Text as Data Workshop

March 8, 2018

1 A note of the version of Python

Please note that this notebook assumes Python 2.7. If you have Python 3 installed on your machine, you make a handful of edits to the code and it should work out-of-the-box. Particularly, you will need to change the `print` statement (Python 2.7) to the `print()` function (Python 3). Python 3 users should also use the normal `csv` library to load CSV files (as opposed to the `unicodecsv` library used in Python 2.7.

2 Prepocessing

Prior to analysis, there is quite a bit of pre-processing that needs to be carried out. Note that the appropriate pre-processing is contingent on the particular objectives of your analysis; however, we will go over some "standard" pre-processing in this section.

We will be using the academic profiles of Politics staff in this section. You can read in the profile CSV as follows:

```
In [ ]: import unicodecsv as csv
        # Note: for python 3, you would simply use,
        # import csv

        # Note that you will need to change the correct
        # working directory
        with open('profiles.csv', 'r') as csvfile:
            csvreader = csv.reader(csvfile, encoding='utf8')
            profiles = [row for row in csvreader]

        # Note: for python 3, you would use print()
        print 'Loaded %s staff profiles...' % len(profiles)
        print profiles[0]
```

For this analysis, we will combine all of the text found on the staff profile page and the "Research" page:

```
In [ ]: # Combine relevant text
        for row in profiles:
            row.append(row[-2] + '\n\n' + row[-1])
```

2.1 NLTK for text pre-processing

We will use a number of functions from the `NLTK` library in Python. Start by loading the relevant functions:

```
In [ ]: # Tokenization
        from nltk.tokenize import sent_tokenize
        from nltk.tokenize import word_tokenize
        from nltk.tokenize import RegexpTokenizer

        # Stopwords
        from nltk.corpus import stopwords

        # Stemming and lemmatization
        from nltk.stem.porter import PorterStemmer
        from nltk.stem import WordNetLemmatizer
```

2.2 Tokenization

We often start by "tokenizing" our text into smaller units. For instance, we can use `sent_tokenize` function from NLTK break a staff profile into sentences:

```
In [ ]: sentences = sent_tokenize(profiles[0][-1])
        print sentences[1]
        print len(sentences)
```

Next, we can tokenize into individual words:

```
In [ ]: words = word_tokenize(sentences[1])
        print words
```

I tend to use a **regular expression** tokenizer,

```
In [ ]: words = RegexpTokenizer(r'\w+').tokenize(sentences[1])
        print words
```

Which only keeps "words" (`\w`) and thus removes punctuation.

So far we have tokenized an individual sentence in Rob's profile. How do we go about tokenizing the entire corpus? I typically go with something like the following:

```
In [ ]: def tokenize(text):
            return RegexpTokenizer(r'\w+').tokenize(text)

        tokens = [tokenize(row[-1].lower()) for row in profiles]

        # Print the first 10 tokens of Rob's profile
        print tokens[0][0:10]
```

2.3 Stopword removal

After tokenization, the next step is usually to remove so-called **stopwords**. These are words that show up frequently in a text, but add very little meaning.

```
In [ ]: # Pull in NLTK's English language stopword list
        stops = stopwords.words('english')
        print stops[0:10]
```

So these are the words we want to remove. What's the best way to remove stopwords? I typically go with something like:

```
In [ ]: # Define a function that removes stopwords
        def remove_stops(text, stops):
            return [token for token in text if token not in set(stops)]

        print remove_stops(tokens[0], stops)
```

Note that there is still some junk at the begining of Rob's profile (email, extension, etc.). It's not uncommon to have "corpus-specific" (or "extended") stopwords. We can add additional words to the stopword list:

```
In [ ]: extended_stops = [u'email', u'extension', u'telephone', 'office', 'hours']
        for token in extended_stops:
            stops.append(token)

        print stops[-10:-1]
```

```
In [ ]: print remove_stops(tokens[0], stops)
```

And we can remove stopwords for each profile in the corpus using our now standard list comprehension:

```
In [ ]: tokens_no_stops = [remove_stops(doc, stops) for doc in tokens]
        print tokens_no_stops[1]
```

2.3.1 Removing numbers

As we can see in the last couple of profiles, there are numbers scattered throughout. We often (but not always!) don't care about numbers and thus they can be removed.

```
In [ ]: import re # remove number function uses a RegEx

        def remove_number(text):
            return [token for token in text if re.search(r'^\d+$', token) == None]

        tokens_clean = [remove_number(doc) for doc in tokens_no_stops]

        print tokens_clean[0][0:10]
```

2.4 Text normalization: stemming and lemmatizing

It's often the case that we want to combine similar words into a single token. For instance, we might not want to make a distinction between statistic, statistics, and statistical in our staff profile example. There two primary ways in which this sort of normalization is done: stemming and lemmatizing.

Stemming is perhaps the simplest normalization techique. Let's use the well-known "Porter stemmer" to normalize our "statistics" keywords:

```
In [ ]: print PorterStemmer().stem('statistic')
        print PorterStemmer().stem('statistics')
        print PorterStemmer().stem('statistical')
```

So our stemmer is pretty crude: it just lops off the end of our tokens. In contrast, **lemmatization** tries to do things a bit more properly, returning the base dictionary form of a word:

```
In [ ]: # Instantiate an NLTK lemmatizer object
        lemmatizer =  WordNetLemmatizer()

        print lemmatizer.lemmatize('statistic')
        print lemmatizer.lemmatize('statistics')
        print lemmatizer.lemmatize('statistical')
```

We can stem or lemmatize our entire corpus by defining functions in the usual way:

```
In [ ]: def stem(text):
            return [PorterStemmer().stem(token) for token in text]

        def lemmatize(text, lemmatizer):
            return [lemmatizer.lemmatize(token) for token in text]
```

Which we can then use in a list comprehension:

```
In [ ]: tokens_final = [lemmatize(doc, lemmatizer) for doc in tokens_clean]
```

2.5 Wrapping up

Let's wrap things up by making a dataset with relevant meta-data and our `tokens_final` list.

```
In [ ]: content = []
        for i, row in enumerate(profiles):
            content.append({'name': row[0],
                            'position': row[1],
                            'url': row[3],
                            'text': row[-1],
                            'tokens': tokens_final[i]})
```

The final pre-processing step (at least for us) is to remove profiles with no text. Obviously, we want to ask ourselves why these cases are missing. But, for now, let's just drop them:

```
In [ ]: for row in content:
            if len(row['tokens']) == 0:
                print row['name']

        content = [row for row in content if len(row['tokens']) != 0]
```

3 Dictionary-based methods for text analysis

The use of dictionaries (or "lexicons") for text analysis has a long history in the literature. Using dictionaries has a number of advantages: 1) they are simple to undertand, 2) they are transparent, and 3) they are generally computationally efficient. Nevertheless, there are also a number of key disadvantages, which we will discuss below.

The purpose of this section is to show you how to implement your own dictionaries in Python. We will also cover perhaps the success story when it comes to dictionary-based methods: sentiment analysis.

3.1 Loading our "quantitative" dictionary

We will start by loading and pre-processing a small "quantitative analysis" dictionary:

```
In [ ]: with open('quant_dictionary.txt') as txtfile:
            quant = txtfile.readlines()

        quant = [row.strip() for row in quant]
```

And we need to remember to pre-process our dictionary terms using the same process as for the text:

```
In [ ]: quant = lemmatize(quant, lemmatizer)
        print quant
```

3.2 Using dictionaries

There are a number of ways to implement dictionary

```
In [ ]: import numpy as np
        from sklearn.feature_extraction.text import CountVectorizer
```

By default, the `CountVectorizer` class requires untokenized data, so let's take care of this now:

```
In [ ]: texts = [' '.join(row['tokens']) for row in content]
        print texts[0]
```

We are now ready to use the `CountVectorizer` class to produce a **document-term matrix**:

```
In [ ]: # Instantiate class
        vectorizer = CountVectorizer()

        # Call the fit_transform method to build the DTM
```

```
dtm = vectorizer.fit_transform(texts)

print dtm.shape
```

```
In [ ]: labels = vectorizer.get_feature_names()
        print labels[1000:1010]
        term_freq = np.sum(dtm, axis = 0)
        print term_freq.shape
```

```
In [ ]: idx = np.in1d(labels, quant, assume_unique=True)
        print idx.shape
        print np.sum(idx)
```

With the **document-term matrix** in hand, we can calculate both the total number of words in a document and the number of our dictionary "keywords" that are in each document – i.e., the two key ingredients for dictionary-based methods. We can now lookup the relevant proportion of words for each document in the following way:

```
In [ ]: import scipy

        # Get the token labels that we passed to CountVectorizer()
        labels = vectorizer.get_feature_names()

        # Get a boolean vector for columns (i.e. tokens) where the
        # DTM labels and our dictionary keywords intersect
        idx = np.in1d(labels, quant, assume_unique=True)

        # Sum the keyword frequencies in each document
        keyword_count = np.sum(dtm[:,idx], axis = 1, dtype='float')

        # Sum the total number of tokens in each document
        N = np.sum(dtm, axis = 1)

        # Calculate the proportion of keywords in each document
        keyprop = keyword_count/N
        print keyprop[0:10]
```

```
In [ ]: keyprop_list = keyprop.tolist()
        for i,row in enumerate(content):
            row['keyword_prop'] = keyprop_list[i]
```

```
In [ ]: # Print list, sorted by the keyword proportion
        result = [[row['name'], row['keyword_prop'][0]] for row in content]
        sorted_result = sorted(result, key=lambda x: x[1], reverse=True)

        for row in sorted_result:
            print row[0], row[1]
```

3.3 Sentiment analysis

Sentiment analysis is are of text analysis where dictionaries have preformed really well. Let's take a look at sentiment analysis using the `vaderSentiment` library. First, open the Anaconda prompt, and

`pip install vaderSentiment`

Using vaderSentiment is easy:

```
In [ ]: from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

        analyser = SentimentIntensityAnalyzer()

        print analyser.polarity_scores("This class fucking great!")
```

4 Topic models

We will use the `gensim` library to estimate topic models in Python. To install, open the Anaconda prompt and type:

`pip install gensim`

After installing, we can load the necessary functions:

```
In [ ]: import gensim
        from gensim import corpora
        from gensim.models import LdaModel
        from gensim.models.coherencemodel import CoherenceModel
```

Next, we need to find some text data to use and get it in a format that `gensim` likes. We will pull in some data on the Heartland Institute (i.e., a climate change skeptic think tank in the US):

```
In [ ]: import pickle

        with open('/home/tcoan/heartland.pkl', 'r') as pfile:
            heartland = pickle.load(pfile)
```

```
In [ ]: # Prepare data for gensim
        documents = [row['text_postprocessed'] for row in heartland]
        texts = [doc.split() for doc in documents]

        # Make gensim corpus object
        dictionary = corpora.Dictionary(texts)

        # Gensim corpus object
        corpus = [dictionary.doc2bow(text) for text in texts]
```

Next, we are ready to estimate our model:

```
In [ ]: # Estimate!
        k = 30
        lda = LdaModel(corpus, num_topics=k, iterations=500, alpha = 'auto',
                       id2word=dictionary, random_state=12345)
```

```
In [ ]: from prettytable import PrettyTable

        def view_keywords(model, num_topics, num_words = 10, prettyprint = True):
            # Return keywords from gensim
            keywords = model.show_topics(num_topics = num_topics,
                                         num_words = num_words,
                                         formatted=False)

            # Reformat keyword results for easy viewing
            output = []
            for row in keywords:
                tokens = ' '.join([token[0] for token in row[1]])
                output.append([row[0], tokens])

            # Print a nicely formatted table
            if prettyprint:
                tbl = PrettyTable()

                # Column labels
                tbl.field_names = ["Topic ID", "Keywords"]

                # Populate table
                for row in output:
                    tbl.add_row(row)

                # Output formatted table
                tbl.align = "l"
                print tbl

            return output

        keywords = view_keywords(lda, lda.num_topics, num_words = 15)
```

4.1 Topic interpretation

One useful way to view topic models generated by `gensim` is using the `PyLDAvis` library. To install, type the following in the Anaconda console:

```
pip install pyldavis
```

And then to run:

```
In [ ]: import pyLDAvis.gensim as gensimvis
        import pyLDAvis

        vis_data = gensimvis.prepare(lda, corpus, dictionary)
        pyLDAvis.display(vis_data)

        # You save an HTML file with the same information, using
        # pyLDAvis.save_html(vis_data,'vis.html')
```

4.2 Topic quality and semantic coherence

One useful measure of topic quality is so-called semantic (or topic) coherence. It is easy to implement a wide-range of coherence measures in `gensim`:

```
In [ ]: # Pull out and tokenize the topic keywords
        topics = [row[1].split(' ') for row in keywords]

        # Estimate coherence
        co = CoherenceModel(topics=topics,
                            texts=texts,
                            dictionary=dictionary,
                            coherence='c_v')

        # Extract semantic coherence for each topic
        semantic = co.get_coherence_per_topic()
```

While you could view coherence in the Python console, I typically prefer to write everything out to a CSV file:

```
In [ ]: import unicodecsv as csv

        topics_to_write = []
        for i,topic in enumerate(topics):
            topics_to_write.append([i, semantic[i], ' '.join(topic)])

        # Insert labels
        topics_to_write.insert(0, [u'id', u'coherence', u'keywords'])

        # Write
        with open('/home/tcoan/Desktop/topics_heartland.csv', 'w') as csvfile:
            csvwriter = csv.writer(csvfile, encoding='utf8')
            csvwriter.writerows(topics_to_write)
```

4.3 Extracting topic model data

Once you have a model that you are happy with, you can extact the document-topic matrix for future use. For instance, consider the following code:

```
In [ ]: # Get the document-topic matrix
        theta = lda.get_document_topics(corpus, minimum_probability = 0)

        # Parse "theta" and store topic proportions
        proportions = []
        for row in theta:
            proportions.append([el[1] for el in row])

        print proportions[0]
```

This will provide a Python list of the relevant topic probabilities in the same order as the orginal `heartland` corpus. We can append these probabilities to our dataset and write to disk:

```
In [ ]: # Define a little function to "flatten" a list
        def flatten(list_of_lists):
            return [item for sublist in list_of_lists for item in sublist]

        # Prepare file to write
        data_to_write = []
        for i,row in enumerate(proportions):
            meta = [heartland[i]['docid'], heartland[i]['date'], heartland[i]['title']]
            data_to_write.append(flatten([meta, row]))

        # Write a CSV file
        with open('/home/tcoan/Desktop/heartland_data.csv', 'w') as csvfile:
            csvwriter = csv.writer(csvfile, encoding='utf8')
            csvwriter.writerows(topics_to_write)
```