Gensim – Vectorizing Text and Transformations and n-grams

In the previous lab, we've seen how we clean our data before starting processing it. In this lab we'll introduce the data structures largely used in ***text analysis*** involving ***machine learning*** techniques – **vectors.**

This means that we are still in the domain of ***preprocessing*** and getting our data ready for further machine learning analysis.

Gensim library:

We're now moving toward the machine learning part of text analysis - this means that we will now start playing a little less with words and a little more with **numbers**. Even when we used **spaCy**, the **POS-tagging** and **NER-tagging,** for example, was done through statistical models - but the inner workings were largely hidden.

For Gensim however, we're expected to pass **vectors** as **inputs** to the IR algorithms (such as LDA or LSI). This means that we have to **represent** what was previously a **string** as a **vector** - and these kind of representations or models are called **Vector Space Model.**

Vectors are a way of projecting words onto a mathematical space while preserving the information provided by these words. Machine learning algorithms use these vectors to make predictions.

Let's have a reminder of some well-known vector representations.

Bag-of-words

The bag-of-words model is arguably the most **straightforward** form of representing a sentence as a vector.

S1:"The dog sat by the mat."

S2:"The cat loves the dog."

If we follow the same **preprocessing** steps we did in the previous tutorial, we will end up with the following sentences:

S1:"dog sat mat."

S2:"cat love dog."

If we want to represent this as a vector, we would need to first construct our **vocabulary**, which would be the **unique words** found in the sentences. Our **vocabulary vector** is now as follows:

Vocab = ['dog', 'sat', 'mat', 'love', 'cat']

The **bag-of-words model** involves using word frequencies to construct our vectors.

S1: [1, 1, 1, 0, 0]

S2: [1, 0, 0, 1, 1]

If the first sentence has 2 occurrences of the word dog, it would be represented as:

S1: [2, 1, 1, 0, 0]

The bag-of-words model is an order less document representation - only the counts of the words matter.

Hence, it can be used is in spam filtering - emails that are marked as spam are likely to contain spam-related words, such as *buy*, *money*, and *stock*. By converting the text in emails into a bag of words models, we can use **Bayesian probability** to determine if it is more likely for a mail to be in the spam folder or not.

TF-IDF

TF-IDF is short for term frequency-inverse document frequency. Largely used in **search engines** to find **relevant documents** based on a query, it is a rather intuitive approach to converting our sentences into vectors.

TF(t) = (number of times term t appears in a document) / (total number of terms in the document)

IDF(t) = log_e (total number of documents / number of documents with term t in it)

TF-IDF is simply the product of these two factors - TF and IDF. Together it encapsulates more information into the vector representation, instead of just using the count of the words like in the bag-of-words vector representation.

Vector transformations in Gensim

Install genism:

pip install gensim

We will now perform these vector transformations with Gensim using a corpus. A corpus is set of documents, in our case each document will be a sentence.

```
import spacy
nlp = spacy.load("en_core_web_sm")
from gensim import corpora
documents = [u"Football club Arsenal defeat local rivals this weekend.",
u"Weekend football frenzy takes over London.",
u"Bank open for takeover bids after losing millions.",
u"London football clubs bid to move to Wembley stadium.",
u"Arsenal bid 50 million pounds for striker Kane.",
u"Financial troubles result in loss of millions for bank.",
u"Western bank files for bankruptcy after financial losses.",
u"London football club is taken over by oil millionaire from Russia.",
u"Banking on finances not working for Russia."]
texts = []
for document in documents:
    text = []
    doc = nlp(document)
    for w in doc:
        if not w.is_stop and not w.is_punct and not w.like_num:
            text.append(w.lemma_)
    texts.append(text)
```

```
print(texts)
```

Let's start by whipping up a bag-of-words representation for our mini-corpus. Gensim allows us to do this very conveniently through its dictionary class.

```
dictionary = corpora.Dictionary(texts)
print(dictionary.token2id)
```

There are 32 unique words in our corpus, all of which are represented in our dictionary with each word being assigned an index value.

A reminder: you might see different numbers in your list, this is because each time you create a dictionary, different mappings will occur.

We will be using the **doc2bow** method, which, as the name suggests, helps convert our document to bag-of-words.

```
corpus = [dictionary.doc2bow(text) for text in texts]
```

This produces a list of lists, where each individual list represents a documents bag-of-words representation. Unlike the example we demonstrated, where an absence of a word was a 0, we use tuples that represent **(word_id, word_count).**

We can also notice in this case each document has not greater than one count of each word - in smaller corpuses, this tends to happen.

Our corpus is assembled, and we are ready to work some machine learning/information retrieval on them whenever we would like.

Converting a bag of words representation into **TF-IDF**, for example, is also made very easy with Gensim. We first choose the **model**/representation we want from the Gensim models' directory.

```
#TF-IDF
from gensim import models
tfidf = models.TfidfModel(corpus)

for document in tfidf[corpus]:
    print(document)
```

This means that tfidf now represents a TF-IDF table **trained** on our corpus.

Note that in case of TFIDF, the training consists simply of going through the supplied corpus once and computing document frequencies of all its features. Training other models, such as latent semantic analysis or latent dirichlet allocation, is much more involved and, consequently, takes much more time.

N-grams and some more preprocessing

Context can be very important in text analysis. We sometimes lose this context in vector representations, knowing only the count of each word. **N-grams**, and in particular, **bi-grams** are going to help us solve this problem, to some extent.

An **n-gram** is a **contiguous** sequence of n **items** in the text. In our case, the words represent the item, but depending on the use case, it could be even letters, syllables, or sometimes in the case of speech, phonemes.

A bi-gram is when **n = 2**. We're trying to find pairs of words that are more likely to appear around each other. For example, *New York* or *Machine Learning* could be two possible pairs of words created by bi-grams. Based on the **training data** (usually the corpus), we identify that it is with **high probability** that the word York follows the word New, and that it is worth considering New York as **one identity**. New York certainly provides us with **more information** than the words New and York separately.

Gensim approaches bigrams by simply combining the two high probability tokens with an underscore. The tokens new and york will now become new_york instead. Similar to the TF-IDF model, bigrams can be created using another Gensim model - Phrases .

```
import gensim
bigram = gensim.models.Phrases(texts)
trigram= gensim.models.Phrases(texts2) #texts with the new vocabulary
```

We now have a trained bi-gram model for our corpus. We can perform our transformation on the text the same way we used TF-IDF. We recreate our corpus like this:

```
texts = [bigram[line] for line in texts]
```

Each line will now have all possible bi-grams created. It should be noted that in our example, we will have no bi-grams or meaningless bi-grams being created.

Since by creating new phrases we add words to our dictionary, this step must be done before we create our dictionary. We would have to run this:

```
dictionary = corpora.Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]
```

After we are done creating our bi-grams, we can create tri-grams, and other n-grams by simply running the phrases model multiple times on our corpus. Bi-grams still remains the most used n-gram model, though it is worth one's time to glance over the other uses and kinds of n-gram implementations.

Homework:

Using the corpora available on python, test the different vector representation models on your chosen corpus and print the extracted bigrams and trigrams.