day_four

May 11, 2016

1 Text Data

1.1 Pre-introduction

We'll be spending a lot of time today manipulating text. Make sure you remember how to split, join, and search strings.

1.2 Introduction

We've spent a lot of time in python dealing with text data, and that's because text data is everywhere. It is the primary form of communication between persons and persons, persons and computers, and computers and computers. The kind of inferential methods that we apply to text data, however, are different from those applied to tabular data.

This is partly because documents are typically specified in a way that expresses both structure and content using text (i.e. the document object model).

Largely, however, it's because text is difficult to turn into numbers in a way that preserves the information in the document. Today, we'll talk about dominant language model in NLP and the basics of how to implement it in Python.

1.2.1 The term-document model

This is also sometimes referred to as "bag-of-words" by those who don't think very highly of it. The term document model looks at language as individual communicative efforts that contain one or more tokens. The kind and number of the tokens in a document tells you something about what is attempting to be communicated, and the order of those tokens is ignored.

To start with, let's load a document.

```
In [1]: import nltk
        #nltk.download('webtext')
        document = nltk.corpus.webtext.open('grail.txt').read()
```

```
/Users/dillon/anaconda/lib/python3.5/site-packages/sklearn/utils/fixes.py:64: DeprecationWarning: inspe
  if 'order' in inspect.getargspec(np.copy)[0]:
```

Let's see what's in this document

```
In [2]: len(document.split('\n'))
```

```
Out[2]: 1192
```

```
In [3]: document.split('\n')[0:10]
```

```
Out[3]: ['SCENE 1: [wind] [clop clop clop] ',
         'KING ARTHUR: Whoa there!  [clop clop clop] ',
         'SOLDIER #1: Halt!  Who goes there?',
```

```
'ARTHUR: It is I, Arthur, son of Uther Pendragon, from the castle of Camelot.  King of the Brit
'SOLDIER #1: Pull the other one!',
'ARTHUR: I am, ...  and this is my trusty servant Patsy.  We have ridden the length and breadth
'SOLDIER #1: What?  Ridden on a horse?',
'ARTHUR: Yes!',
"SOLDIER #1: You're using coconuts!",
'ARTHUR: What?']
```

It looks like we've gotten ourselves a bit of the script from Monty Python and the Holy Grail. Note that when we are looking at the text, part of the structure of the document is written in tokens. For example, stage directions have been placed in brackets, and the names of the person speaking are in all caps.

1.3 Regular expressions

If we wanted to read out all of the stage directions for analysis, or just King Arthur's lines, doing so in base python string processing will be very difficult. Instead, we are going to use regular expressions. Regular expressions are a method for string manipulation that match patterns instead of bytes.

```
In [4]: import re
        snippet = "I fart in your general direction! Your mother was a hamster, and your father smelt of
        re.search(r'mother', snippet)
```

```
Out[4]: <_sre.SRE_Match object; span=(39, 45), match='mother'>
```

Just like with `str.find`, we can search for plain text. But `re` also gives us the option for searching for patterns of bytes - like only alphabetic characters.

```
In [5]: re.search(r'[a-z]', snippet)
```

```
Out[5]: <_sre.SRE_Match object; span=(2, 3), match='f'>
```

In this case, we've told re to search for the first sequence of bytes that is only composed of lowercase letters between `a` and `z`. We could get the letters at the end of each sentence by including a bang at the end of the pattern.

```
In [6]: re.search(r'[a-z]!', snippet)
```

```
Out[6]: <_sre.SRE_Match object; span=(31, 33), match='n!'>
```

If we wanted to pull out just the stage directions from the screenplay, we might try a pattern like this:

```
In [7]: re.findall(r'[a-zA-Z]', document)[0:10]
```

```
Out[7]: ['S', 'C', 'E', 'N', 'E', 'w', 'i', 'n', 'd', 'c']
```

So that's obviously no good. There are two things happening here:

1. `[` and `]` do not mean 'bracket'; they are special characters which mean 'any thing of this class'
2. we've only matched one letter each

A better regular expression, then, would wrap this in escaped brackets, and include a command saying more than one letter.

Re is flexible about how you specify numbers - you can match none, some, a range, or all repetitions of a sequence or character class.

character	meaning
{x}	exactly x repetitions
{x,y}	between x and y repetitions
?	0 or 1 repetition
*	0 or many repetitions
+	1 or many repetitions

```
In [8]: re.findall(r'\[[a-zA-Z]+\]', document)[0:10]

Out[8]: ['[wind]',
         '[thud]',
         '[clang]',
         '[clang]',
         '[clang]',
         '[clang]',
         '[clang]',
         '[clang]',
         '[clang]',
         '[clang]']
```

This is better, but it's missing that `[clop clop clop]` we saw above. This is because we told the regex engine to match any alphabetic character, but we did not specify whitespaces, commas, etc. to match these, we'll use the dot operator, which will match anything expect a newline.

Part of the power of regular expressions are their special characters. Common ones that you'll see are:

character	meaning
`.`	match anything except a newline
`^`	match the start of a line
`$`	match the end of a line
`\s`	matches any whitespace or newline

Finally, we need to fix this `+` character. It is a 'greedy' operator, which means it will match as much of the string as possible. To see why this is a problem, try:

```
In [9]: snippet = 'This is [cough cough] and example of a [really] greedy operator'
        re.findall(r'\[.+\]', snippet)

Out[9]: ['[cough cough] and example of a [really]']
```

Since the operator is greedy, it is matching everything inbetween the first open and the last close bracket. To make `+` consume the least possible amount of string, we'll add a `?`.

```
In [10]: p = re.compile(r'\[.+?\]')
         re.findall(p, document)[0:10]

Out[10]: ['[wind]',
          '[clop clop clop]',
          '[clop clop clop]',
          '[clop clop clop]',
          '[thud]',
          '[clang]',
          '[clang]',
          '[clang]',
          '[clang]',
          '[clang]']
```

What if we wanted to grab all of Arthur's speech? This one is a little trickier, since:

1. It is not conveniently bracketed; and,
2. We want to match on ARTHUR, but not to capture it

If we wanted to do this using base string manipulation, we would need to do something like:

```
split the document into lines
create a new list of just lines that start with ARTHUR
create a newer list with ARTHUR removed from the front of each element
```

Regex gives us a way of doing this in one line, by using something called groups. Groups are pieces of a pattern that can be ignored, negated, or given names for later retrieval.

character	meaning
(x)	match x
(?:x)	match x but don't capture it
(?P<x>)	match something and give it name x
(?=x)	match only if string is followed by x
(?!x)	match only if string is not followed by x

```
In [11]: p = re.compile(r'(?:ARTHUR: )(.+)')
         re.findall(p, document)[0:10]

Out[11]: ['Whoa there!  [clop clop clop] ',
          'It is I, Arthur, son of Uther Pendragon, from the castle of Camelot.  King of the Britons, d
          'I am, ...  and this is my trusty servant Patsy.  We have ridden the length and breadth of th
          'Yes!',
          'What?',
          'So?  We have ridden since the snows of winter covered this land, through the kingdom of Merc
          'We found them.',
          'What do you mean?',
          'The swallow may fly south with the sun or the house martin or the plover may seek warmer cli
          'Not at all.  They could be carried.']
```

Because we are using `findall`, the regex engine is capturing and returning the normal groups, but not the non-capturing group. For complicated, multi-piece regular expressions, you may need to pull groups out separately. You can do this with names.

```
In [12]: p = re.compile(r'(?P<name>[A-Z ]+)(?::)(?P<line>.+)')
         match = re.search(p, document)
         match

Out[12]: <_sre.SRE_Match object; span=(34, 77), match='KING ARTHUR: Whoa there!  [clop clop clop] '>

In [13]: match.group('name'), match.group('line')

Out[13]: ('KING ARTHUR', ' Whoa there!  [clop clop clop] ')
```

Now let's try a small challenge! To check that you've understood something about regular expressions, we're going to have you do a small test challenge. Partner up with the person next to you - we're going to do this as a pair coding exercise - and choose which computer you are going to use.

Then, navigate to `challenges/04_text/` and read through challenge A. When you think you've completed it successfully, run `py.test test_A.py` .

1.4 Tokenizing

Let's grab Arthur's speech from above, and see what we can learn about Arthur from it.

```
In [14]: p = re.compile(r'(?:ARTHUR: )(.+)')
         arthur = ' '.join(re.findall(p, document))
         arthur[0:100]
```

```
Out[14]: 'Whoa there!  [clop clop clop]  It is I, Arthur, son of Uther Pendragon, from the castle of Ca
```

In our model for natural language, we're interested in words. The document is currently a continuous string of bytes, which isn't ideal. You might be tempted to separate this into words using your newfound regex knowledge:

```
In [15]: p = re.compile(r'\w+', flags=re.I)
         re.findall(p, arthur)[0:10]

Out[15]: ['Whoa', 'there', 'clop', 'clop', 'clop', 'It', 'is', 'I', 'Arthur', 'son']
```

But this is problematic for languages that make extensive use of punctuation. For example, see what happens with:

```
In [16]: re.findall(p, "It isn't Dav's cheesecake that I'm worried about")

Out[16]: ['It',
          'isn',
          't',
          'Dav',
          's',
          'cheesecake',
          'that',
          'I',
          'm',
          'worried',
          'about']
```

The practice of pulling apart a continuous string into units is called "tokenizing", and it creates "tokens". NLTK, the canonical library for NLP in Python, has a couple of implementations for tokenizing a string into words.

```
In [17]: from nltk import word_tokenize
         word_tokenize("It isn't Dav's cheesecake that I'm worried about")

Out[17]: ['It',
          'is',
          "n't",
          'Dav',
          "'s",
          'cheesecake',
          'that',
          'I',
          "'m",
          'worried',
          'about']
```

The distinction here is subtle, but look at what happened to "isn't". It's been separated into "IS" and "N'T", which is more in keeping with the way contractions work in English.

```
In [18]: tokens = word_tokenize(arthur)
         tokens[0:10]

Out[18]: ['Whoa', 'there', '!', '[', 'clop', 'clop', 'clop', ']', 'It', 'is']
```

At this point, we can start asking questions like what are the most common words, and what words tend to occur together.

```
In [19]: len(tokens), len(set(tokens))

Out[19]: (2393, 596)
```

So we can see right away that Arthur is using the same words a whole bunch - on average, each unique word is used four times. This is typical of natural language.

> Not necessarily the value, but that the number of unique words in any corpus increases much more slowly than the total number of words.
>
> A corpus with 100M tokens, for example, probably only has 100,000 unique tokens in it.

For more complicated metrics, it's easier to use NLTK's classes and methods.

```
In [20]: from nltk import collocations
         fd = collocations.FreqDist(tokens)
         fd.most_common()[:10]

Out[20]: [(',', 135),
          ('.', 129),
          ('!', 119),
          ('the', 70),
          ('?', 61),
          ('you', 51),
          ('of', 45),
          (']', 38),
          ('[', 38),
          ('I', 34)]

In [21]: measures = collocations.BigramAssocMeasures()
         c = collocations.BigramCollocationFinder.from_words(tokens)
         c.nbest(measures.pmi, 10)

Out[21]: [("'Til", 'Recently'),
          ('ARTHUR', 'chops'),
          ('An', 'African'),
          ('BLACK', 'KNIGHT'),
          ('Bloody', 'peasant'),
          ('Castle', 'Aaagh'),
          ('Chop', 'his'),
          ('Cut', 'down'),
          ('Divine', 'Providence'),
          ('Eternal', 'Peril')]

In [22]: c.nbest(measures.likelihood_ratio, 10)

Out[22]: [('I', 'am'),
          ('Well', ','),
          ('boom', 'boom'),
          ('Run', 'away'),
          ('of', 'the'),
          ('Holy', 'Grail'),
          (']', '['),
          ('Brother', 'Maynard'),
          ('Jesus', 'Christ'),
          ('Round', 'Table')]
```

We see here that the collocation finder is pulling out some things that have face validity. When Arthur is talking about peasants, he calls them "bloody" more often than not. However, collocations like "Brother Maynard" and "BLACK KNIGHT" are less informative to us, because we know that they are proper names.

If you were interested in collocations in particular, what step do you think you would have to take during the tokenizing process?

1.5 Stemming

This has gotten us as far identical tokens, but in language processing, it is often the case that the specific form of the word is not as important as the idea to which it refers. For example, if you are trying to identify the topic of a document, counting 'running', 'runs', 'ran', and 'run' as four separate words is not useful. Reducing words to their stems is a process called stemming.

A popular stemming implementation is the Snowball Stemmer, which is based on the Porter Stemmer. It's algorithm looks at word forms and does things like drop final 's's, 'ed's, and 'ing's.

Just like the tokenizers, we first have to create a stemmer object with the language we are using.

```
In [23]: snowball = nltk.SnowballStemmer('english')
```

Now, we can try stemming some words

```
In [24]: snowball.stem('running')

Out[24]: 'run'

In [25]: snowball.stem('eats')

Out[25]: 'eat'

In [26]: snowball.stem('embarassed')

Out[26]: 'embarass'
```

Snowball is a very fast algorithm, but it has a lot of edge cases. In some cases, words with the same stem are reduced to two different stems.

```
In [27]: snowball.stem('cylinder'), snowball.stem('cylindrical')

Out[27]: ('cylind', 'cylindr')
```

In other cases, two different words are reduced to the same stem.

This is sometimes referred to as a 'collision'

```
In [28]: snowball.stem('vacation'), snowball.stem('vacate')

Out[28]: ('vacat', 'vacat')

In [29]: snowball.stem('organization'), snowball.stem('organ')

Out[29]: ('organ', 'organ')

In [30]: snowball.stem('iron'), snowball.stem('ironic')

Out[30]: ('iron', 'iron')

In [31]: snowball.stem('vertical'), snowball.stem('vertices')

Out[31]: ('vertic', 'vertic')
```

A more accurate approach is to use an English word bank like WordNet to call dictionary lookups on word forms, in a process called lemmatization.

```
In [32]: # nltk.download('wordnet')
         wordnet = nltk.WordNetLemmatizer()

In [33]: wordnet.lemmatize('iron'), wordnet.lemmatize('ironic')

Out[33]: ('iron', 'ironic')

In [34]: wordnet.lemmatize('vacation'), wordnet.lemmatize('vacate')

Out[34]: ('vacation', 'vacate')
```

Nothing comes for free, and you've probably noticed already that the lemmatizer is slower. We can see how much slower with one of IPYthon's `magic functions`.

```
In [35]: %timeit wordnet.lemmatize('table')

100000 loops, best of 3: 5.39 s per loop

In [36]: 4.45 * 5.12

Out[36]: 22.784000000000002

In [37]: %timeit snowball.stem('table')

100000 loops, best of 3: 16.7 s per loop
```

Time for another small challenge! Switch computers for this one, so that you are using your partner's computer, and try your hand at challenge B!

1.6 Sentiment

Frequently, we are interested in text to learn something about the person who is speaking. One of these things we've talked about already - linguistic diversity. A similar metric was used a couple of years ago to settle the question of who has the largest vocabulary in Hip Hop.

> Unsurprisingly, top spots go to Canibus, Aesop Rock, and the Wu Tang Clan. E-40 is also in the top 20, but mostly because he makes up a lot of words; as are OutKast, who print their lyrics with words slurred in the actual typography

Another thing we can learn is about how the speaker is feeling, with a process called sentiment analysis. Before we start, be forewarned that this is not a robust method by any stretch of the imagination. Sentiment classifiers are often trained on product reviews, which limits their ecological validity.

We're going to use TextBlob's built-in sentiment classifier, because it is super easy.

```
In [38]: from textblob import TextBlob

In [39]: blob = TextBlob(arthur)

In [40]: for sentence in blob.sentences[10:25]:
             print(sentence.sentiment.polarity, sentence)
```

```
-0.3125 What do you mean?
0.8 The swallow may fly south with the sun or the house martin or the plover may seek warmer climes in w
0.0 Not at all.
0.0 They could be carried.
0.0 It could grip it by the husk!
0.0 Well, it doesn't matter.
0.0 Will you go and tell your master that Arthur from the Court of Camelot is here.
0.0 Please!
-0.15625 I'm not interested!
0.25 Will you ask your master if he wants to join my court at Camelot?!
0.125 Old woman!
0.0 Man.
-0.5 Sorry.
0.13636363636363635 What knight live in that castle over there?
0.0 I-- what?
```

1.7 Semantic distance

Another common NLP task is to look for semantic distance between documents. This is used by search engines like Google (along with other things like PageRank) to decide which websites to show you when you search for things like 'bike' versus 'motorcycle'.

It is also used to cluster documents into topics, in a process called topic modeling. The math behind this is beyond the scope of this course, but the basic strategy is to represent each document as a one-dimensional array, where the indices correspond to integer ids of tokens in the document. Then, some measure of semantic similarity, like the cosine of the angle between unitized versions of the document vectors, is calculated.

Luckily for us there is another python library that takes care of the heavy lifting for us.

```
In [41]: from gensim import corpora, models, similarities
```

We already have a document for Arthur, but let's grab the text from someone else to compare it with.

```
In [42]: p = re.compile(r'(?:GALAHAD: )(.+)')
         galahad = ' '.join(re.findall(p, document))
         arthur_tokens = tokens
         galahad_tokens = word_tokenize(galahad)
```

Now, we use gensim to create vectors from these tokenized documents:

```
In [43]: dictionary = corpora.Dictionary([arthur_tokens, galahad_tokens])
         corpus = [dictionary.doc2bow(doc) for doc in [arthur_tokens, galahad_tokens]]
         tfidf = models.TfidfModel(corpus, id2word=dictionary)
```

Then, we create matrix models of our corpus and query

```
In [44]: query = tfidf[dictionary.doc2bow(['peasant'])]
         index = similarities.MatrixSimilarity(tfidf[corpus])
```

```
WARNING:gensim.similarities.docsim:scanning corpus to determine the number of features (consider settin
```

And finally, we can test our query, "peasant" on the two documents in our corpus

```
In [45]: list(enumerate(index[query]))
```

```
Out[45]: [(0, 0.017683197), (1, 0.0)]
```

So we see here that "peasant" does not match Galahad very well (a really bad match would have a negative value), and is more similar to the kind of speach output that we see from King Arthur.

2 Practice

In the time remaining, pull up a dataset that you have, and that you'd like to work with in Python. The instructors will be around to help you apply what you've learned today to problems in your data that you are dealing with.

If you don't have data of your own, you should practice with the test data we've given you here. For example, you could try to figure out:

1. Is King Arthur happier than Sir Robin, based on his speech?
2. Which character in Monty Python has the biggest vocabulary?

```
In [46]:
```