Crystal Mosley
MSDS 453 Sec 55

First Vectorized Representation

Introduction

TF-IDF stands for *term frequency-inverse document frequency*, and this weight is used for information retrieval (IR) and text mining. TF-IDF evaluates how important a word or phrase is to a document in a collection, or corpus. The importance increases equally to the number of times a word/phrase appears in a document.

The Process

To process the text within the class corpus, we needed to create a function to handle the below actions:

1. *Split document into individual words – tokenize*
2. *Remove punctuation from each word*
3. *Remove remaining tokens that are not alphabetic*
4. *Filter out short tokens*
5. *Lowercase all words*
6. *Filter out stop words*
7. *Word stemming - stemming is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma.*

Next, we needed to process the text into lists. We took the class corpus and created multiple *for loops* to: append the doc titles to the titles list, create and empty list to store the text doc's, appended the text to the text_body list, create an empty list to store all processed doc's to a processed_text list, and then put the individual words back together to reform the body of the text.

We then utilized the *sklearn* package to call a TF-IDF vectorizer and fit the vectorizer using the final processed docs to put the whole document back together as one. This package converts a collection of raw documents to a matrix of TF-IDF features.

Lastly, we used the *gensim word2vec* package to surface similar concepts, find unrelated concepts, and compute similarity between two or more words. Once this is complete, we created a *for loop* to obtain the weights for each word and put these into a final *dataframe* to view all the word vectors.

Important or prevalent terms that I believe are good candidates for the RTV's are below from my selected two DSI's:

- Hagel
- Syria
- Iran
- Administration
- Policy
- US

RTV's (Reference Term Vectors) from prior Cohorts that are relevant

- Trump
- US
- Syria
- Iran

Looking at the titles of the documents in the class corpus, here are my estimates for the number of documents in which each term appears:

- Trump - 22
- US - 4
- Syria - 0
- Iran – 3

Experiment 1 – I used the word stemmer for this first round against the class corpus without any other changes.

Table 1: DSI 1 TF-IDF Scores (DNE = does not exist)

DSI 1 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Trump	0.15436	0.08669
US	0.03926	DNE
Syria	0	0.29415
Iran	0.0298	0

Table 2: DSI 2 TF-IDF Scores (DNE = does not exist)

DSI 2 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Trump	0.17915	0.05748

US	0.03674	DNE
Syria	0	0.21671
Iran	0.0298	0

The results of manual vs python came out very different. Python found all lower scores than my manual scoring, and "US" did not come up at all in the data matrix. I did see other variations of "Iran", "US", and "Iran" that look more like two words put together as one, such as "iranalign", "iranback", and "iranianback". This could be because of the word stemming.

Experiment 2 – I removed the word stemmer for this next round against the class corpus without any other changes.

Table 3: DSI 1 TF-IDF Scores (DNE = does not exist)

DSI 1 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Trump	0.15436	0.08675
US	0.03926	DNE
Syria	0	0.28688
Iran	0.0298	0

Table 4: DSI 2 TF-IDF Scores (DNE = does not exist)

DSI 2 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Trump	0.17915	0.05834
US	0.03674	DNE
Syria	0	0.21436
Iran	0.0298	0

The results for this round were nearly the same for the python scores, with the only differences being that the "Trump" score was a little higher for this round, while the "Syria" score was a little lower than the previous round. I was surprised to find that the same two-word phrases were still in existence, even without the word stemming.

Experiment 3 – I tried other terms that were prevalent within both of my DSI's that also had Python scores without word stemming. I found that using word stemming caused these same words to not show as their full names.

Important or prevalent terms within both DSI's:

- Policy
- Administration

- Tehran

Estimates for the number of documents within the class corpus in which each term appears:

- Policy - 14
- Administration - 21
- Tehran - 6

Table 5: DSI 1 TF-IDF Scores

DSI 1 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Policy	0.11512	0.18368
Administration	0.17255	0.07833
Tehran	0.0445	0.04255

Table 6: DSI 2 TF-IDF Scores

DSI 2 Word/Phrase	Manual TF-IDF Score	Python TF-IDF Score
Policy	0.08201	0.03431
Administration	0.11504	0.08779
Tehran	0.0445	0.19074

The results of this third round with different terms were interesting in that my manual score for DSI 1 “policy” was smaller than the python score, but the other two terms had a lower Python score than my manual. In DSI 2, my manual score for “Tehran” was smaller than Python’s score, while the other two terms were lower for Python versus my manual scoring.

Appendix A: Python Code

```
###############################################################################
### packages required to run code.  Make sure to install all required packages.
###############################################################################
import re,string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

from sklearn.feature_extraction.text import TfidfVectorizer

import pandas as pd
import os

from gensim.models import Word2Vec

###############################################################################
### Function to process documents
###############################################################################
def clean_doc(doc):
   #split document into individual words
   tokens=doc.split()
   re_punc = re.compile('[%s]' % re.escape(string.punctuation))
   # remove punctuation from each word
   tokens = [re_punc.sub('', w) for w in tokens]
   # remove remaining tokens that are not alphabetic
   tokens = [word for word in tokens if word.isalpha()]
   # filter out short tokens
   tokens = [word for word in tokens if len(word) > 4]
   #lowercase all words
   tokens = [word.lower() for word in tokens]
   # filter out stop words
   stop_words = set(stopwords.words('english'))
   tokens = [w for w in tokens if not w in stop_words]
   # word stemming
#    ps=PorterStemmer()
#    tokens=[ps.stem(word) for word in tokens]
   return tokens

###############################################################################
### Processing text into lists
###############################################################################

#set working Directory to where class corpus is saved.
os.chdir(r'C:\Users\crmo\Desktop\Class\Predict 453\For Python Code')


#read in class corpus csv into python
```

```
data=pd.read_csv('Class Corpus.csv')

#create empty list to store text documents titles
titles=[]

#for loop which appends the DSI title to the titles list
for i in range(0,len(data)):
   temp_text=data['DSI_Title'].iloc[i]
   titles.append(temp_text)

#create empty list to store text documents
text_body=[]

#for loop which appends the text to the text_body list
for i in range(0,len(data)):
   temp_text=data['Text'].iloc[i]
   text_body.append(temp_text)

#Note: the text_body is the unprocessed list of documents read directly form
#the csv.

#empty list to store processed documents
processed_text=[]
#for loop to process the text to the processed_text list
for i in text_body:
   text=clean_doc(i)
   processed_text.append(text)

#Note: the processed_text is the PROCESSED list of documents read directly form
#the csv.  Note the list of words is separated by commas.


#stitch back together individual words to reform body of text
final_processed_text=[]

for i in processed_text:
   temp_DSI=i[0]
   for k in range(1,len(i)):
      temp_DSI=temp_DSI+' '+i[k]
   final_processed_text.append(temp_DSI)

#Note: We stitched the processed text together so the TFIDF vectorizer can work.
#Final section of code has 3 lists used.  2 of which are used for further processing.
#(1) text_body - unused, (2) processed_text (used in W2V),
#(3) final_processed_text (used in TFIDF), and (4) DSI titles (used in TFIDF Matrix)

##############################################################################
### Sklearn TFIDF
```

```
##############################################################################

#Call Tfidf Vectorizer
Tfidf=TfidfVectorizer()

#fit the vectorizer using final processed documents.  The vectorizer requires the
#stiched back together document.

X=Tfidf.fit_transform(final_processed_text)

#creating datafram from TFIDF Matrix
matrix=pd.DataFrame(X.toarray(), columns=Tfidf.get_feature_names(), index=titles)

#can export matrix to csv and explore further if necessary

##############################################################################
### Gensim Word2vec
##############################################################################

#word to vec
model = Word2Vec(processed_text, size=100, window=5, min_count=1, workers=4)

#join all processed DSI words into single list
processed_text_w2v=[]
for i in processed_text:
   for k in i:
      processed_text_w2v.append(k)

#obtian all the unique words from DSI
w2v_words=list(set(processed_text_w2v))

#can also use the get_feature_names() from TFIDF to get the list of words
#w2v_words=Tfidf.get_feature_names()

#empty dictionary to store words with vectors
w2v_vectors={}

#for loop to obtain weights for each word
for i in w2v_words:
   temp_vec=model.wv[i]
   w2v_vectors[i]=temp_vec

#create a final dataframe to view word vectors
w2v_df=pd.DataFrame(w2v_vectors).transpose()
```