Lab 4: Investigating the Performance Advantages of NumPy Arrays

At a high level, the purpose of this lab is to test how much more efficient bulk, vectorized operations can by (using NumPy arrays) than standard, dictionary-oriented Python computations.

In this lab, your task is to compare three different solutions to the *co-occurrence problem*. This is a very fundamental computation in text analytics. In a nutshell, we have *n* documents. We have a dictionary with some number of words; we know how many times each document has each word in the dictionary. For each of the possible (word, word) pairs, we want to compute the number of documents (between zero and *n*) that have that particular (word, word) pair.

For example, let's say our documents are:

doc 1: {word1, word2, word4, word5}
doc 2: {word1, word2, word5}
doc 3: {word2, word3, word5}

Then the result of the co-occurrence computation is:

{word1, word1}: 2 co-occurs (means word1 occurs twice in the corpus)
{word1, word2}: 2 co-occurs
{word1, word4}: 1 co-occurs
{word1, word5}: 2 co-occurs
{word2, word2}: 3 co-occurs
{word2, word3}: 1 co-occurs
{word2, word4}: 1 co-occurs
{word2, word5}: 3 co-occurs
{word3, word3}: 1 co-occurs
{word3, word5}: 1 co-occurs
{word4, word4}: 1 co-occurs
{word4, word5}: 1 co-occurs
{word5, word5}: 1 co-occurs

Given that, tere are the lab's three subtasks:

Subtask One

First, run the pure, dictionary-based LDA implementation provided (this is the first one) to build a document corpus (just like in the last lab). This will build the `wordsInCorpus` object, which is a Python dictionary. The dictionary's key is a document identifier, and the value is another dictionary. For the dictionary associated with a particular document identifier, the key is a word identifier, and the value is the number of occurrences of the word in the document.

Given this setup, write and execute the following code:

```
start = time.time()

# coOccurrences will be a map where the key is a
# (wordOne, wordTwo) pair, and the value is the number of times
# those two words co-occurred in a document, so this will be a
# value between zero and 50
coOccurrences = {}

# now, have a nested loop that fils up coOccurrences
# YOUR CODE HERE

end = time.time()
end - start
```

In case you don't remember, the way to loop through all of the keys in a Python dictionary is something like the following:

```
for wordOne in wordsInCorpus[doc]:
      # code here
```

Subtask Two

Now, run the NumPy array-based LDA implementation (again, see OwlSpace). This will build the `wordsInCorpus` array. It stores the same data, but in a NumPy array. `wordsInCorpus[34, 355]` is the number of times that the 355th word in the dictionary occurred in the 34th document.

Given this, write a NumPy code that loops through all of the documents. Treat each document as a vector whose length is the length of the corpus. Then you can use the outer product of the vector giving the word counts for that document with itself to create a matrix of co-occurrences for that document (if you don't remember what an outer product is, do a quick review on Wikipedia). Summing the 50 matrices gives the answer. Hint: in order to take a NumPy array `foo` and "clip" all of its entries so that none of them is greater than one, use the code `np.clip (foo, 0, 1)`. And `np.outer (foo, bar)` computes the outer product of two vectors/matrices. And Here is the skeleton:

```
start = time.time()

# coOccurrences[i, j] will give the count of the number of times that
# word i and word j appear in the same document in the corpus
coOccurrences = np.zeros ((2000, 2000))

# now, have a nested loop that fils up coOccurrences
```

```
# YOUR CODE HERE

end = time.time()
end - start
```

Subtask Three

Now, write a code that uses a single matrix multiply to create coOccurrences. Note that you can use the `transpose` operation to transpose a matrix, and `np.dot ()` will multiply two matrices.

```
start = time.time()

# now, create coOccurrences via a matrix multiply
# YOUR CODE HERE

end = time.time()
end - start
```