Homework 8

Due: Nov. 7, 2025 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 * 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 (the 3-grams in "words in a sentence" are "words in a" and "in a sentence") or over the characters in a sentence (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 `_` (underscores) 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" of English and Spanish have different distributions). This homework will build n-gram distributions for texts in different languages for various values of n, and you will find the pair of languages that are the most similar by analyzing their n-gram distributions. You will also see how the similarity value changes as we increase n.

TF-IDF

We also discussed TF-IDF as a metric to find the "most distinctive" words in documents. However, TF-IDF can be used to analyze other parts of documents as well! In this homework, we will compute TF-IDF scores for a set of documents and use that to determine the most distinctive **n-gram** in each document. You can refer to the class notes on Brightspace for help with this part of the homework. You may use functions written in Problem 1 to complete this part.

NLTK

NLTK is the Natural Language Toolkit, consisting of 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 use NLTK to clean the documents before processing the documents. To clean the documents, we:

1) Remove *stop words* (words such as "the", "is", "and", etc.) 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) Remove all spaces between the words.

This part has been provided in `read_and_clean_doc` function of `hw8_2.py` that you will need later, so you don't need to implement it by yourself.

Instructions

0) Set up your repository for this homework.

The repository should contain several files:

1. This README.
2. Two starter files with some function stubs, `hw8_1.py` and `hw8_2.py`.
3. A helper file `helper.py` (this contains code to remove punctuation from a string).
4. 8 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`, `mystery.txt`.
5. A directory, `lecs/` that contains 14 text files. These are the documents you will process for Problem 2.

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

Step 1:

We have filled the functions `get_formatted_text` and `get_ngrams` (to read an input file into a list of its lines, then process those lines to construct a *list* of n-grams). You are responsible for filling in the functions `get_dict` and `top_N_common` in `hw8_1.py` to finally output a dictionary containing the N most frequent n-grams as keys and their count (frequency) as their corresponding values, in **decreasing order**.

In addition to the `n` argument, which specifies the length of each n-gram, the function `top_N_common` also takes an optional `threshold` argument. This allows you to filter out any n-grams that appear less than `threshold` times in the input. Only n-grams that appear at least (greater or equal) `threshold` times will be considered in the final output. By default, the `threshold` is set to 0, meaning all n-grams are included unless specified otherwise.

For example, the output of the following `top_N_common` function call should return the following dictionary (**Note**: we make the default value of threshold to 0, so if you use the function directly, it will not filter anything):

```
>>> top_N_common('ngrams/english.txt', 20, 4)
{' the': 126, 'the ': 119, 'and ': 106, ' and': 106, ' of ': 89, 'tion': 88, ' to ': 84, 'at
```

```
>>> top_N_common('ngrams/english.txt', 20, 4, threshold=50)
{' the': 126, 'the ': 119, ' and': 106, 'and ': 106, ' of ': 89, 'tion': 88, ' to ': 84, 'at
```

Note that, if the values are identical, as observed in the final three entries of the displayed dictionary, then sort by the **key** in **descending order**, as shown in the above output.

When writing `top_N_common`, 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. Nonetheless, these are only suggestions and not the only way to approach this problem.

Note:

Unless specified, the threshold paramter should remain as default value 0 if the function needs to be used in other problems.

Step 2:

We have filled `get_all_dicts` and `get_all_ngrams`.

`get_all_dicts` takes in a list of filepath strings and returns a list of the n-gram dictionaries for all the files. `get_all_ngrams` takes in a list of filepaths and

returns one list of all n-grams across all the files.

You are responsible for filling in the code for `dict_union`.

`dict_union` 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`.

You may test your code at this point to see if you get a list of alphabetized n-grams output in addition to the output from step 1. It should look something like the following, for n=4 (You can get this output by uncommenting `print(get_all_ngrams(path_list,4)[:50])` in `if __name__ == '__main__':`):

```
[' a a', ' a b', ' a c', ' a d', ' a e', ' a f', ' a g', ' a h', ' a i', ' a l', ' a m', ' a
```

Step 3:

Finally, fill in the code for `compare_langs`. `compare_langs` takes in a file that you want to determine the language of, a group of files of different languages for comparison, and finally, a value N that indicates 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: `set([key for key in top_N_common(test_file,N,n)])` (**Note that we do not use any filtering this case, which means the threshold is 0**) 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.

After uncommenting all parts of the code in the main function and Running `hw8_1.py`, you should now see outputs from steps 1, 2 and 3.

Submit the filled-in version of `hw8_1.py`

2) Homework Problem 2: TF-IDF

For this problem, we will compute the tf-idf scores for the set of n-grams in each document in the `lecs/` folder.

Fill in the missing functions in `hw8_2.py`, according to their specifications and requirements. You can use any functions written `hw8_1.py` to help you in this problem. You may find the lecture notes about n-grams on Brightspace helpful for thinking about the format of the doc-word matrix, and referring to the notebooks shown in class may be helpful for thinking about how to construct a doc-word matrix.

Step 1:

We have filled in the code for `read_and_clean_doc`. This function takes in a file path and outputs a cleaned string generated from the text in the file at the filepath. This function reads the file into a single string, removes punctuation (see `helper.py`), removes stopwords, and removes all whitespace from the string such that it is a sequence of only characters. This function returns the cleaned string.

You should fill in the code for `build_doc_word_matrix`. This function takes in a list of file paths and a number `n` equal to the number of characters in an n-gram. You should generate a document-word matrix with columns representing n-grams and rows representing each document in the list given. You may use functions previously written in `hw8_1.py` to complete this part if you wish.

Additionally, the `build_doc_word_matrix` function should include an option to **normalize** the document-word matrix. This option, enabled by setting the `normalize` parameter to `True`, modifies the matrix so that the values represent the frequency of each n-gram relative to the total number of n-grams in each document.

Normalization Formula:

For each document (d), the relative frequency of each n-gram (t) is calculated as:

```
\text{normalized\_freq}(t, d) = \frac{\text{count}(t, d)}{\sum_{t'} \text{count}(t', d)}
```

Where: - `count(t, d)` is the frequency of n-gram `t` in document `d`, - The summation of `count(t, d)` is the total number of n-grams in document (d).

This normalization ensures that the frequency values for each document sum up to 1, making it easier to compare n-gram distributions across documents of different lengths.

Note that **you should round the normalized values to 4 decimal places.**

Example:

Let's say we have the following document-word matrix (before normalization) for 3 documents and 4 unique n-grams (`"abc"`, `"bcd"`, `"cde"`, `"def"`):

Document	abc	bcd	cde	def
Doc 1	2	3	0	1
Doc 2	0	1	2	1

If the `normalize` parameter is set to `True` (the default value is False), we normalize each row by dividing the count of each n-gram by the total number of n-grams in that document:

For **Doc 1**: - Total n-grams: (2 + 3 + 0 + 1 = 6) - Normalized frequencies: 2/6, 3/6, 0/6, 1/6 = 0.33, 0.5, 0, 0.17

For **Doc 2**: - Total n-grams: (0 + 1 + 2 + 1 = 4) - Normalized frequencies: 0/4, 1/4, 2/4, 1/4 = 0, 0.25, 0.5, 0.25

The normalized document-word matrix would look like this:

Document	abc	bcd	cde	def
Doc 1	0.3333	0.5	0	0.17
Doc 2	0	0.25	0.5	0.25

Output

If you run `hw8_2.py` after completing step 1, you should get the following output.

```
>>> build_doc_word_matrix(doclist, 4)
*** Testing build_doc_word_matrix ***
(2, 3413)
3413
[ 1.  1.  2.  2.  1. 16.  0.  0.  1.  1.]
['0038', '0098', '00mb', '00me', '0138', '0211', '02co', '02fa', '02pa', '0380']
[0. 0. 0. 0. 0. 2. 1. 1. 2. 0.]

>>> build_doc_word_matrix(doclist, 4, normalize=True)
*** Testing build_doc_word_matrix normalization ***
(2, 3413)
3413
[0.0002 0.0002 0.0005 0.0005 0.0002 0.004  0.     0.     0.0002 0.0002]
['0038', '0098', '00mb', '00me', '0138', '0211', '02co', '02fa', '02pa', '0380']
[0.     0.     0.     0.     0.     0.0008 0.0004 0.0004 0.0008 0.    ]
```

Note:

Unless specified, the `normalize` paramter will remain as default value False if the function needs to be used in other problems.

Step 2:

1. Fill in the code for `build_tf_matrix` and `build_idf_matrix`. In addition to the instructions given in `hw8_2.py`, you may want to refer to the lecture notes on brightspace for how to complete these functions.
2. Fill in the code for `build_tfidf_matrix`.

After completing these 3 function you can uncomment the subsequent code block in the main fuction to test them out. The output corresponding to this code block should be as follows:

```
*** Testing build_tf_matrix ***
[0.00024795 0.00024795 0.00049591 0.00049591 0.00024795 0.00396727
 0.         0.         0.00024795 0.00024795]
[0.         0.         0.         0.         0.         0.00078401
 0.000392   0.000392   0.00078401 0.        ]
[1. 1.]

*** Testing build_idf_matrix ***
[0.30103 0.30103 0.30103 0.30103 0.30103 0.      0.30103 0.30103 0.
 0.30103]

*** Testing build_tfidf_matrix ***
(2, 3413)
[7.46417049e-05 7.46417049e-05 1.49283410e-04 1.49283410e-04
 7.46417049e-05 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 7.46417049e-05]
[0.       0.       0.       0.       0.       0.       0.000118 0.000118
 0.       0.      ]
```

Step 3:

Fill in the code for `find_distinctive_ngrams`. This function calculates and outputs the top 3 most unique words found in each document (Note: the most unique words, not the most common). This function takes in a doc-word matrix, a list of ngrams, and a list of filepaths. You should return a dictionary with the following qualities: each filepath becomes a key in the dictionary. The value for each filepath key should be a list of the top 3 most unique ngrams in the file.

You can uncomment the subsequent code block to test the `find_distinctive_ngrams` function and expect the following output.

```
*** Testing find_distinctive_words ***
```

```
{'lecs/1_vidText.txt': ['econ', 'seco', 'rsec'], 'lecs/2_vidText.txt': ['uter', 'sspo', 'out
```

Feel free to uncomment and recomment the code blocks in the main function as you complete the different functions.

Note: You may get an output like `{'lecs\\1_vidText.txt': ['econ', 'seco', 'rsec'], 'lecs\\2_vidText.txt': ['uter', 'sspo', 'oute']}` when testing `find_distinctive_words`. **This is also accepted**. Different operating systems have different filesystem naming conventions, hence the `\\` vs `/` difference.

Submit your completed `hw8_2.py` file.

Please refer to the slides fo better understanding of the problem. # What you need to submit

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

Submitting your code

Separately submit `hw8_1.py` and `hw8_2.py` on Gradescope.