Natural Language Processing

Muchang Bahng

July 6, 2023

Contents

We will use NLTK, PyTorch.

1 Basics

1.1 Regular Expressions

Regular expressions allow us to find patterns in strings. In Python, the `r` in front of the string stands for **raw strings**, which literally takes in special characters (e.g. escape characters). Below, we search for the string `and`.

```
import re

data = """Natural Language Processing (NLP) is an interdisciplinary field that empowers
67 machines to understand, interpret, and generate human language. Its 4 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. """

pattern = re.compile(r"and")
matches = pattern.finditer(data)

for match in matches:
    print(match)

# <re.Match object; span=(100, 103), match='and'>
# <re.Match object; span=(116, 119), match='and'>
# <re.Match object; span=(255, 258), match='and'>

print(data[100:103], data[116:119], data[255:258])
# and and and
```

There are special characters that allows us to identify other more specific patterns. We list them below.

```
.       - Any Character Except New Line
\d      - Digit (0-9)
\D      - Not a Digit (0-9)
\w      - Word Character (a-z, A-Z, 0-9, _)
\W      - Not a Word Character
\s      - Whitespace (space, tab, newline)
\S      - Not Whitespace (space, tab, newline)

\b      - Word Boundary
\B      - Not a Word Boundary
^       - Beginning of a String
$       - End of a String

[]      - Matches Characters in brackets
[^ ]    - Matches Characters NOT in brackets
|       - Either Or
( )     - Group
```

1.2 Preprocessing

1.2.1 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.

1.2.2 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
```

1.3 Minimum Edit Distance

2 Classical Learning Approaches

2.1 N Gram Model

An N-gram model basically computes a giant Markov chain.

```
data = open("/home/mbahng/Desktop/Cookbook/NLP/NLTK/alice.txt")

f = data.read().lower().replace("\n", " ")

dict = {}

tokens = word_tokenize(f)

for i in range(len(tokens)-1):
    if tokens[i] not in dict.keys():
        dict[tokens[i]] = []

    dict[tokens[i]].append(tokens[i+1])

phrase = "i"
word = "i"

for i in range(100):
    time.sleep(0.1)
    print(word, end = " ")
    word = random.choice(dict[word])

# i mentioned dinah was walking away , that lovely garden ! " \ till she looked at
# first , " as an important air , " said the earls of it had a caucus-race. " said
# the moment to pa" said the mock turtle : | \ in : \ and new pair of things|
# everything that loose network odo managed to herself that all the hatter : \ i
# wish to the best , and nobody moved off
```

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

3.1.1 Term-Document Matrix

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.

Example 3.1. Given the following three documents

D_1 : `I like deep learning.`
D_2 : `I like NLP.`
D_3 : `I enjoy flying.`

we can construct a term-document matrix as

	D_1	D_2	D_3
I	1	1	1
like	1	1	0
deep	1	0	0
learning	1	0	0
NLP	0	1	0
enjoy	0	0	1
flying	0	0	1

Table 1: Term-Document Matrix

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. This is known as the **bag-of-words model**. 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.

Upon inspection you can notice that this is exactly feature extraction. Each document in the corpus is a data point and the count of specific words are the features. We can implement this with scikit learn's feature extractor, which gives the word count of a corpus, represented as a list of strings. We first load in our corpus.

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

corpus = [
    "I like deep learning.",
    "I like NLP.",
    "I enjoy flying."
]

cvectorizer = CountVectorizer()
X = cvectorizer.fit_transform(corpus)
terms = cvectorizer.get_feature_names_out()

# List of all terms in order of document-term matrix columns.
print(terms)
# ['deep' 'enjoy' 'flying' 'learning' 'like' 'nlp']

# Document term matrix
print(X.toarray())
# [[1 0 0 1 1 0]
#  [0 0 0 0 1 1]
#  [0 1 1 0 0 0]]

# Data type of matrix and shape (note this is a sparse matrix type)
print(type(X))      # <class 'scipy.sparse._csr.csr_matrix'>
print(X.shape)      # (3, 6)

documentTermMatrix = pd.DataFrame(X.toarray(),
                                  index=["Doc 1", "Doc 2", "Doc 3", ],
                                  columns=terms)

print(documentTermMatrix.to_string())

#        deep  enjoy  flying  learning  like  nlp
# Doc 1     1      0       0         1     1    0
# Doc 2     0      0       0         0     1    1
# Doc 3     0      1       1         0     0    0
```

3.1.2 Term-Term Matrix

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

Definition 3.2 (Term-Term Matrix)**.** A **term-term matrix**, or a **co-occurence 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.

Example 3.2. Given a window length of 1 (most common for window length to be 5 through 10) and a corpus of three documents:

D_1 : `I like deep learning.`
D_2 : `I like NLP.`
D_3 : `I enjoy flying.`

the co-occurence matrix (note that it should be symmetric). is

	I	like	deep	learning	NLP	enjoy	flying
I	0	2	1	0	0	1	0
like	2	0	1	0	1	0	0
deep	1	1	0	1	0	0	0
learning	0	0	1	0	0	0	0
NLP	0	1	0	0	0	0	0
enjoy	1	0	0	0	0	0	1
flying	0	0	0	0	0	1	0

Table 2: Co-occurrence Matrix

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 TF-IDF and PMI

Let us focus on the vectors representing words, where the dimensions are documents. The words v_1, v_2 are essentially defined by their frequency in a corpus of documents $D_1, \ldots, D_m$. The TF-IDF algorithm focuses on two main modifications of the frequency representation mentioned above.

1. *TF*: The raw frequency of the words may be a skewed representation of the token, since the difference between 0 and 1 occurrence in document D_i is not the same as the difference between 1000 and 1001 occurrences in another document. Unimportant stop words like "the" that occur frequently enough shouldn't have as much of an effect on the representation of the word. We can filter these stop words out like before, but depending on context, there may be other words that are a less drastic impact. The additional impact of another instance of a word should have diminishing returns.

2. *IDF*: "Special" words that occur in only a few documents should have higher weights, since they are useful for discriminating those documents from the rest of the collection. Therefore, we would want this measure to be inversely proportional to the number of documents it is in. We can define

$$\frac{N}{\text{df}_t}$$

where N is the total number of documents and df_t is the number of documents that word t is in. Due to the large number of documents in a corpus, this value is also squashed down by a logarithm.

Given these two ideas, we define the two following forms of measure.

Definition 3.3 (TF)**.** The **term frequency** is a measure of the frequency of the word t in the document d, squashed by the logarithm function (by diminishing returns) and adding 1 so that this weight is 0 when there are 0 occurences.

$$\text{tf}_{t,d} = \log_{10}\big(\text{count}(t,d) + 1\big)$$

Definition 3.4 (IDF)**.** The **inverse document frequency** is defined

$$\mathrm{idf}_t = \log_{10}\left(\frac{N}{\mathrm{df}_t}\right)$$

Definition 3.5 (tf-idf)**.** The **tf-idf** weighted value for word t in document d thus combines the two values together.

$$w_{t,d} \coloneqq \mathrm{tf}_{t,d} \cdot \mathrm{idf}_t$$

You can implement this in sklearn with the Tf-idf vectorizer, which again extracts features from a corpus. Let us use the same corpus as above. We can compare this matrix to the document-term matrix.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

corpus = [
    "I like deep learning.",
    "I like NLP.",
    "I enjoy flying."
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

tfidfMatrix = pd.DataFrame(X.toarray(),
                           index=["Doc 1", "Doc 2", "Doc 3", ],
                           columns=terms)

print(tfidfMatrix)
#             deep     enjoy    flying  learning      like       nlp
# Doc 1  0.622766  0.000000  0.000000  0.622766  0.473630  0.000000
# Doc 2  0.000000  0.000000  0.000000  0.000000  0.605349  0.795961
# Doc 3  0.000000  0.707107  0.707107  0.000000  0.000000  0.000000
```

An alternative weighting function to tf-idf is the PPMI (positive PMI), which is used for term-term matrices, when the vector dimensions correspond to words rather than documents. PPMI draws on the intuition that the best way to weigh the association between two words is to ask how much more the two words co-occur in our corpus than we would have a priori expected to appear by chance.

Definition 3.6 (PMI)**.** The **pointwise mutual information** between a target word w and context word c is defined as

$$\mathrm{PMI}(w, c) = \log_2 \frac{P(w, c)}{P(w)\, P(c)}$$

We can see that the numerator observes the joint distribution of the two words observed together, while the denominator represents their observation independently. We take the logarithm so that its range is $(-\infty, +\infty)$, where a value of greater than 0 indicates that the words co-occur more often and values less than 0 indicates that words are co-occuring less often than chance. However, negative PMI values tend to be unreliable since these probabilities can get very small, unless our corpus is enormous. Therefore, in practicality, we use the **positive PMI**, which just sets all negative values to 0.

$$\mathrm{PPMI}(w, c) = \max\left\{0, \log_2 \frac{P(w, c)}{P(w)\, P(c)}\right\}$$

Given a term-term matrix, we can organize the PPMIs in a matrix. That is, let f_{ij} be the number of times word w_i occurs in context c_j.

3.3 Latent Semantic Analysis (LSA)

Clearly, we can see that text data suffers from high dimensionality. Even after preprocessing methods such as stop word removing, tokenization, and stemming, the document-term or TF-IDF matrices are extremely high dimensional. A linear dimensionality reduction technique is to use principal component analysis. We basically take the SVD of the matrix, which reformualtes the text data of r **latent (hidden) features**. This allows us to

3.4 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.

The intuition for word2vec is that instead of counting how often each word w occurs near, say *apricot*, we'll instead train a classifier on a binary prediction task: "Is word w likely to show up near *apricot*?" We don't actually care about this prediction task. Intead, we will take the learned classifier weights as the word embeddings. Furthermore, we can just use running text as implicitly supervised training data. A context word c that occurs near *apricot* acts as a sample of "yes" to the question above.

Imagine a sentence like the following, with **target word** *apricot*, with a windows of ± 2 **context words**.

```
... lemon, a [ tablespoon of apricot jam, a ] pinch ...
```

Now our goal is to train a classifier where given the pair $\mathbf{x} = (\mathbf{w}, \mathbf{c})$, our output must be $y \in \{0, 1\}$, where 0 means not existing and 1 means within. This means that $y \mid \mathbf{x} \sim \text{Bernoulli}(\mu)$, where μ is dependent on $(\mathbf{w}, \mathbf{c})$. This probability should be higher the more similar $\mathbf{w}$ and $\mathbf{c}$ are. We have already established that the similarity metric can be effectively represented by the dot product, we have

$$\mu = \mu(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{c})}$$

3.4.1 Skip Gram with Negative Sampling

Let us formalize this concept a bit. Let us one-hot encode every word in our corpus to the set

$$\mathcal{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_{|\mathcal{V}|}\}$$

Now, what we wish to do is to embed all these vectors in some $\mathbb{R}^E$, where E is a hyperparameter that we select and preferably $E < |\mathcal{V}|$. The embeddings $\mathbf{w}_i$ for every $\mathbf{v}_i = \mathbf{e}_i$ can be simply stored as the columns of a $E \times |V|$ matrix.

$$\begin{pmatrix} | & \ldots & | \\ \mathbf{w}_1 & \ldots & \mathbf{w}_{|\mathcal{V}|} \\ | & \ldots & | \end{pmatrix} \begin{pmatrix} | \\ \mathbf{e}_i \\ | \end{pmatrix}$$

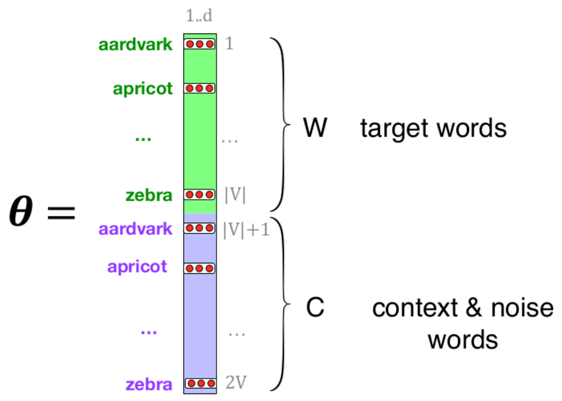
Let us denote this matrix $\boldsymbol{\theta}_{\mathbf{w}}$. Therefore, our model given the one-hot encoded target and context words $\mathbf{w}, \mathbf{c}$ is

$$\mu_{\boldsymbol{\theta}_w}(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp[-(\boldsymbol{\theta}_w \mathbf{w}) \cdot (\boldsymbol{\theta}_w \mathbf{c})]}$$

Now it turns out that the computations become easier if we have two embedding matrices, $\boldsymbol{\theta}_w$ for the target word $\mathbf{w}$ and $\boldsymbol{\theta}_c$ for the context word $\mathbf{c}$, so our model really is

$$\mu_{\boldsymbol{\theta}}(\mathbf{w}, \mathbf{c}) = \frac{1}{1 + \exp[-(\boldsymbol{\theta}_w \mathbf{w}) \cdot (\boldsymbol{\theta}_c \mathbf{c})]}$$

where $\boldsymbol{\theta} = (\boldsymbol{\theta}_w, \boldsymbol{\theta}_c)$. We can imagine these two matrices sort of "stacked" on top of each other.



Now given whatever sequence of $\pm c$ context words around a target, we will make the simplifying assumption that these words occur independently, and so the probability that a sequence of words $\mathbf{c}_{1:L}$ occur with $\mathbf{w}$ is

$$\mu_{\boldsymbol{\theta}}(\mathbf{w}, \mathbf{c}_{1:L}) = \prod_{i=1}^{L} \mu_{\boldsymbol{\theta}}(\mathbf{w}, \mathbf{c}_i)$$

Now that we have our probabilistic model, we show how to generate our relevant training data. Consider the phrase

`... lemon, a [ tablespoon of apricot jam, a ] pinch ...`

with `apricot` as our target word. Just from this, we can generate 4 positive samples

$$\begin{aligned}
\mathbf{x}^{(1)}, y^{(1)} &= (\text{apricot}, \text{tablespoon}), 1 \\
\mathbf{x}^{(2)}, y^{(2)} &= (\text{apricot}, \text{of}), 1 \\
\mathbf{x}^{(3)}, y^{(3)} &= (\text{apricot}, \text{jam}), 1 \\
\mathbf{x}^{(4)}, y^{(4)} &= (\text{apricot}, \text{a}), 1
\end{aligned}$$

where the words above would be in their one-hot encoded form. For training a binary classifier we need to have negative samples, too. Skip gram with negative sampling in fact uses more negative samples than positive ones (with the ration between them set by a parameter R). So for every positive sample, we pick a random noise word from the entire vocabulary, constrained not to be the target word $\mathbf{w}$. For example, if $R = 2$, then we would have 8 negative samples as such:

$$\begin{aligned}
\mathbf{x}^{(5)}, y^{(5)} &= (\text{apricot}, \text{aardvark}), 0 \\
\mathbf{x}^{(6)}, y^{(6)} &= (\text{apricot}, \text{my}), 0 \\
\mathbf{x}^{(7)}, y^{(7)} &= (\text{apricot}, \text{where}), 0 \\
\mathbf{x}^{(8)}, y^{(8)} &= (\text{apricot}, \text{coaxial}), 0 \\
\mathbf{x}^{(9)}, y^{(9)} &= (\text{apricot}, \text{seven}), 0 \\
\mathbf{x}^{(10)}, y^{(10)} &= (\text{apricot}, \text{forever}), 0 \\
\mathbf{x}^{(11)}, y^{(11)} &= (\text{apricot}, \text{dear}), 0 \\
\mathbf{x}^{(12)}, y^{(12)} &= (\text{apricot}, \text{if}), 0
\end{aligned}$$

But these negative samples are not chosen randomly or according to counts. Rather, we sample them according to their **weighted unigram frequency**, usually of form

$$p_\alpha(c) = \frac{\text{count}(c)^\alpha}{\sum_v \text{count}(v)^\alpha}$$

and setting $\alpha = 3/4$ gives the best results in practice, since it gives rare noise words slightly higher probability so most of the probability measure wouldn't be dominated by stop words like "the".

To train the model, we should compute the likelihood of the entire training data. Given that we have our training data of form $(\mathbf{x}^{(n)}, y^{(n)}) = (\mathbf{w}^{(n)}, \mathbf{c}^{(n)}, y^{(n)})$ for $n = 1, \ldots, N$, our likelihood for the entire dataset is

$$L(\boldsymbol{\theta}) = \prod_{n=1}^{N} \mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})^{y^{(n)}} \big(1 - \mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})\big)^{1-y^{(n)}}$$

where we can take the negative log and simplify further, but this is the extent that we will go for now.

$$-\ell(\boldsymbol{\theta}) = -\sum_{n=1}^{N} y^{(n)} \log\big(\mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})\big) + (1 - y^{(n)}) \log\big(1 - \mu_{\boldsymbol{\theta}}(\mathbf{w}^{(n)}, \mathbf{c}^{(n)})\big)$$

Obviously we can use optimization techniques to minimize this w.r.t. $\boldsymbol{\theta}$.

3.4.2 Continuous Bag of Words (CBOW)

3.4.3 Doc2Vec

3.5 Global Vectors (GloVe)

3.6 Introduction to 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 Sequence Labeling

5 Language Models with MLPs

6 Language Models with LSTMs

7 Language Models with Transformers