Natural Language Processing

Erich Wellinger

09/27/2016

Morning Objectives:

- Text as data
- Definitions
- Text Processing Steps
- Text Vectorization
  - Stop Words
  - Text Stemming/Lemmatization
  - POS Tagging
  - N-grams
- Document similarity

Text as Data

Why do we care about Natural Language Processing? There is an *immense* amount of data that exists in text form and we would like to make sense of it just how we'd like to make sense of any other source of data. If we completely ignore text, we are effectively throwing out valuable data that could be used to create standalone models or even be incorporated into a greater model that incorporates other data (think combining not only the rating a customer gave a product but also incorporating the comment they wrote about their experience).

Natural Language Processing is a subfield of machine learning focused on making sense of text. Text is inherently unstructured but there are a variety of techniques for converting (vectorizing) text into a format that a machine learning algorithm can interpret.

Definitions

- **Token**: Words in a list. Each document is a list of tokens.
- **Document**: Text in a string.
- **Corpus**: Collection of all documents you're interested in.
- **Vocab**: Set of words.
- **Vectorization**: Vector Representation of our list of words.
- **Stop Words**: Words that don't matter

---

d_0: "Clinton stuns crowd, scares viewers, with debate performance"

d_1: "Viewers impressed by how male Trump looked during debate"

d_2: "Trump and Clinton clash over Economy and Race Relations"

d_3: “Impressed by debate, voters in York County, VA seriously ambivalent”

d_4: “I am serious, and don’t call me Shirley”

d_5: “Breaking: Trump made it to debate in New York state”

Text Processing

The first thing we need to do is process our text. Common steps include:

- Lower all of your text (although you could do this depending on the POS)
- Strip out misc. spacing and punctuation, much to a grammar enthusiasts chagrin
- Remove stop words
  - Stop words are words which have no real meaning but make the sentence grammatically correct. Words like ‘I’, ‘me’, ‘my’, ‘you’, & c. NLTK contains 153 words for the English set of stop words
  - These can also be domain specific.
- Stem/Lemmatize our text
  - The goal of this process is to transform a word into its base form.
  - e.g. “ran”, “runs” -> “run”
  - You can think of the base form as what you would look up in a dictionary
  - Popular techniques include stemming and lemmatization. Stemming removes the suffix whereas Lemmatization attempt to change all forms of the word to the same form. Stemmers tend to operate on a single word without knowledge of the overall context.
  - These are not perfect, however (e.g. taking the lemma of “Paris” and getting “pari”)
- Part-Of-Speech Tagging
- N-grams

After running the following processing this is the resulting text:

```python
import pattern.en as en
from string import punctuation
from nltk.corpus import stopwords

def lemmatize_string(doc):
    stop_words = stopwords.words('english')
    doc = doc.lower().translate(None, punctuation)
    return ' '.join([en.lemma(w) for w in doc.split() if w not in stop_words])


if __name__=="__main__":
    corpus = ['Clinton stuns crowd, scares viewers, with debate performance',
              'Viewers impressed by how male Trump looked during debate',
              'Trump and Clinton clash over Economy and Race Relations',
              'Impressed by debate, voters in York County, VA seriously ambivalent',
              "I am serious, and don't call me Shirley",
              'Breaking: Trump made it to debate in New York state']
    processed = map(lemmatize_string, corpus)
```

---

d_0: ‘clinton stun crowd scare viewer debate performance’

d_1: ‘viewer impress male trump look debate’

d_2: ‘trump clinton clash economy race relation’

d_3: ‘impress debate voter york county va seriously ambivalent’

d_4: ‘seriou dont call shirley’

d_5: ‘break trump make debate new york state’

Text Vectorization

Term Frequency

In order to utilize many of the machine learning algorithms we have learned, we must convert our text data into something that these algorithms can work with. This is typically done by converting our corpus of text data into some form of numeric matrix representation. The most simple form of numeric representation is called a Term-Frequency matrix whereby each column of the matrix is a word, each row is a document, and each cell represents the count of that word in a document.

Note: The matrix shown below is transposed from how it is typically represented in code (i.e. the documents will be rows with the tokens as columns). Also, only a subset of the tokens are displayed in the table below.

- $f_{t,d}$ = count of term t in document d where $t \in T$ and $d \in D$

Token	d_0	d_1	d_2	d_3	d_4	d_5
ambivalent	0	0	0	1	0	0
clinton	1	0	1	0	0	0
trump	0	1	1	0	0	1
impress	0	1	0	1	0	0
debate	1	1	0	1	0	1
serious	0	0	0	1	1	0
break	0	0	0	0	0	1
economy	0	0	1	0	0	0
new	0	0	0	0	0	1
york	0	0	0	1	0	1
race	0	0	1	0	0	0
shirley	0	0	0	0	1	0
voter	0	0	0	1	0	0
viewer	1	1	0	0	0	0

- What problems do you see with this approach?

One issue with this approach is due to the potential difference in document lengths. A longer article that contains the complete transcript of a political debate is necessarily going to have larger values for the term counts than an article that is only a couple of sentences long. This also serves to scale up the frequent terms and scales down the rare terms which are empirically more informative. We could normalize the term counts by the length of a document which would alleviate some of this problem.

- L2 Normalization is the default in Sklearn

$$tf(t,d) = \frac{f_{t,d}}{\sqrt{\sum_{i \in V}(f_{i,d})^2}}$$

For example:

Token	d_0	d_1	d_2	d_3	d_4	d_5
ambivalent	0	0	0	$\frac{1}{\sqrt{6}}$	0	0
clinton	$\frac{1}{\sqrt{3}}$	0	$\frac{1}{\sqrt{4}}$	0	0	0
trump	0	$\frac{1}{\sqrt{4}}$	$\frac{1}{\sqrt{4}}$	0	0	$\frac{1}{\sqrt{5}}$

But we can go further and have the value associated with a document-term be a measure of the importance in relation to the rest of the corpus. We can achieve this by creating a Term-Frequency Inverse-Document-Frequency (TF-IDF) matrix. This is done by multiplying the Term-Frequency by a statistic called the Inverse-Document-Frequency, which is a measure of how much information a word provides (i.e. it is a measure of whether a term is common or rare across all documents).

$$idf(t,D) = log\frac{|D|}{|\{d \in D : t \in d\}| + 1}$$

For example:

$$\begin{aligned} idf('ambivalent', D) &= log\frac{6}{1+1} &= 1.099 \\ idf('clinton', D) &= log\frac{6}{2+1} &= 0.693 \\ idf('trump', D) &= log\frac{6}{3+1} &= 0.405 \end{aligned}$$

This rational for using the logarithmic scale being that a term that occurs 10 times more than another isn't 10 times more important than it. The 1 term on the bottom is known as a smoothing constant and is there to ensure that we don't have a zero in the denominator.

- Why do we need the smoothing constant?

The end result then is thus...

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

For example:

$$tfidf('ambivalent', d_3, D) = \frac{1}{\sqrt{6}} \cdot \frac{6}{1+1} = 0.449$$

Token	d_0	d_1	d_2	d_3	d_4	d_5
ambivalent	0	0	0	0.449	0	0
clinton	0.400	0	0.347	0	0	0

Token	d_0	d_1	d_2	d_3	d_4	d_5
trump	0	0.203	0.203	0	0	0.181

What does this intuitively tell us? What does a high score mean? Roughly speaking a $tfidf$ score is an attempt to identify the most important words in a document. If a word appears a lot in a particular document it will get a high tf score. But if a word also appears in every other document in your corpus, it clearly doesn't convey anything unique about what that document is about. Thus, a term will get a high score if it occurs many times in a document and appears in a small fraction of the corpus.

TF-IDF in Scikit-Learn

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Sklearn provides a convenient manner for creating this matrix. Some of the arguments to note are:

- `max_df`
  - Can either be absolute counts or a between 0 and 1 indicating a proportion. Specifies words which should be excluded due to appearing in more than a given number of documents.
- `min_df`
  - Can either be absolute counts or a between 0 and 1 indicating a proportion. Specifies words which should be excluded due to appearing in less than a given number of documents.
- `max_features`
  - Specifies the number of features to include in the resulting matrix. If not `None`, build a vocabulary that only considers the top `max_features` ordered by term frequency across the corpus.

Document Similarity

Now that we have a matrix representation of our corpus, how should we go about comparing documents to identify those which are most similar to one another?

- What are some metrics that you can think of and why might you prefer one over the other?
- Cosine Similarity

$$similarity = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

- Euclidean Distance

$$d(A, B) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics.pairwise import euclidean_distances
```

Afternoon Objectives:

- Naive Bayes and why it's Naive
- Multiclass Classification
- Laplace Smoothing

Naive Bayes

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' Theorem with strong (naive) independence assumptions between the features. Why is it naive?

- Naive Bayes classifiers are considered naive because we assume that all words in the string are *independent* from one another

While this clearly isn't true, they still perform remarkably well and historically were deployed as spam classifiers in the 90's. Naive Bayes handles cases where our number of features vastly outnumber our datapoints (i.e. we have more words than documents). These methods are also computationally efficient in that we just have to calculate sums.

Let's say we have some arbitrary document come in, $(w_1, ..., w_n)$, and we would like to calculate the probability that it was from the sports section. In other words we would like to calculate…

$$P(y_c|w_1, ..., w_n) = P(y_c)\prod_{i=1}^{n} P(w_i|y_c)$$

where…

$$P(y_c) = \frac{\sum y == y_c}{|D|}$$

$$\begin{aligned} P(w_i|y_c) &= \frac{count(w_{D,i}|y_c) + 1}{\sum_{w \in V}[count(w|y_c) + 1]} \\ &= \frac{count(w_{D,i}|y_c) + 1}{\sum_{w \in V}[count(w|y_c)] + |V|} \end{aligned}$$

$$P(y_c|w_{d,1}, ..., w_{d,n}) = P(y_c)\prod_{i=1}^{n} P(w_{d,i}|y_c)$$

$$log(P(y_c|w_{d,1}, ..., w_{d,n})) = log(P(y_c)) + \sum_{i=1}^{n} log(P(w_{d,i}|y_c))$$

- Why do we add 1 to the numerator and denominator? This is called **Laplace Smoothing** and serves to remove the possibility of having a 0 in the denominator or the numerator, both of which would break our calculation.

Doc	Occurrence of 'ball'	Total # of words	class
0	5	101	Sports
1	7	93	Sports
2	3	122	Sports
3	0	39	Politics

Doc	Occurrence of 'ball'	Total # of words	class
4	0	81	Politics
5	0	142	Politics
6	2	77	Art
7	0	198	Art

Now what are the probabilities for each of the classes in this case?

$P(y_s) = \frac{3}{8}$, $P(y_p) = \frac{3}{8}$, and $P(y_a) = \frac{2}{8}$

Let's say we observe the word 'ball' in a document and would like to know the probability that this document belongs to each of the above classes.

$P('ball'|y_s) = \frac{16}{316+|V|}$, $P('ball'|y_p) = \frac{1}{262+|V|}$, $P('ball'|y_p) = \frac{3}{275+|V|}$

Maybe I would like to also calculate the probability of the document "The Cat in The Hat" being in the sports section. Let's assume that we lower the text but don't drop any stop words.

$$\begin{aligned} P(y_s|'the','cat','in','the','hat') &= P(y_s) \cdot \prod_{w \in d} P(w|y_s) \\ &= P(y_s) \cdot P('the'|y_s) \cdot P('cat'|y_s) \cdot P('in'|y_s) \cdot P('the'|y_s) \cdot P('hat'|y_s) \end{aligned}$$

This will in turn yield us...

$$log(P(y_s|'the','cat','in','the','hat')) = log(P(y_s)) + \sum_{w \in d} count_{w,d} \cdot log(P(w|y_s))$$