

Feature Engineering for Text Data



<https://github.com/learn-co-curriculum/dsc-feature-engineering-for-text-data>



<https://github.com/learn-co-curriculum/dsc-feature-engineering-for-text-data/issues/new>

Introduction

In this lesson, we'll examine some common approaches to feature engineering for text data.

Objectives

You will be able to:

- Explain what stop words are and why they are frequently removed
- Explain stemming and lemmatization
- Define bigrams and n-grams
- Define mutual information in the context of NLP

Common Approaches to NLP Feature Engineering

As you've likely noticed by now, working with text data comes with *a lot* of ambiguity. When all we start with is an arbitrarily-sized string of words, there's no clear answer as to what sorts of features we should engineer, or even where we should start! The goal of this lesson is to provide a framework for working with text data, and help us figure out exactly what sorts of features we should create when working with text data.

In this lesson, we'll focus on the following topics:

- Stopword Removal
- Frequency Distributions
- Stemming and Lemmatization
- Bigrams, N-grams, and Mutual Information Score

Removing Stop Words

When working with text data, one of the first steps to try is to remove the **Stop Words** from the text. One common feature of text data (regardless of language!) is the inclusion of stop words for grammatical structure. Words such as "a", "and", "but", and "or" are examples of stop words. While a sentence would be both grammatically incorrect and hard to understand without them, from a modeling standpoint, stop words provide little to no actual value. If we create a **Frequency**

Distribution to see the number of times each word is used in a corpus, we'll almost always find that the top spots are dominated by stop words, which tell us nothing about the actual content of the corpus. Removing stop words allows us to reduce the overall dimensionality of our dataset (which is always a good thing), while also distilling the overall vocabulary of our bag-of-words down only to the words that really matter.

NLTK makes it extremely easy to remove stopwords. The library includes a full corpus of all stopwords for all the languages *NLTK* supports. Since we usually only want the stopwords relevant to the language our text data is in, *NLTK* even makes it easy to filter out the unneeded stop words and grab only the ones that pertain to our problem.

The following example shows how we can get all the stopwords for English from *NLTK*:

```
from nltk.corpus import stopwords
import string

# Get all the stop words in the English language
stopwords_list = stopwords.words('english')

# It is generally a good idea to also remove punctuation

# Now we have a list that includes all english stopwords, as well as all punctuation
stopwords_list += list(string.punctuation)
```

Once we have a list of stopwords, we can easily remove them from our text data after we've tokenized our data. Recall that we can easily tokenize text data using *NLTK*'s `word_tokenize()` function. Once we have a list of word tokens, all we need to do is use a list comprehension, and omit any tokens that can be found in our stopwords list. For example:

```
from nltk import word_tokenize

tokens = word_tokenize(some_text_data)

# It is usually a good idea to lowercase all tokens during this step, as well
stopped_tokens = [w.lower() for w in tokens if w not in stopwords_list]
```

Frequency Distributions

Once we have tokenized our data and removed all the stop words, the next step is usually to explore our text data through a **Frequency Distribution**. This is just a fancy way of saying that we create a histogram that tells us the total number of times each word is used in a given corpus.

Once we have tokenized our text data, we can use *NLTK* to easily create a frequency distribution using `nltk.FreqDist()`. A frequency distribution is analogous to a Python dictionary, with a few more

bells and whistles attached to make it easier to use for NLP tasks. Each key is a word token, and each value is the corresponding number of times that token appeared in the tokenized corpus given to the `FreqDist` object at instantiation.

We can easily filter a `FreqDist()` object to see the most common words by using the `.most_common()` built-in method, as seen below:

```
from nltk import FreqDist
freqdist = FreqDist(tokens)


# get the 200 most common words
most_common = freqdist.most_common(200)
```

Once we have the most common words, we can easily use this to filter out the text and reduce the dimensionality of particularly large datasets, as needed.

Stemming and Lemmatization

Consider the words 'run', 'running', 'ran', and 'runs'. If we create a basic frequency distribution, each of these words will be treated as a separate token. After all, they are different words. However, we know that they pretty much mean the same thing. Counting these words as individual separate tokens can sometimes hurt our model by needlessly increasing dimensionality, and hiding important information from our model. Although we instinctively know that those four words are all talking about the same action, our model will default to thinking that they are four completely different concepts. The way we deal with this is to remove suffixes through techniques such as **Stemming** or **Lemmatization**.

People often get stemming and lemmatization confused, because they are extremely similar. They generally accomplish the same task, but they use different means to do so.

Stemming follows a predetermined set of rules to reduce a word to its *stem*. Words like 'running' and 'runs' will be reduced down to 'run', because the stemmer contains rules that understands how to deal with suffixes such as '-ing' and '-s'. The best stemmer currently available is the **Porter Stemmer**. For code samples demonstrating how to use it, check out NLTK's documentation for the [Porter Stemmer](http://www.nltk.org/howto/stem.html)  <http://www.nltk.org/howto/stem.html>.

Lemmatization differs from stemming in that it reduces each word down to a linguistically valid **lemma**, or root word. It does this through stored linguistic mappings. Lemmatization is generally more complex, but also more accurate. This is because the rules that guide things like the Porter Stemmer are good, but far from perfect. For example, stemmers commonly deal with the suffix **-ed** by just dropping it from the word. This usually works, until it runs into an edge case like the word 'agreed'. When stemmed, 'agreed' becomes 'agre'. Lemmatization does not make this mistake,

because it contains a mapping for the word that tells it what 'agreed' should be reduced down to. Generally, most lemmatizers make use of the famous **WordNet** lexical database.

NLTK makes it quite easy to make use of lemmatization, as demonstrated below:

```
from nltk.stem.wordnet import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

lemmatizer.lemmatize('feet') # foot
lemmatizer.lemmatize('running') # run
```

Bigrams and Mutual Information Score

Another alternative to tokenization is to instead create **Bigrams** out of the text. A bigram is just a pair of adjacent words, treated as a single unit.

Consider the sentence "the dog played outside". If we created bigrams out of this sentence, we would get ('the', 'dog'), ('dog', 'played'), ('played', 'outside'). From a modeling perspective, this can be quite useful, because sometimes pairs of words are greater than the sum of their parts. Note that bigrams are just a special case of **n-grams** -- we can choose any number of words for a sequence. Alternatively, it's quite common to create n-grams at the character level, rather than the word level.

One handy feature of bigrams is that we can apply a frequency filter to only keep bigrams that show up more than a set number of times. In this way, we can get rid of all bigrams that only occur because of random chance, and keep the bigrams that must mean something, because they occur together multiple times. How strict your frequency filter should be depends on a number of factors, and generally, it's something you'll have to experiment with to get right. However, most experts tend to apply a minimum frequency filter of 5.

Another way we can make use of bigrams is to calculate their **Pointwise Mutual Information Score**. This is a statistical measure from information theory that generally measures the mutual dependence between two words. In plain english, this measures how much information the bigram itself contains by computing the dependence between the two words in the bigram. For instance, the bigram ('San', 'Francisco') would likely have a high mutual information score, because when these tokens appear in the text, it is highly likely that they appear together, and unlikely that they appear next other words.

In practice, you don't need to worry too much about how to calculate mutual information, because NLTK provides an easy way to do this for us. We'll explore this in detail in the next lab. Instead, your main takeaway on this topic should be that mutual information scores are a type of feature that you

can engineer for text data that may provide good information for you when it comes to exploring the text data or fitting a model to it.

Summary

In this lesson, we learned about various types of feature engineering we can perform on text data, and what each one means!