Homework 9

Due: 16th April, 2021 at 11:59pm ET

This homework asks you to do basic textual analysis to study the n-gram distribution of different languages, and examine a "mystery" text to determine what language it is in.

You will then perform a TF-IDF analysis

Goals

In this assignment you will:

- Write functions to perform a basic n-gram analysis of texts
- Visualize the n-gram distribution of different texts
- Use this information to analyze a new text.
- Compute TF-IDF scores for a set of documents, then find the most distinctive words.

Background

N-grams

In this week's lecture we discussed n-grams: a way of breaking up text into smaller chunks to analyze their distribution. n-grams are defined over either words in a sentence (so the 3-grams in "words in a sentence" are "words in a" and "in a sentence") or over the characters in a sentence (so the 3-grams in "sentence" are "sen" "ent" "nte" "ten" "enc" "nce"). In this homework, we will use characters.

It is common to "pad" strings with dummy characters (e.g., `_`) to produce more useful n-grams (to correctly capture, for example, that 's' is the most common letter to start words through the 3-gram `__s`). The easiest way to do this is to alter the sentence that you're building n-grams over by adding `_` s to the beginning and end.

The interesting feature about n-grams is that different languages have different *distributions* of n-grams. (In the simple case, the most common letter in English is `e`, but the most common letter in Spanish is `a`. The "1-grams" for English and Spanish have different distributions). This homework will build *3-gram* distributions for texts in different languages, and you will then use this information to make inferences about a "mystery" text.

TF-IDF

We also discussed TF-IDF as a metric to find the "most distinctive" words in documents. In this homework, we will compute TF-IDF scores for a set of documents and use that to determine the most distinctive word in each document. You can refer to the class notesfor help with this part of the homework.

NLTK

NLTK is the Natural Language Toolkit, a set of common text-processing tools for Python. You can install NLTK using:

```
> python3 -mpip install --user nltk
```

> Note: if you are using Jupyter Notebook on Scholar to do your assignments, you are going to want to run this command from a terminal window on Scholar (by SSHing to scholar, or opening a terminal window through Thinlinc). Some students have been having trouble trying to install modules using Jupyter Notebook's built-in terminal.

We will use NLTK to clean and *stem* the documents before processing the documents. To clean the documents, we will:

1. Remove *stop words* from each document:
2. Remove punctuation from the document (you may use the `remove_punc` helper method in `helper.py` to help with this)
3. Make the words lower case
4. Stem the words

There are examples of steps 1 and 3 in the notebook.

Instructions

0) Set up your repository for this homework.

Use the link on Piazza to set up Homework 9.

The repository should contain several files:

1. This README
2. Two starter files with some function stubs called `hw9_1.py` and `hw9_2.py`
3. A helper file called `helper.py` (this contains code to remove punctuation from a string)
4. 7 translations of the UN Declaration on Human Rights in different languages, in the subdirectory `ngrams/` : `english.txt` , `french.txt` , `german.txt` , `italian.txt` , `portuguese.txt` , `spanish.txt` , `swahili.txt` .
5. A "mystery" file, `mystery.txt` where you are supposed to detect the language it is in, in the subdirectory `ngrams/`
6. A directory, `1ecs/` that contains 14 text files. These are the documents you will process for Problem 4

1) Homework Problem 1: Identifying a Language from n-gram Distributions

Step 1:

First fill in the functions `getFormattedText` , `getNgrams` , `getDict` , and `topNCommon` as described in `hw9_1.py` to read an input file into a list of its lines, then process those lines to construct a *dictionary* of n-grams, and finally output a list of tuples containing the N largest (n-gram, count) pairs in decreasing order. For example:

```
> topNCommon('ngrams/english.txt',10)
> [('the', 149), (' th', 142), (' an', 129), ('he ', 121), ('nd ', 113), ('and', 111), ('ion', 102), (' of', 93),
('of ', 89), ('tio', 88)]
```

It is recommended you review material on dictionaries, and when needed, sets. Class notes can be found here.

The dictionaries you will create will map an n-gram (the dictionary key) to the number of times that the n-gram appears (the dictionary value). Pay careful attention to the processing we want you to do on each line (pad it out with `_` s, make all the text lowercase) and *how* we want you to do the processing.

When writing `topNCommon` , you may find the following StackOverflow post helpful: https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-value. You may also find the dictionary method popitem() to be useful.

There will be a line of comment symbols in `hw9_1.py` , and once you have filled in the functions up to this point, you may run hw9_1.py to test that what you have so far works. (It should output the above output for `topNCommon` when you run it, followed by potentially and empty list and empty string that will change as you fill out the functions in the later steps).

Step 2:

Next, fill in the code for `getAllDicts` , `dictUnion` , and `getAllNGrams` . `getAllDicts` takes in a list of filepath strings and returns a list of the n-gram dictionaries for all the files. `dictUnion` takes in a list of dictionaries of n-grams for a group of files and creates one large alphabetically **sorted list** of all the n-grams across all the input dictionaries. Note that this larger output list doesn't involve the counts, it only involves the n-grams. It is highly recommended that you look into the "set" data type and methods such as `union` or `update` , as well as the dictionary method `keys` . `getAllNGrams` takes in a list of filepaths and returns one list of all n-grams across all the files. Big hint: consider the functions you have written in step 2 already.

You may test your code at this point to see if you get a large list of alphabetized n-grams output in addition to the output from step 1. This list is omitted to not overly clutter this readme, however, the beginning of it should look something like:

[' a ', ' ab', ' ac', ' ad', ' af', ' ag', ' ai', ' aj', ' ak', ' al', ' am', ' an', ' ao', ' ap', ' aq', ' ar', ' as', ' at', ' au', ' av', ' ay', ' az', ' ba', ' be', ' bi', ' bo', ' br', ' bu', ' by', ' ca', ' ce', ' ch', ' ci', ' cl', ' co', ' cr', ' cu', ' da', ' de', ' di', ' do', ' dr', ' du', ' e ', ' ea', ' ec', ' ed', ' ef', ' eg', ' eh', ' ei', ' ej', ' el', ' em', ' en', ' ep', ' eq', ' er', ' es', ' et', ' eu', ' ev', ' ex', ' fa', ' fe', ' fi', ' fo', ' fr', ' fu', ' ga', ' ge', ' gi', ' gl', ' go', ' gr', ' gu', ' ha', ' he', ' hi', ' ho', ' hu', ' i ', ' id', ' if', ' ig', ' ih', ' il', ' im', ' in', ' ir', ' is', ' it', ' je', ' jo', ' ju', ' ka', ' ke', ...

Step 3:

Finally, fill in the code for `compareLang.` `compareLang` takes in a file that you want to determine the language of, also a group of files for different languages as a basis of comparison, and finally, a value N that tells how many of the top n-grams must be compared between the mystery file and each language file. What it should do is find the intersection of the top N n-grams between the mystery file and each language file, and determine which language file has the largest intersection. That language file's filepath should be the output.

Some notes:

- Once again, consider using the "set" data type and, for this situation, its method `intersection()`.
- The line: `mystTopNGrams = set([tup[0] for tup in topNCommon(testFile,N)])` should give a starting point on how to construct a set of just the top N n-grams and discarding the count values for the mystery language file. You'll have to also do something similar for the language files to compare them.
- The list method index() may be useful.

Running `hw9_1.py` now should output something for steps 1,2 and 3.

Submit the filled-in version of `hw9_1.py`

2) Homework Problem 2: TF-IDF

For this problem, we will compute the tf-idf scores for all the terms in each document in the `lecs/` folder.

Fill in the missing functions in `hw9_2.py`, according to their specifications. You may find the lecture notes linked above helpful for thinking about the format of the doc-word matrix, and the notebook linked above helpful for thinking about how to construct a doc-word matrix.

Note that even after installing nltk and importing it, you may need to add the following below your nltk import statement:

```
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
```

While we will test you using our own tests, at the end of `hw9_2.py` are some good test cases to check your code as you go. Feel free to uncomment and recomment them where convenient for you as you write the different functions. The output to those should be:

```
*** Testing readAndCleanDoc ***
['let', "'s", 'look', 'wifi', "'s"]
*** Testing buildDocWordMatrix ***
(2, 429)
429
[2. 9. 2. 1. 1. 1. 3. 2. 1. 4.]
["'re", "'s", "'ve", '.11', '1', '1.2', '100', '11', '1997', '1999']
[ 6. 11.  0.  0.  0.  0.  0.  0.  0.  0.]
*** Testing buildTFMatrix ***
[0.00305344 0.01374046 0.00305344 0.00152672 0.00152672 0.00152672
 0.00458015 0.00305344 0.00152672 0.00610687]
[0.01438849 0.0263789  0.         0.         0.         0.
 0.         0.         0.         0.        ]
[1. 1.]
```

```
*** Testing buildIDFMatrix ***
[0.      0.      0.30103 0.30103 0.30103 0.30103 0.30103 0.30103 0.30103
 0.30103]
*** Testing buildTFIDFMatrix ***
(2, 429)
[0.         0.         0.00091918 0.00045959 0.00045959 0.00045959
 0.00137876 0.00091918 0.00045959 0.00183835]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
*** Testing findDistinctiveWords ***
{'lecs/1_vidText.txt': array(['second', 'per', 'megabit'], dtype='<U12'), 'lecs/2_vidText.txt': array(['point',
'router', 'set'], dtype='<U12')}
```

Submit your filled in version of `hw9_2.py`

What you need to submit

Each of the homework problems specify what file(s) to generate and submit for that problem.

Submitting your code

Push your completed `hw9_1.py` and `hw9_2.py` to your repository before the deadline.