# Natural Language Processing

Erich Wellinger

January 18, 2017

## Contents



# Morning

## 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

---

## Our Corpus

$d_0$: "Trump honors sacrifices civil rights activists will have to make under his presidency."

$d_1$: "Don't point that gun at him, he's an unpaid intern!"

$d_2$: "Sales of PS4 trump that of Xbox One."

$d_3$: "Trump wonders how best to divide nation."

$d_4$: "Intern wonders if sacrificing basic human dignity worth it."

$d_5$: "Report points to declining healthcare coverage, rising human sacrifice."

---

## 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. Scikit-Learn's contains 318 words for the English set of stop words.
  - These can also be domain specific and so extending your set of stop words based on the use case is common practice.
- 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 this processing[1] we get the following text as the result:

---

$d_0$: ‘trump honor sacrifice civil right activist make presidency’

$d_1$: ‘point gun unpaid intern’

$d_2$: ‘sale ps4 trump xbox’

$d_3$: ‘trump wonder best divide nation’

$d_4$: ‘intern wonder sacrifice basic human dignity worth’

$d_5$: ‘report point decline healthcare coverage rise human sacrifice’

---

```python
import sys
import spacy
from string import punctuation
from sklearn.feature_extraction.stop_words import ENGLISH_STOP_WORDS

# Load the spacy.en module if it hasn't been loaded already
# When in ipython, execute the script using %run -i my_file.py to avoid
# repeatedly loading in the english module
if not 'nlp' in locals():
    print("Loading English Module...")
    nlp = spacy.load('en')

def lemmatize_string(doc, stop_words):
    # First remove punctuation form string
    # .translate works differently from 2 to 3 so check version number
    if sys.version_info.major == 3:
        PUNCT_DICT = {ord(punc): None for punc in punctuation}
        doc = doc.translate(PUNCT_DICT)
    else:
        # spaCy expects a unicode object
        doc = unicode(doc.translate(None, punctuation))

    # Run the doc through spaCy
    doc = nlp(doc)

    # Lemmatize and lower text
    tokens = [token.lemma_.lower() for token in doc]

    return ' '.join(w for w in tokens if w not in stop_words)


if __name__=="__main__":
    corpus = [
        "Trump honors sacrifices civil rights activists will have to make under "\
         "his presidency.",
        "Don't point that gun at him, he's an unpaid intern!",
        "Sales of PS4 trump that of Xbox One.",
        "Trump wonders how best to divide nation.",
        "Intern wonders if sacrificing basic human dignity worth it.",
        "Report points to declining healthcare coverage, rising human sacrifice."
        ]

    # Example of extending our STOPLIST
    STOPLIST = set(list(ENGLISH_STOP_WORDS) + ["n't", "'s", "'m"])

    processed = [lemmatize_string(doc, STOPLIST) for doc in corpus]
```

Source Code 1: Example code for lemmatizing text in either Python 2 or 3

## 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 | $d_0$ | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ |
|---|---|---|---|---|---|---|
| human | 0 | 0 | 0 | 0 | 1 | 1 |
| intern | 0 | 1 | 0 | 0 | 1 | 0 |
| point | 0 | 1 | 0 | 0 | 0 | 1 |
| sacrifice | 0 | 0 | 0 | 0 | 1 | 1 |
| trump | 1 | 0 | 1 | 1 | 0 | 0 |
| unpaid | 0 | 1 | 0 | 0 | 0 | 0 |
| wonder | 0 | 0 | 0 | 1 | 1 | 0 |
| activist | 1 | 0 | 0 | 0 | 0 | 0 |
| civil | 1 | 0 | 0 | 0 | 0 | 0 |
| decline | 0 | 0 | 0 | 0 | 0 | 1 |
| divide | 0 | 0 | 0 | 1 | 0 | 0 |
| gun | 0 | 1 | 0 | 0 | 0 | 0 |
| healthcare | 0 | 0 | 0 | 0 | 0 | 1 |
| xbox | 0 | 0 | 1 | 0 | 0 | 0 |

**NOTE**: The above terms are only a subset of the terms from our corpus

- 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 | $d_0$ | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ |
|---|---|---|---|---|---|---|
| human | 0 | 0 | 0 | 0 | $\frac{1}{\sqrt{4}}$ | $\frac{1}{\sqrt{5}}$ |
| intern | 0 | $\frac{1}{\sqrt{4}}$ | 0 | 0 | $\frac{1}{\sqrt{4}}$ | 0 |
| point | 0 | $\frac{1}{\sqrt{4}}$ | 0 | 0 | 0 | $\frac{1}{\sqrt{5}}$ |
| sacrifice | 0 | 0 | 0 | 0 | $\frac{1}{\sqrt{4}}$ | $\frac{1}{\sqrt{5}}$ |
| trump | $\frac{1}{\sqrt{3}}$ | 0 | $\frac{1}{\sqrt{2}}$ | $\frac{1}{\sqrt{3}}$ | 0 | 0 |

**Term-Frequency Inverse-Document-Frequency**

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:

$$\begin{aligned} idf('human', D) &= log\frac{6}{2+1} &= 0.693 \\ idf('intern', D) &= log\frac{6}{2+1} &= 0.693 \\ idf('point', D) &= log\frac{6}{2+1} &= 0.693 \\ idf('sacrifice', D) &= log\frac{6}{2+1} &= 0.693 \\ idf('trump', D) &= log\frac{6}{3+1} &= 0.405 \end{aligned}$$

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:

$$tfidf('human', d_4, D) = \frac{1}{\sqrt{4}} \cdot log\frac{6}{2+1} = 0.347$$

| Token | $d_0$ | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ |
|---|---|---|---|---|---|---|
| human | 0 | 0 | 0 | 0 | 0.347 | 0.31 |
| intern | 0 | 0.347 | 0 | 0 | 0.347 | 0 |
| point | 0 | 0.347 | 0 | 0 | 0 | 0.31 |
| sacrifice | 0 | 0 | 0 | 0 | 0.347 | 0.31 |
| trump | 0.234 | 0 | 0.287 | 0.234 | 0 | 0 |

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
```

Scikit-Learn 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.

```
1  from sklearn.feature_extraction.text import TfidfVectorizer
2
3  c_train = ['Here is my corpus of text it says stuff and things',
4             'Here is some other document']
5  c_test = ['Yet another document',
6            'This time to test on']
7
8  tfidf = TfidfVectorizer()
9  tfidf.fit(c_train)
10 test_arr = tfidf.transform(c_test).todense()
11
12 # Print out the feature names
13 print(tfidf.get_feature_names())
```

Source Code 2: Example Usage of sklearn's TfidfVectorizer

## 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
```

## spaCy

The spaCy package is an industrial-strength Natural Language Processing tool in Python. spaCy can be used to perform lemmatization, part-of-speech tagging, sentence extraction, entity extraction, and more; all while excelling at large-scale information extraction tasks. Leveraging the power of Cython, spaCy is the fastest syntactic parser in the world and is capable of parsing over 13,000 words per minute.[1]

Additionally, spaCy comes with pre-trained word vectors for thousands of common words[2] allowing you to easily leverage the power of Word2Vec. It is also a simple matter to overload these vector representations with our own, use-case dependent, model if you so desire. In fact, spaCy is one of the best ways to prepare your text for deep learning purposes due to the seamless nature with which it integrates with TensorFlow, Keras, Scikit-Learn, and Gensim. For a small taste of how seamless this integration is, check out the tutorial on hooking a deep learning model into spaCy.

And in case all that still isn't peaking your interest, check out spaCy visualization tool for peering into spaCy's guess at the syntactic structure of a sentence!

### Installing spaCy

We must first install spaCy by running the following command:

```
$ conda install spacy
```

After downloading the package we must also download the English module by running the following command. **NOTE**: The English module is a hefty 500+ MB. This can take a long time so be patient!

[1] https://spacy.io/docs/api/#benchmarks
[2] By default, spaCy 1.0 downloads and uses 300-dimensional GloVe word vectors.

```
$ python -m spacy.en.download
```

### Basic Usage Examples

The following will give you some general ideas at how to start leveraging the power of `spaCy`, but I would highly encourage you to check out this intro to NLP with `spaCy` tutorial which much of this code was inspired by.

**NOTE**: When using `spaCy` in Python 2 the text processor is expecting text in **unicode** form!

### Lemmatizing text

As we saw before, lemmatizing text is a fairly straightforward process. The following code assumes we have loaded `spaCy`'s English module into the variable `nlp`.

```
doc = nlp(my_string)

lemmatized_tokens = [token.lemma_ for token in doc]
```

### Extracting sentences

```
def sentence_list(doc):
    ''' Extract sentences from a spaCy document object

    Parameters
    ----------
    doc: spacy.token.Doc

    Returns
    --------
    list of sentences as strings
    '''
    sents = []
    # The sents attribute returns spans which have indices into the original
    # spacy.tokens.Doc. Each index value represents a token
    for span in doc.sents:
        sent = ''.join(doc[i].string for i in range(span.start,
                       span.end)).strip()
        sents.append(sent)
    return sents
```

Source Code 3: Extracting Sentences using ‘spaCy‘

### Getting similar words using word2vec

The following examples shows how you can leverage the power of the built-in vector representations of words.

```
import numpy as np

def get_similar_words(wrd, top_n=10):
    token = nlp(wrd)[0]
    if not token.has_vector:
        raise ValueError("{} doesn't have a vector representation".format(wrd))

    cosine = lambda v1, v2: np.dot(v1, v2) / (norm(v1) * norm(v2))

    # Gather all words spaCy has vector representations for
    all_words = list(w for w in nlp.vocab if w.has_vector
                     and w.orth_.islower() and w.lower_ != token.lower_)

    # Sort by similarity to token
    all_words.sort(key=lambda w: cosine(w.vector, token.vector))
    all_words.reverse()

    print("Top {} most similar words to {}:".format(top_n, token))
    for word in all_words[:top_n]:
        print(word.orth_, '\t', cosine(word.vector, token.vector))
```

Source Code 4: Extract Similar Words Using Vector Representation

# Afternoon

## 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 data points (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...

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

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

$$= \frac{count(w_{D,i}|y_c) + 1}{\sum_{w \in V}[count(w|y_c)] + |V|}$$

---

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

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

## Laplace Smoothing

- 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.

## Example of Multi-class Classification

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

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

$$= P(y_s) \cdot P('the'|y_s) \cdot P('cat'|y_s) \cdot P('in'|y_s) \cdot P('the'|y_s) \cdot P('hat'|y_s)$$

This will in turn yield us...

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