Containment_Exercise

October 21, 2020

0.1 Calculating Containment

In this notebook, you'll implement a containment function that looks at a source and answer text and returns a *normalized* value that represents the similarity between those two texts based on their n-gram intersection.

```
In [24]: import numpy as np
         import sklearn
```

0.1.1 N-gram counts

One of the first things you'll need to do is to count up the occurrences of n-grams in your text data. To convert a set of text data into a matrix of counts, you can use a CountVectorizer.

Below, you can set a value for n and use a CountVectorizer is used to count up the n-gram occurrences. In the next cell, we'll see that the CountVectorizer constructs a vocabulary, and later, we'll look at the matrix of counts.

```
In [31]: from sklearn.feature_extraction.text import CountVectorizer

         a_text = "This is an answer text."
         s_text = "This is a source text."

         # set n
         n = 1

         # instantiate an ngram counter
         counts = CountVectorizer(analyzer='word', ngram_range=(n,n))

         # create a dictionary of n-grams by calling `.fit`
         vocab2int = counts.fit([a_text, s_text]).vocabulary_

         # print dictionary of words:index
         print(vocab2int)
```

```
{'this': 5, 'is': 2, 'an': 0, 'answer': 1, 'text': 4, 'source': 3}
```

0.1.2 EXERCISE: Create a vocabulary for 2-grams (aka "bigrams")

Create a `CountVectorizer`, `counts_2grams`, and fit it to our text data. Print out the resultant vocabulary.

```
In [32]: '''a_text = "This is an answer text"
         s_text = "This is a source text"
         '''
         n = 2

         # instantiate an ngram counter
         counts_2grams = CountVectorizer(analyzer='word', ngram_range=(n,n))

         # create a vocabulary for 2-grams
         counts_2grams = counts_2grams.fit([a_text, s_text]).vocabulary_
         print(counts_2grams)
```

```
{'this is': 5, 'is an': 2, 'an answer': 0, 'answer text': 1, 'is source': 3, 'source text': 4}
```

0.1.3 What makes up a word?

You'll note that the word "a" does not appear in the vocabulary. And also that the words have been converted to lowercase. When `CountVectorizer` is passed `analyzer='word'` it defines a word as *two or more* characters and so it ignores uni-character words. In a lot of text analysis, single characters are often irrelevant to the meaning of a passage, so leaving them out of a vocabulary is often desired behavior.

For our purposes, this default behavior will work well; we don't need uni-character words to determine cases of plagiarism, but you may still want to experiment with uni-character counts.

> If you *do* want to include single characters as words, you can choose to do so by adding one more argument when creating the `CountVectorizer`; pass in the definition of a token, `token_pattern = r"(?u)\b\w+\b"`.

This regular expression defines a word as one or more characters. If you want to learn more about this vectorizer, I suggest reading through the source code, which is well documented.

Next, let's fit our `CountVectorizer` to all of our text data to make an array of n-gram counts!

The below code, assumes that `counts` is our `CountVectorizer` for the n-gram size we are interested in.

```
In [33]: '''a_text = "This is an answer text. hello my friend"
         s_text = "This is a source text. hello how is my friend doing?"
         {'an': 0, 'answer': 1, 'is': 2, 'source': 3, 'text': 4, 'this': 5}
         '''
         # create array of n-gram counts for the answer and source text
         ngrams = counts.fit_transform([a_text, s_text])

         # row = the 2 texts and column = indexed vocab terms (as mapped above)
         # ex. column 0 = 'an', col 1 = 'answer'.. col 4 = 'text'
         ngram_array = ngrams.toarray()
         print(ngram_array)
```

```
[[1 1 1 0 1 1]
 [0 0 1 1 1 1]]
```

So, the top row indicates the n-gram counts for the answer text `a_text`, and the second row indicates those for the source text `s_text`. If they have n-grams in common, you can see this by looking at the column values. For example they both have one "is" (column 2) and "text" (column 4) and "this" (column 5).

```
[[1 1 1 0 1 1]    =   an  answer  [is]  ______  [text] [this]
 [0 0 1 1 1 1]]   =   __  ______  [is]  source  [text] [this]
```

0.1.4 EXERCISE: Calculate containment values

Assume your function takes in an `ngram_array` just like that generated above, for an answer text (row 0) and a source text (row 1). Using just this information, calculate the containment between the two texts. As before, it's okay to ignore the uni-character words.

To calculate the containment: 1. Calculate the n-gram **intersection** between the answer and source text. 2. Add up the number of common terms. 3. Normalize by dividing the value in step 2 by the number of n-grams in the answer text.

The complete equation is:

$$\frac{\sum count(\text{ngram}_A) \cap count(\text{ngram}_S)}{\sum count(\text{ngram}_A)}$$

```
In [34]: def containment(ngram_array):
             ''' Containment is a measure of text similarity. It is the normalized,
                intersection of ngram word counts in two texts.
                :param ngram_array: an array of ngram counts for an answer and source text.
                :return: a normalized containment value.'''

             # your code here
             intersection_ngram_sum = np.sum(ngram_array[0] * ngram_array[1])
             return intersection_ngram_sum/np.sum(ngram_array[0])
             pass
         #print((ngram_array[0], ngram_array[1]))
         #print(np.sum(ngram_array[0] * ngram_array[1]))

In [35]: # test out your code
         containment_val = containment(ngrams.toarray())

         print('Containment: ', containment_val)

         # note that for the given texts, and n = 1
         # the containment value should be 3/5 or 0.6
         assert containment_val==0.6, 'Unexpected containment value for n=1.'
         print('Test passed!')
```

```
Containment:  0.6
Test passed!
```

```
In [36]: # test for n = 2
         counts_2grams = CountVectorizer(analyzer='word', ngram_range=(2,2))
         bigram_counts = counts_2grams.fit_transform([a_text, s_text])

         # calculate containment
         containment_val = containment(bigram_counts.toarray())

         print('Containment for n=2 : ', containment_val)

         # the containment value should be 1/4 or 0.25
         assert containment_val==0.25, 'Unexpected containment value for n=2.'
         print('Test passed!')
```

```
Containment for n=2 :  0.25
Test passed!
```

I recommend trying out different phrases, and different values of n. What happens if you count for uni-character words? What if you make the sentences much larger?

I find that the best way to understand a new concept is to think about how it might be applied in a variety of different ways.