# Week 10 Notebook

# Student Name: Muhammad Rakibul Islam

Date:

Class:

Notes: (Anything you'd like to share with me before I read your notebook)

This week's notebook is based on Chapter 5 from **Blueprints for Text Analysis Using Python** by Jens Albrecht, Sidharth Ramachandran, Christian Winkler. I've added cells to this week's notebook to help explain

## Vectors, Features, and Similarity

### Review: Preparing Data for Operationalization

- Tokenizing: breaking documents into units (words, characters, sentences) called tokens.
  - Multiple tokens can be grouped together into n-grams.
- Stemming:
  - Words = root + prefix, suffix
  - Root = where the core meaning is held
  - Stemming reduced words that have similar meanings but multiple forms, such as tense, plural, gerunds, etc.
- Dimension Reduction
  - Simplify the number of observations
  - Stopword reduction (removing words that don't convey "meaninig")
  - Remove words that are too frequent or too unique
  - remove other terms that are likely to confuse the counting.

### Vectors

Vectors are mathematical objects that encode length and direction (represents a position or change in a framework or space) A 1-dimensional array of numbers (components) is displayed as a distribution. When represented geometrically, vectors represent coordinates in an n-dimensional space where n is the number of dimensions (units being compared).

- In machine learning, text is represented in an array of numbers
- Natural extension of real numbers in mathematics is a tuple (pairs of two numbers)
- Vectors are useful because they are numerical representations that are spatial and therefore have norms and distances
- We use spatial properties to measure similarity
- Measuring similarity is a fundamental principle for analyzing texts with computers
- Occupying the same, or related space, represents similarities between vectors

## Feature Engineering and Syntactic Similarity

### Things to consider

While this notebook follows Chapter 5 from *Blueprints for Text Analysis with Python*, I have also made some changes. Some changes reflect updates and modifications to arguments or methods in the code, while other changes are about trying to streamline and focus on the concept of vectorizing rather than looking too closely at computing power and resource management, which is something that you have to begin to take into consideration. For example, in the second half of the assignment, we'll be working with the ABC News Headline dataset from Kaggle Rather than going into detail about how to complete the lemmatizing in batches, I truncate the dataset so that we can improve computing time.

If preparing text corpora for machine learning is something that you are interested in doing, you'll want to spend some time learning about batch processing and maybe even working with a remote high power computing environment to improve the performance of the model.

## Setup

We are going to use the set up and directory locations from the book. If working on Google Colab: copy files and install required libraries. The following cell imports a github repository for chapter 5. The files are downloaded into the /content directory. During this phase, you're also installing the package Spacy.

```
import sys, os
ON_COLAB = 'google.colab' in sys.modules

if ON_COLAB:
    GIT_ROOT = 'https://github.com/blueprints-for-text-analytics-python/blueprints-tex
    os.system(f'wget {GIT_ROOT}/ch05/setup.py')

%run -i setup.py
```

```
    You are working on Google Colab.
    Files will be downloaded to "/content".
    Downloading required files ...
    !wget -P /content https://github.com/blueprints-for-text-analytics-python/bluepr
    !wget -P /content/data/abcnews https://github.com/blueprints-for-text-analytics-
    !wget -P /content/ch05 https://github.com/blueprints-for-text-analytics-python/b

    Additional setup ...
    !pip install -r ch05/requirements.txt
    !python -m spacy download en
```

## Load Python Settings

Common imports, defaults for formatting in Matplotlib, Pandas etc.

```
%run "$BASE_DIR/settings.py"

%reload_ext autoreload
%autoreload 2
%config InlineBackend.figure_format = 'png'
```

## Data preparation

We begin by creating a very small corpus of sentences that come from the opening lines of *A Tale of Two Cities* by Charles Dickens. We're going to work on a very simplified version of vectorization. Our hope is that by working with this simplified example that it becomes easier to then extend that mental model to much larger tasks.

The following cell takes the list of sentences and then tokenizes the words in each sentence. Next, we are going through each token in the list `sentences` and we are taking each word and putting it into a list that we store as `vocabulary`. Finally, we import Pandas and we use the `enumerate` function to create a list of tuples that include each word as a string and the position of the word in the list.

```
sentences = ["It was the best of times",
             "it was the worst of times",
             "it was the age of wisdom",
             "it was the age of foolishness"]

tokenized_sentences = [[t for t in sentence.split()] for sentence in sentences]
```

```
vocabulary = list(dict.fromkeys([w for s in tokenized_sentences for w in s]))

import pandas as pd
[[w, i] for i,w in enumerate(vocabulary)]
```

```
[['It', 0],
 ['was', 1],
 ['the', 2],
 ['best', 3],
 ['of', 4],
 ['times', 5],
 ['it', 6],
 ['worst', 7],
 ['age', 8],
 ['wisdom', 9],
 ['foolishness', 10]]
```

## One-hot by hand

In the following cell, we're defining a function called onehot_encode. That function goes through the list of `tokenized_sentences` one word at a time and checks it against the vocabulary. If a word from the vocabulary exists in the tokenized sentence, then it places a 1 in a list. If the word from the vocabulary does not occur in the sentence, it places a 0.

The result is a vector of binaries: ones and zeroes that represent the sentence by whether or not words in the vocabulary are present.

```
#defining a "onehot_encode" function to return 1 or 0 to check if word from vocab exis
#note capitalization treats words differently. "It" and "it" are different

def onehot_encode(tokenized_sentence):
    return [1 if w in tokenized_sentence else 0 for w in vocabulary]

onehot = [onehot_encode(tokenized_sentence) for tokenized_sentence in tokenized_senten

for (sentence, oh) in zip(sentences, onehot):
    print("%s: %s" % (oh, sentence))
```

```
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]: It was the best of times
[0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0]: it was the worst of times
[0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0]: it was the age of wisdom
[0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]: it was the age of foolishness
```

```
# This is a sample vector created with a sentence that wasn't
# included in the vocabulary list but has some of the words from the vocabulary.
```

```
onehot_encode("the age of wisdom is the best of times".split())
```

```
[0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0]
```

```
# If none of the words in a sentence appear in the vocabulary list, then
# you will end up with a vector that has all null values.
onehot_encode("John likes to watch movies. Mary likes to watch movies too.".split())
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

## The Document-Term Matrix

The document term matrix is the vector representation of all documents adn is the most basic building block for nearly all machine learning tasks we will do this semester. Each sentence from the list of sentences is represented in a row. Each of the words in the vocabulary is represented in the columns. The following cell takes the one_hot encoding and turns it into a dataframe.

```
#creates a matric with sentences in rows & words in vocab as columns

import pandas as pd
pd.DataFrame(onehot, columns=vocabulary)
```

| | It | was | the | best | of | times | it | worst | age | wisdom | foolishness |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| **2** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| **3** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

## Calculating similarities

Calculating the similarities between documents works by calculating the number of common 1s at the corresponding positions. In one-hot encoding, this is an extremely fast operation, as it can be calculated on the bit level by ANDing the vectors and counting the number of 1s in the resulting vectors.

```
# calculate the similarities between the first 2 sentences
# the result is the number of 1s that are shared between the 2 sentences.

# calculates the number of "1"s common at each corresponding position for the first tw
```

```
sim = [onehot[0][i] & onehot[1][i] for i in range(0, len(vocabulary))]
sum(sim)
```

```
4
```

## Scalar Product or Dot Product

calculated by multiplying corresponding components of the two vectors and adding up the products. If a product can only be 1 if both factors are 1, we can calculate the number of common 1s in the vectors.

```
#corresponding components of the vectors are multipled and products are summed.
#both have to be 1 for product to be 1. 1*1=1; 1*0=0; 0*0=0

import numpy as np
np.dot(onehot[0], onehot[1])
```

```
4
```

## Out of vocabulary

```
onehot_encode("the age of wisdom is the best of times".split())
```

```
[0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0]
```

```
onehot_encode("John likes to watch movies. Mary likes movies too.".split())
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

## document term matrix

A matrix is a vector of vectors. In other words, each of the sentences is represented by a vector of 0s and 1s that represent whether or not a word in the vocabulary appears in the sentence. When you take all four vectors and put them in a list, it becomes a matrix.

```
onehot
```

```
[[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
 [0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0],
 [0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0],
 [0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]]
```

## Similarities

We can also create a matrix of the similarity scores. So, in the following matrix example, we calculate the scalar or dot product similarity of each sentence compared to each of the other sentences in the corpus.

```
import numpy as np
np.dot(onehot, np.transpose(onehot))
```

```
    array([[6, 4, 3, 3],
           [4, 6, 4, 4],
           [3, 4, 6, 5],
           [3, 4, 5, 6]])
```

## Scikit learn one-hot vectorization

We did onehot vectorization by hand, but you can also use a tool like scikit learn to do it. Scikit Learn's OneHotEncoder is designed for the specific purpose of categorizing features (it encodes the features into the data), and that's not what we're doing here. We just want to see the encoding in action, so we use the MultiLabelBinarizer (because we want to make it as complicated as possible...).

```
#using scikit learn to do onehot vectorization intead of manually doing it

from sklearn.preprocessing import MultiLabelBinarizer
lb = MultiLabelBinarizer()
lb.fit([vocabulary])
lb.transform(tokenized_sentences)
```

```
    array([[1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0],
           [0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1],
           [0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0],
           [0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0]])
```

### Note:

Vectorizing has two different methods: fit and transform. Fit *learns* the vocabulary. Transform *converts* the documents into vectors.

## Text Statistics

## These are some cool stats!

Counting words is the simplest approach to analyzing language and text. While language is fluid and dynamic, there are some predictable elements. For example, we can count on the fact that there are words that are frequently found across all documents and all language domains (even poetry). The most common words used in English documents are: the, of, to, and, in, is, for, The, that, and said. Inversely, rare words or words that appear only one time in a corpus are also very common and can comprise about 1/2 of the total words. These are *hapax legomena* (words that appear only once).

Two "laws" describe this phenomeonon in text analysis: *Zipf's law* and *Heaps' Law*.

## Bag of Words Models

Bag-of-words representations create vectors for documents that also preserve the frequency of words that appear in each document as a feature. The frequency of the words are used as part of the weighting of the model. Models like Latent Dirichlet Allocation (LDA) explicitly require a BoW approach.

## CountVectorizer

Creating vectorizers from scratch can be very time intensive, and since a similar method can be repurposed for several types of modeling, we can use the same algorithm over and over again. If we use scikit-learn, that algorithm is part of the class called CountVectorizer. The process of turning documents into vectors is also called feature extraction.

## **Note to self:

Learning to do what we did before from scratch but using scikit learn

```
#import sklearn's CountVectorizer and then rename the function as cv
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
```

```
more_sentences = sentences + ["John likes to watch movies. Mary likes movies too.",
                              "Mary also likes to watch football games."]
pd.DataFrame(more_sentences)
```

| | 0 |
|---|---|
| 0 | It was the best of times |
| 1 | it was the worst of times |
| 2 | it was the age of wisdom |
| 3 | it was the age of foolishness |
| 4 | John likes to watch movies. Mary likes movies too. |
| 5 | Mary also likes to watch football games |

```
# Use the CountVectorizer to learn the new vocabulary in more_sentences
cv.fit(more_sentences)
```

```
▾ CountVectorizer
CountVectorizer()
```

```
# then print out the whole vocabulary
print(cv.get_feature_names_out())
```

```
['age' 'also' 'best' 'foolishness' 'football' 'games' 'it' 'john' 'likes'
 'mary' 'movies' 'of' 'the' 'times' 'to' 'too' 'was' 'watch' 'wisdom'
 'worst']
```

```
# then we use transform to vectorize all the sentences in the more_sentences variable
dt = cv.transform(more_sentences)
```

```
# when we call the vairable dt, it will describe the matrix of vectors that CountVecto
# that matrix is a vector of vectors
dt
```

```
<6x20 sparse matrix of type '<class 'numpy.int64'>'
	with 38 stored elements in Compressed Sparse Row format>
```

```
# When we turn the vector of vectors (or matrix) into a dataframe,
# we can see the features for each sentence
pd.DataFrame(dt.toarray(), columns=cv.get_feature_names_out())
```

| | age | also | best | foolishness | football | games | it | john | likes | mary | movies | of |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

## Calculating Cosign Similarities

If you want to find similarities between documents, the trick is harder than just finding the 1s. The number of occurrences of each word can be greater, and that needs to be given additional weight in our calculation of similarity. We can use the angle between two vectors to measure the similarity between them. The output is limited to numbers between 0 and 1 with 0 representing no similarity and 1 representing exact similarity. Scikit-learn has a function that helps us to do this.

## **Note to self:

I was literally just thinking about this! Like the number of times a word occurs can obviously be more than one so how do we take that into account.

```
#import cosine_similarity and then use it to calculate the angle between the first an
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarity(dt[0], dt[1])
```

```
array([[0.83333333]])
```

```
# the more_sentences variable has 6 sentences (or docs), so we will need to compare e
# six sentences to each of the other sentences.

len(more_sentences)
```

```
6
```

```
# Each sentence is a row and a column. The numbers they share are their calculated
# similarity. Obviously, document 1 and document 1 are the same, so their cosine
# similarity score is 1. The added sentences have no overlap with the first 4, so
# their cosine similarity is 0.
pd.DataFrame(cosine_similarity(dt, dt))
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|

## Review

Vectorizing textual data turns strings of vocabulary into a numerical array, which also becomes a set of features that we can use to compare one document to another. One way to do this is with the Bag-of-Words approach. ***Bag-of-words vectorizing does not take the order of the vocabulary into account, but it does take the frequency that a word appears in a document and gives that term higher weight.***

In the next section, we'll work with TF-IDF vectorizing which essentially "punishes" words that appear too frequently in the corpus.

## TF/IDF

Term Frequency - Inverse Document Frequency (tf-idf) counts the number of total word occurrences in the corpus in addition to the occurrences in a single document. It uses the relationship between the frequency that a term is used in a document and compares it to the frequency that it appears in the entire collection so that words that are frequently found in both one document and in all documents is then inversely weighted overall.

The logic behind this approach is that if there are words that are used frequently in every document, then they are not likely to convey important information. Instead, important information will likely be found in uncommon words that convey something different.

Inverted document frequency creates a penalty for common words. In fact, we can arrive at a tf-idf weighting even if we start with a Bag-of-Words matrix.

```
# from the feature_extraction text package in sklearn, we import TfidfTransformer
# then we fit and transform the variable dt from above.
from sklearn.feature_extraction.text import TfidfTransformer
tfidf = TfidfTransformer()
tfidf_dt = tfidf.fit_transform(dt)
```

```
# next, we turn the matrix into a dataframe so we can see it.
pd.DataFrame(tfidf_dt.toarray(), columns=cv.get_feature_names_out())
```

| | age | also | best | foolishness | football | games | it | john | likes | mary | movies | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00 | 0.00 | 0.57 | 0.00 | 0.00 | 0.00 | 0.34 | 0.00 | 0.00 | 0.00 | 0.00 | 0 |
| **1** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.34 | 0.00 | 0.00 | 0.00 | 0.00 | 0 |
| **2** | 0.47 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.34 | 0.00 | 0.00 | 0.00 | 0.00 | 0 |

```
# We can do the same cosine similarity calculation on the tf-idf matrix
pd.DataFrame(cosine_similarity(tfidf_dt, tfidf_dt))
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **0** | 1.00 | 0.68 | 0.46 | 0.46 | 0.00 | 0.00 |
| **1** | 0.68 | 1.00 | 0.46 | 0.46 | 0.00 | 0.00 |
| **2** | 0.46 | 0.46 | 1.00 | 0.68 | 0.00 | 0.00 |
| **3** | 0.46 | 0.46 | 0.68 | 1.00 | 0.00 | 0.00 |
| **4** | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.43 |
| **5** | 0.00 | 0.00 | 0.00 | 0.00 | 0.43 | 1.00 |

```
# Let's look at a more "real world" example. The ABC News headline
# dataset are headlines from the Australian news outlet from 2013-2017.
# We read in the files as a csv, convert it to a dataframe and print the first few ent
headlines = pd.read_csv(ABCNEWS_FILE, parse_dates=["publish_date"])
headlines.head()
```

| | publish_date | headline_text |
|---|---|---|
| **0** | 2003-02-19 | aba decides against community broadcasting licence |
| **1** | 2003-02-19 | act fire witnesses must be aware of defamation |
| **2** | 2003-02-19 | a g calls for infrastructure protection summit |
| **3** | 2003-02-19 | air nz staff in aust strike for pay rise |
| **4** | 2003-02-19 | air nz strike to affect australian travellers |

```
# Let's see how many rows the dataset has.
len(headlines.index)
```

```
1103663
```

```
# That's a lot of rows! We can't possibly do this by hand.
# The following tkaes the sklearn tfidf vectorizer, learns the vocabulary, then
# builds the vectors for each headline.
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
tfidf = TfidfVectorizer()
dt = tfidf.fit_transform(headlines["headline_text"])
```

```
# If we look at a description of the new dataframe, we see that the size of the
# dataframe becomes enormous and most of the positions in the dataframe are
# null values, which makes it very sparse.
dt
```

```
<1103663x95878 sparse matrix of type '<class 'numpy.float64'>'
        with 7001357 stored elements in Compressed Sparse Row format>
```

```
dt.data.nbytes
```

```
56010856
```

```
# One of the biggest challenges we face because of the size of the dataframe
# is the amount of time that it would take to calculate similarity scores.
# The following just calculates it for the first 10,000 rows.
%%time
cosine_similarity(dt[0:10000], dt[0:10000])
```

```
CPU times: user 206 ms, sys: 1.7 s, total: 1.9 s
Wall time: 1.91 s
array([[1.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 1.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 1.        , ..., 0.        , 0.        ,
        0.        ],
       ...,
       [0.        , 0.        , 0.        , ..., 1.        , 0.16913596,
        0.16792138],
       [0.        , 0.        , 0.        , ..., 0.16913596, 1.        ,
        0.33258708],
       [0.        , 0.        , 0.        , ..., 0.16792138, 0.33258708,
        1.        ]])
```

## **Note to self:

What TF/IDF does is basically comparing the frequency that certain words appear in a document to the frequency with which it appears in the collection.

The idea is to find out which words appear frequently in all documents in the collection which means that such words aren't really important but instead the words which appear less can give better context as to what the documents are about.

## Dimension Reduction

In order to make the operation / model run more efficiently, we need to reduce the number of dimensions (vector spaces) in the dataset. We can do this by removing stopwords using a set vocabulary list, removing the most frequent and infrequently used words, and combining words by reducing them just to their word roots so that the vocabulary is smaller. That's what the next few cells will do.

### Stopwords

```
from spacy.lang.en.stop_words import STOP_WORDS as stopwords
print(len(stopwords))
tfidf = TfidfVectorizer(stop_words="english")
dt = tfidf.fit_transform(headlines["headline_text"])
dt
```

```
326
<1103663x95588 sparse matrix of type '<class 'numpy.float64'>'
        with 5616010 stored elements in Compressed Sparse Row format>
```

## min_df

```
tfidf = TfidfVectorizer(stop_words="english", min_df=2)
dt = tfidf.fit_transform(headlines["headline_text"])
dt
```

```
<1103663x58516 sparse matrix of type '<class 'numpy.float64'>'
        with 5578938 stored elements in Compressed Sparse Row format>
```

```
tfidf = TfidfVectorizer(stop_words="english", min_df=.0001)
dt = tfidf.fit_transform(headlines["headline_text"])
dt
```

```
<1103663x6767 sparse matrix of type '<class 'numpy.float64'>'
        with 4788559 stored elements in Compressed Sparse Row format>
```

## max_df

```
tfidf = TfidfVectorizer(stop_words="english", max_df=0.1)
dt = tfidf.fit_transform(headlines["headline_text"])
dt
```

```
<1103663x95588 sparse matrix of type '<class 'numpy.float64'>'
	with 5616010 stored elements in Compressed Sparse Row format>
```

```
tfidf = TfidfVectorizer(max_df=0.1)
dt = tfidf.fit_transform(headlines["headline_text"])
dt
```

Double-click (or enter) to edit

```
#the above cells are fairly straightforward. Explanation provided before.
```

## n-grams

You are not limited to just looking at the similarity between single words. You can calculate the similarities between 1, 2, and 3 words that appear together. That will create a bit more contextual information.

```
#Note for self: Definition of n-gram:
#N-grams are continuous sequences of words or symbols, or tokens in a document.
#In technical terms, they can be defined as the neighboring sequences of items in a dc

tfidf = TfidfVectorizer(stop_words="english", ngram_range=(1,2), min_df=2)
dt = tfidf.fit_transform(headlines["headline_text"])
print(dt.shape)
print(dt.data.nbytes)
tfidf = TfidfVectorizer(stop_words="english", ngram_range=(1,3), min_df=2)
dt = tfidf.fit_transform(headlines["headline_text"])
print(dt.shape)
print(dt.data.nbytes)
```

```
(1103663, 557787)
66868960
(1103663, 742391)
71782784
```

## Lemmas

```
#Note for self:
#In natural language processing, lemmatization is the text preprocessing normalizatior
#More context: https://exchange.scale.com/public/blogs/preprocessing-techniques-in-nlp
```

## **Note to self:

In natural language processing, lemmatization is the text preprocessing normalization task concerned with bringing words down to their root forms.

More context: https://exchange.scale.com/public/blogs/preprocessing-techniques-in-nlp-a-guide

```
# The following example cuts the dataset down considerably. We're only running the
# lemmatizing on the first 25 headlines. It took about 3 hours to run the entire set
# through the lemmatizer. Some of the values that follow will be incomplete, but it
# should still give you a sense of how it all works.

from tqdm.auto import tqdm
import spacy
nlp = spacy.load("en_core_web_sm")
nouns_adjectives_verbs = ["NOUN", "PROPN", "ADJ", "ADV", "VERB"]

# for i, row in tqdm(headlines.iterrows(), total=len(headlines)):
for i, row in tqdm(headlines[:24].iterrows(), total=len(headlines)):
    doc = nlp(str(row["headline_text"]))
    headlines.at[i, "lemmas"] = " ".join([token.lemma_ for token in doc])
    headlines.at[i, "nav"] = " ".join([token.lemma_ for token in doc if token.pos_ in
```

```
0%                                   24/1103663 [00:00<3:13:37, 95.00it/s]
```

```
headlines.head()
```

| | publish_date | headline_text | lemmas | nav |
|---|---|---|---|---|
| **0** | 2003-02-19 | aba decides against community broadcasting licence | aba decide against community broadcasting licence | aba decide community broadcasting licence |
| **1** | 2003-02-19 | act fire witnesses must be aware of defamation | act fire witness must be aware of defamation | act fire witness aware defamation |
| **2** | 2003-02-19 | a g calls for infrastructure protection summit | a g call for infrastructure protection summit | g call infrastructure protection summit |

```
tfidf = TfidfVectorizer(stop_words="english")
dt = tfidf.fit_transform(headlines["lemmas"].map(str))
dt
```

```
<1103663x109 sparse matrix of type '<class 'numpy.float64'>'
	with 1103760 stored elements in Compressed Sparse Row format>
```

```
tfidf = TfidfVectorizer(stop_words="english")
dt = tfidf.fit_transform(headlines["nav"].map(str))
dt
```

```
<1103663x105 sparse matrix of type '<class 'numpy.float64'>'
	with 1103756 stored elements in Compressed Sparse Row format>
```

## Remove top 10,000

We could also use other dictionaries to remove common words in English. In the following example, we use a list from Google that is very popular of the most common words. Using a much larger list like this one will decrease the size of the dataframe to a more manageable size.

```
top_10000 = pd.read_csv("https://raw.githubusercontent.com/first20hours/google-10000-e
tfidf = TfidfVectorizer(stop_words=list(set(top_10000.iloc[:,0].values)))
dt = tfidf.fit_transform(headlines["nav"].map(str))
dt
```

```
<1103663x35 sparse matrix of type '<class 'numpy.float64'>'
	with 1103676 stored elements in Compressed Sparse Row format>
```

```
tfidf = TfidfVectorizer(ngram_range=(1,2), stop_words=list(set(top_10000.iloc[:,0].val
dt = tfidf.fit_transform(headlines["nav"].map(str))
dt
```

```
<1103663x4 sparse matrix of type '<class 'numpy.float64'>'
	with 1103645 stored elements in Compressed Sparse Row format>
```

## Finding document most similar to made-up document

Using this process, we can do something similar to what we did with the one hot encoding above. We can take a sentence that is not included in the vocabulary, vectorize it, and then calculate how similar it is to sentences in the existing corpus.

## **Note to self:

Running a version of the similarity test we did in hot encoding above but now with TF/IDF instead.

```
tfidf = TfidfVectorizer(stop_words="english", min_df=2)
dt = tfidf.fit_transform(headlines["lemmas"].map(str))
dt
```

```
<1103663x14 sparse matrix of type '<class 'numpy.float64'>'
	with 1103665 stored elements in Compressed Sparse Row format>
```

```
made_up = tfidf.transform(["australia and new zealand discuss optimal apple size"])
```

```
sim = cosine_similarity(made_up, dt)
```

```
sim[0]
```

```
array([0., 0., 0., ..., 0., 0., 0.])
```

```
headlines.iloc[np.argsort(sim[0])[::-1][0:5]][["publish_date", "lemmas"]]
```

| | publish_date | lemmas |
|---|---|---|
| **9** | 2003-02-19 | australia be lock into war timetable opp |
| **10** | 2003-02-19 | australia to contribute 10 million in aid to iraq |
| **1103662** | 2017-12-31 | NaN |
| **367882** | 2008-03-04 | NaN |
| **367884** | 2008-03-04 | NaN |

## Finding most related words

We can go through the headlines now and estimate which words in the headlines are most similar to other words in headlines. In this case, we're using cosine similarity distributions to compare words that are most likely to be found in similar contexts as other words.

```
#First creating a matrix and then calculating cosine similarity
#finally top 40 pair of words (words in the headlinges most similar to other words in
```

```
tfidf_word = TfidfVectorizer(stop_words="english", min_df=1000)
dt_word = tfidf_word.fit_transform(headlines["headline_text"])
```

```
r = cosine_similarity(dt_word.T, dt_word.T)
np.fill_diagonal(r, 0)
```

```
voc = tfidf_word.get_feature_names_out()
size = r.shape[0] # quadratic
for index in np.argsort(r.flatten())[::-1][0:40]:
    a = int(index/size)
    b = index%size
    if a > b:  # avoid repetitions
        print('"%s" related to "%s"' % (voc[a], voc[b]))
```

```
"sri" related to "lanka"
"hour" related to "country"
```

```
"seekers" related to "asylum"
"springs" related to "alice"
"pleads" related to "guilty"
"hill" related to "broken"
"trump" related to "donald"
"violence" related to "domestic"
"climate" related to "change"
"driving" related to "drink"
"care" related to "aged"
"gold" related to "coast"
"royal" related to "commission"
"mental" related to "health"
"wind" related to "farm"
"flu" related to "bird"
"murray" related to "darling"
"north" related to "korea"
"hour" related to "2014"
"world" related to "cup"
```