Natural Language Processing

Muchang Bahng

June 23, 2023

Contents

We will use NLTK, PyTorch.

1 Basics

1.1 Regular Expressions

1.2 Tokenization and Stop Words

Tokenization is the process of dividing up a corpus or a complex sentence into words, also known as tokens. Using a streamlined series of regular expression evaluation, we are able to detect various words to tokenize. Note the following facts:

1. Uppercase and lowercase versions of the same word (e.g. `Language` vs `language`) are two different tokens.
2. Punctuations and special characters also count as a token (e.g. `.`, `,`, `(`, `)`).
3. Hyphenated words and some words with apostrophes are not separated, even though both components are valid words, since they are meant to be used together (e.g. `rock’n’roll` or `long-term`).
4. Some words with apostrophes that are abbreviations are indeed separated (e.g. `We’re` to `We`, `’re`).

```
from nltk.tokenize import sent_tokenize, word_tokenize

example_string = """Natural Language Processing (NLP) is an interdisciplinary
field that empowers machines to understand, interpret, and generate human
language. Its applications span across various domains, including chatbots,
language translation, sentiment analysis, and information extraction. We’re
going to rock’n’roll in the long-term. """


print(sent_tokenize(example_string))
# [’Natural Language Processing (NLP) is an interdisciplinary field that empowers
# machines to understand, interpret, and generate human language.’, ’Its
# applications span across various domains, including chatbots, language
# translation, sentiment analysis, and information extraction.’, "We’re going to
# rock’n’roll in the long-term."]

print(word_tokenize(example_string))
# [’Natural’, ’Language’, ’Processing’, ’(’, ’NLP’, ’)’, ’is’, ’an’,
# ’interdisciplinary’, ’field’, ’that’, ’empowers’, ’machines’, ’to’,
# ’understand’, ’,’, ’interpret’, ’,’, ’and’, ’generate’, ’human’, ’language’,
# ’.’, ’Its’, ’applications’, ’span’, ’across’, ’various’, ’domains’, ’,’,
# ’including’, ’chatbots’, ’,’, ’language’, ’translation’, ’,’, ’sentiment’,
# ’analysis’, ’,’, ’and’, ’information’, ’extraction’, ’.’, ’We’, "’re",
# ’going’, ’to’, "rock’n’roll", ’in’, ’the’, ’long-term’, ’.’]

# 0.004939079284667969 seconds
```

Sometimes, there are words that are used so often that it is pointless to have them (e.g. `the`, `an`, `a`, etc.). We don’t want these words to take up space in our database, so we can use NLTK to store a list of words that we consider to be stop words. The default library of **stop words** in multiple languages can be downloaded with the following command, and we can execute them on the paragraph above. In NLTK 3.8.1, there are a total of 179 stop words in English, and you can add or remove the words in this file manually.

```
nltk.download("stopwords")
from nltk.corpus import stopwords

print(stopwords.words("english"))

# ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
# "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he',
# 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
# ...
# 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',
# "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn',
# "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

To filter the stopwords, we can loop over the words in the token list and remove it if it is a stop word.

1.3 Lemmatization and Stemming

Stemming is the process of producing morphological variants of a root word. For example, a stemming algorithm reduces the words "chocolates", "chocolatey", and "choco" to the root word "chocolate." It helps to reduce different variants into a unified word for easier retrival of data.

```
from nltk.stem import PorterStemmer

ps = PorterStemmer()
words = ["program", "programs", "programmer", "programming", "programmers"]
for w in words:
    print(w, " : ", ps.stem(w))

# program  :  program
# programs  :  program
# programmer  :  program
# programming  :  program
# programmers  :  program
```

2 Classical Learning Approaches

2.1 N Gram Model

Like a Markov model.

2.2 Naive Bayes Classification

2.3 Logistic Regression

3 Embeddings

Our goal is to create some embedding method that maps the vocabulary of words into some vector space $\mathbf{V}$. One could simply one-hot-encode all the words, but this is memory inefficient and the structure of the words are not captured. That is, we would like some associated metric d that tells us how similar two words are. At this point, it is not clear that this similarity entails, whether it'd be similar definitions (dog and canine), similar in groupings (e.g. dog and cat), or similar in context (doctor and scalpel).

There have been many attempts to create this mapping, but the most successful by far has followed the idea of **distributional semantics**. This hypothesis states that words that occur in similar contexts tend to have similar meanings.

3.1 Frequency Semantics

Let's start with the simplest distributional model based on the **co-occurence matrix**, which represents how often words co-occur.

Definition 3.1 (Term-Document Matrix)**.** Given $\mathcal{D} = \{D_1, \ldots, D_m\}$ documents and $\mathcal{V} = \{v_1, \ldots, v_n\}$ total words (tokens) in the document, the **term-document matrix** is a $n \times m$ matrix where the ijth entry represents the number of times token v_i occurred in document D_j. One can see how this will be a sparse matrix, since there may be many infrequent words that appear only once in exactly one document.

Now there are two ways that we can analyze this matrix. First, note that the columns $M_{:,i}$ are vectors that represent the words in document D_i, and the rows $M_{j,:}$ represent the frequency of a certain word v_j in each document. Therefore, we can compare documents by comparing their corresponding vectors D_i, D_j which represent the distribution of its words, and we can compare words by comparing their corresponding vectors v_i, v_j, which represent their distribution over the documents.

Another way to compare words is through a term-term matrix.

Definition 3.2 (Term-Term Matrix)**.** A **term-term matrix** is a $|V| \times |V|$ matrix that represents the number of times the row (target, center) word and the column (context) word co-occur in some context in some training corpus.

1. The context could be the document, in which case the element M_{ij} represents the number of times the two words v_i, v_j appear in the same document.
2. It is most common to use smaller contexts, generally a window around the word, e.g. ± 4 words, in which case M_{ij} represents the number of times (in some training corpus) v_j appears around ± 4 words around v_i.

To measure similarity between two words or documents, it may be natural to take some sort of distance in $\mathbb{R}^{|V|}$. However, due to the size of the documents being nonuniform, we would like to measure how parallel one vector is to another. Two very similar documents D_i, D_j, where one is twice as long as the other, would be expected to have a proportional document vector satisfying $D_i \approx 2D_j$. Therefore, the angle between the two vectors would be best representation of the similarity metric.

$$\text{cosine}(v_i, v_j) = \frac{v_i \cdot v_j}{||v_i||\,||v_j||} = \cos\theta$$

3.2 Word2Vec

Note that the word representations are sparse. There are many computational tricks to work with sparse vectors, but there tends to be lots of noise and problems with them. Rather, we look at a more powerful word representation called **embeddings**, which are short dense vectors ranging in dimension from 50 to 1000. It turns out that dense vectors work better in every NLP task than sparse vectors.

3.2.1 Doc2Vec

3.3 Implementation with gensim

The Python package gensim is used to implement the word2vec algorithm. The core concepts of the package are:

1. *Document*: Some text represented by a string.
2. *Corpus*: A collection of documents.
3. *Vector*: A mathematically convenient representation of a document.
4. *Model*: An algorithm for transforming vectors from one representation to another.

4 Recurrent Neural Networks

In a vanilla feedforward neural net architecture, we had a one to one map, where we take an input of fixed size and we map it to an output of fixed size. Perhaps we would want a one-to-many model, which takes in an image for example and outputs a variable-length description of the image. Or a many-to-many (e.g. machine translation from a sequence of words to a sequence of words) or many-to-one.

5 Transformer Models