", 'happy', 'to', 'have', 'you', '!']
```

Importantly, it is possible for a single token to contain **multiple punctuation marks**. The following example provides an extreme case of this:

```
line = '"Wait-no!", she cried.'
separateWords(line) -> ['"', 'Wait', '-', 'no', '!', '"', ',', 'she',
'cried', '.']
```

To solve this problem, use the following algorithm:

- Break the text into tokens by splitting over spaces.
  - For each token, keep track of a 'start' variable to mark the beginning of the current word
  - Iterate over each character in the given token
    - If the character is a punctuation mark, but not an apostrophe…
      - Add two new words to the result list: the characters from the current start point up to and not including the punctuation mark, then the punctuation mark by itself.
      - Then update the start position to be the next position after the punctuation mark
  - When you reach the end of the token, add any remaining letters (from the current start point to the end of the token) as a new word to the result

Hint: how can you tell if a character is a punctuation mark? Use the library variable `string.punctuation`

Another hint: you may end up with extra empty strings in your result list. Make sure to remove all of them from your result before you return!

To test this function, run `testSeparateWords()`.

Step 8: Clean Data - Compose Text **[5pts]**

Second, write the function `cleanBookData(text)` which takes a string containing the book data you want to analyze and returns a cleaned version of the string. The cleaned version should have the following changes made:

- All words should be made lowercase
- Every line of text should be broken up into words and punctuation marks (by calling `separateWords` on the line of text)
- Every line of text should further be broken up so that each sentence is on its own line

To break up the sentences, iterate over the tokens in each separated line looking for the characters `"."`, `"!"`, and `"?"`, which all end sentences. You can use a similar approach to separate the sentences as you used to separate punctuation from words.

To turn a list of words back into a sentence, try using the `join` method on a delimiter string of `" "`. Don't forget to add newlines after each line and strip any extra newlines from the end! And make sure to call `join` on each individual line, not the whole text - if you call it on the whole text, you'll end up with extra spaces before and after each newline.

```
book = """Hello and welcome to the program, we're happy to have you!
Your first task, writing some code, is easy. Let's start!
Just use the print command with "Hello World" in parentheses."""

cleanBookData(book) ->
"""hello and welcome to the program , we're happy to have you !
your first task , writing some code , is easy .
let's start !
just use the print command with " hello world " in parentheses ."""
```

To test this function, run `testCleanBookData()`.

Note: if you run these functions on your own real data, you'll probably need to add more cleaning steps, some automated and some done by hand. For example, if you're using a book from Project Gutenberg, paragraphs will be broken into lines with empty lines in between them; you need to combine the paragraphs together to process the data correctly. Every dataset is unique!

Once you've finished all seven steps, you can test your functions on larger files! Try printing the variables in `runWeek1()` to see what you get.

Check6-2 - due Friday 04/21 at noon EST

In the second stage, you'll actually generate the three language models (uniform, unigram, and bigram), which will make it possible to identify the most common words in the vocabularies and generate new text. To do this, you will first need to write functions that produce probabilities for each word in the corpus.

Before you start this second stage, go to the bottom of the starter file and uncomment the four test lines associated with Week 2.

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 the course website.

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

Our final language model, the **uniform model**, is a model where all words get the probability 1/N, where N is the size of the vocabulary. Since we can compute the list of all of the words in the corpus using `buildVocabulary(corpus)`, the uniform model can predict the next word in a text by just randomly choosing one of the words.

Write a function `buildUniformProbs(unigrams)` which takes a list of unique unigrams and returns a new list of the same length where `1/len(unigrams)` is the value 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 `testBuildUniformProbs()`.

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. Now we will use those counts to compute the probability of each word being randomly chosen from the whole book.

Write a function `buildUnigramProbs(unigrams, unigramCounts, totalCount)` 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 make a new empty list, then iterate through the indexes of the unigram list. Look up the count of the index's corresponding unigram in `unigramCounts`, then divide the unigram count by the `totalCount` and append that resulting probability to the list.

The probability at each index 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 `testBuildUnigramProbs()`.

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 words "birthday" and "Halloween" probably occur at a much higher rate after "Happy" than after other words.

Write a function `buildBigramProbs(unigramCounts, bigramCounts)` that takes the frequencies of single words (`unigramCounts`) and of pairs of words (`bigramCounts`) and returns a new **nested dictionary**.

Each key of the dictionary is a word in the vocabulary, and each key maps to the bigram probability dictionary for that word. A single word's bigram probability dictionary is like a reduced version of the unigram probability results- it keeps track of all the words that can come after the 'start' word, and the probability for each of them.

As an example, if we're processing the sentence `"It is very nice, it is super cool"`, our result dictionary would look like this:

```
{ "it"    : { "words" : ["is"],            "probs" : [1]        },
  "is"    : { "words" : ["very", "super"], "probs" : [0.5, 0.5] },
  "very"  : { "words" : ["nice"],          "probs" : [1]        },
  "nice"  : { "words" : ["it"],            "probs" : [1]        },
  "super" : { "words" : ["cool"],          "probs" : [1]        } }
```

Note that `"cool"` is not included as a key in the outer dictionary because it is never the first word in a bigram.

To create this dictionary, follow these steps:

1. Make a new dictionary
2. Iterate through each key (we'll call it `prevWord`) in `bigramCounts`
   a. note that `bigramCounts[prevWord]` is a dictionary of all the words that occurred after the previous word in the book
   b. make two new lists, one for the words (keys) in `bigramCounts[prevWord]`, and one for the probabilities of those words
   c. iterate through all of the keys in `bigramCounts[prevWord]`, appending the word to the word list and the word's probability to the probability list.
      i. Note 1: You can determine the probability by dividing the count by `unigramCounts[prevWord]`, which is the total number of times the previous word occurred.
      ii. Note 2: this isn't 100% accurate, because we might have fewer total word occurrences in `bigramCounts` than `unigramCounts` if `prevWord` 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 temporary dictionary mapping the string `"words"` to the word list and the string `"probs"` to the list of probabilities.
   e. add to the new dictionary (from the outer level) the key `prevWord`; the value is the dictionary from part d. In other terms, each previous word maps to a dictionary containing words and probabilities.
3. Return the new dictionary.

To test this function, run `testBuildBigramProbs()`.

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

Write a function `getTopWords(count, words, probs, ignoreList)` which takes a number of words, a list of words, and a list of the corresponding probabilities, plus a list of words to ignore. It finds the top `count` words with the highest probabilities, then returns a dictionary mapping those highest probability words to their probabilities. Only include words that aren't in the list of words `ignoreList`.

For example, if you call `getTopWords` with `count` set to `10`, it will return a dictionary mapping the top 10 words to their probabilities.

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 `ignoreList`. Once you've gone through all the words and found the highest probability word that meets those requirements, add it to the dictionary. Continue the process until the length of the dictionary is equal to `count`.

Note: do not destructively change `ignoreList`! It will be reused across function calls.

To test this function, run `testGetTopWords()`.

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

Now we can use the language models to do more interesting things! Write a function `generateTextFromUnigrams(count, words, probs)` which takes in the number of words to generate, the word list, and the corresponding probabilities, and returns a string generated 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, so you'll need to index into the first element of that list.

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

To test this function, run `testGenerateTextFromUnigrams()`.

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

The text generated from the unigram model kind of looks like gibberish, but we can do better. Write the function `generateTextFromBigrams(count, startWords, startWordProbs, bigramProbs)` which takes in 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 added to the string yet, or the last word added was a period, exclamation mark, or question mark (indicating the end of a sentence), use the `choices` function with `startWords` and `startWordProbs` to generate a new word and add it to the text.

- Otherwise, look up the last word generated in the `bigramProbs` dictionary. Use the `"words"` and `"probs"` lists in the dictionary as parameters to `choices`, then add the returned 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 the number of words specified by `count`.

To test this function, run `testGenerateTextFromBigrams()`.

Once you've finished all six steps, you can test your functions on larger files! Try running `runWeek2()`, 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 Friday 04/28 at noon EST

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

Before you start this last stage, go to the bottom of the starter file and uncomment the two test lines associated with Week 3.

Step 0-A: Complete Check6-1 **[20pts]**

If you got a perfect score on Check6-1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check6-1 and use it to update your Check6-1 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 Check6-2 **[20pts]**

If you got a perfect score on Check6-2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check6-2 and use it to update your Check6-2 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 'Manage Packages' feature in Thonny. This tool will manage the installation process for you, which is much easier than trying to install a module manually.

Once you're in the 'Manage Packages' prompt, search for **numpy**, click on the link titled **numpy**, then click on 'Install'. Repeat the process with the search term **matplotlib**.

If you're using a non-Thonny IDE, you'll need to use the pip command instead. Open your computer's terminal and run the following commands:

```
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.

You can test whether the modules are correctly installed 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.

- **barPlot:** generates a bar chart from a dictionary. The x values are the keys in the dictionary; the y values are the key's values.
- **sideBySideBarPlots:** generates two bar charts side-by-side, for easy comparison. The x values are the values in the `xValues` list, and the y values for the two plots are the values in the `values1` and `values2` lists.
- **scatterPlot:** generates a scatter plot with a line marking where x = y. 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 `graphTop50Words(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 `getTopWords` and the global ignore list defined above to eliminate common words with no meaning).

Then, using the most common 50 words and their probabilities, call the `barPlot` function with a good title to display the words.

To test this function, run `runWeek3()` and check the first graph that is generated. The first word should occur a lot more than the rest, and then there should be a downwards slope.

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

Write a function `graphTopStartWords(corpus)` which takes as input a corpus and uses the functions from previous weeks to compute the start words, find the start word probabilities, then computes the top 50 most frequent start words according to the probabilities (using `getTopWords` 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 the `barPlot` function with a good title to display the words.

To test this function, run `runWeek3()` and check the second graph that is generated. The first three words should show a probability above 2%; the rest should slope downwards from there.

Step 5: Graph the Top Bigrams **[10pts]**

Write a function `graphTopNextWords(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: use `buildBigramProbs` to get all possible next words after `word`, and `getTopWords` to narrow them down to the top 10.

To test this function, run `runWeek3()` 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 **[30pts]**

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

First, write a function `setupData(corpus1, corpus2, topWordCount)` which takes two corpuses (corpora) and a number of top-words to count. It should find the top n unigrams in `corpus1` (ignoring words in the ignore list), and the top n unigrams in `corpus2` (again ignoring words in the ignore list). Combine these lists together to make one top-words list. Note that some words may appear in both, and should not be repeated. For example, if `topWordCount` was set to 50, the combined list could have anywhere from 50 (all overlap) to 100 (no overlap) words.

Then find the probabilities of all the top words in `corpus1`, and put them in a list. You can do this by finding the index of the word in the main unigrams list, then indexing into that location in the unigram probability list. Make another list of probabilities in `corpus2`. If a word does not occur in a corpus, it should be assigned the probability `0`.

When you've created the three lists - the top words and the two probability lists - return all three by using a dictionary. This dictionary should map the key `"topWords"` to the top-words list, the key `"corpus1Probs"` to the `corpus1` probabilities, and the key `"corpus2Probs"` to the `corpus2` probabilities. We'll use this data to draw charts in a moment.

To test this function, run `testSetupData()`. This is called for you in `runWeek3()`.

Now we can finally draw the charts! write the function `graphTopWordsSideBySide(corpus1, name1, corpus2, name2, numWords, title)` which takes two corpuses (corpora) and their names, the number of words to graph, and a title. Call `setupData` on the corpora, looking for the top `numWords` words, to acquire the necessary data. Then call `sideBySideBarPlots` to show the differences in probabilities across those top unigrams. Use `name1` and `name2` as the category names, and the provided title.

Finally, write a function `graphTopWordsInScatterplot(corpus1, corpus2, numWords, title)` which takes two corpuses (corpora). It should also call `setupData` on the provided values to gather the probability data for the `numWords` top words. Once you've produced the probability lists, use the `scatterPlot` function to show the probabilities of `corpus1` on the x axis and `corpus2` on the y axis. Again, make sure to include the title that is provided as a parameter.

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

You can test these final two functions by running `runWeek3()` and checking the final two charts that are generated. In particular, for the side-by-side barplot, make sure it has a y range from 0 to 0.0175, and that one of the words (greda) only has a bar for Andersen, not for Grimm. For the scatterplot, the word 'he' should be separated from the others a bit, and in the main cluster the word 'i' should be above the x=y line, while 'she' is below it (Andersen used 'I' more often; Grimm used 'she' more often).

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. Read the text into a string, then run `cleanBookData` on it to create the proper book format. Save that. Save the book into a new file, then run `loadBook` on the new file to generate a corpus. You can call your functions on the corpus from there!