BASIC PREPROCESSING AND FINDING SIMILARITIES BETWEEN TEXTS

Introduction

This report has been prepared as the mandatory assignment of PGR210 Machine Learning and Natural Language Processing course. There are five questions in this report and the first three of them basically have the same preprocessing steps. These preprocessing steps are tokenization, change to lower case, and removing white spaces, stop words and symbols. At the second step bag-of-word (bow) representations were created along with time frequencies for tokens. At the last step similarity analysis was used find similarities between tokens.

Data

The data of this project is three small texts. These texts were uploaded to a jupyter notebook environment first as raw strings. Three raw texts were generated as raw_tromso, raw_oslo, and raw_nlp:

```
# Creating raw strings for Tromso text
raw_tromso = "Tromsø is a beautiful city between FJORDS, ISLANDS AND MOUNTAINS, "\
"with a visible past, a fascinating history, a lively, colourful city centre, "\
"an inclusive nightlife and numerous attractions. Use the city as a base to "\
"foray into Arctic wilderness chasing Midnight Sun and Northern Lights. 01. 02."
```

Data Preprocessing

The first step of text analysis is tokenization. Tokenization is a fundamental of natural language processing (nlp) and it is described separating texts into smaller pieces called tokens. These tokens can be either words, numbers, characters, or other types of word forms but in our study, we prefer to eliminate everything except words and numbers. Basic preprocessing steps were applied to the texts along with token creation by ‘token_pro’ function. In this function ‘word_tokenize’ is the method which turns single words to tokens. For creating clean token for nlp, there are some other steps to be completed as it is mentioned before. Upper cases were converted to lower case by ‘.lowercase()’ method. White spaces were erased from the texts by ‘.strip()’ method. First stop words of English language were downloaded from nltk package and the words in the texts which are matches with English stops words were eliminated by ‘[word for word in tokens if word not in stop_words]’ part of token_pro function. At the last part of the function all punctuations, characters

which are not letters or numbers were eliminated from the texts by '[word for word in tokens if word.isalnum()]' code.

```
# Creating a general function to generate tokens
def token_pro(proc_str):
    tokens = word_tokenize(proc_str) # Tokenization
    tokens = [token.lower() for token in tokens] # Converting token to lowercase
    tokens = [token.strip() for token in tokens] # Removing white spaces
    stop_words = nltk.corpus.stopwords.words('english') # Creating stop words
    tokens = [word for word in tokens if word not in stop_words] # Removing stop wor
    #Removing punctuations and the other characters which are not a letter or a numb
    tokens = [word for word in tokens if word.isalnum()]

    return tokens
```

After the creation of that general 'token_pro' function, three different tokens were created by this function and tokens were named as token_tromso, token_oslo, and token_nlp:

token_tromso = token_pro(raw_tromso)

The resulting three tokens:

['tromsø', 'beautiful', 'city', 'fjords', 'islands', 'mountains', 'visible', 'past', 'fascinating', 'history', 'lively', 'colourful', 'city', 'centre', 'inclusive', 'nightlife', 'numerous', 'attractions', 'use', 'city', 'base', 'foray', 'arctic', 'wilderness', 'chasing', 'midnight', 'sun', 'northern', 'lights', '01', '02']
['oslo', 'considered', 'global', 'city', 'major', 'norwegian', 'hub', 'trading', 'shipping', 'banking', 'location', 'oslo', 'oslo', 'positioned', 'northernmost', 'end', 'oslofjord', 'occupies', 'around', '40', 'big', 'small', 'islands', 'within', 'limits', 'climate', 'region', 'temperate', 'humid']
['aim', 'course', 'introduce', 'students', 'concepts', 'techniques', 'natural', 'languages', 'processing', 'analysis', 'unstructured', 'information', 'analysis', 'management', 'better', 'decisionmaking', 'deriving', 'valuable', 'insights', 'enterprise', 'content', 'regardless', 'source', 'format', 'course', 'provides', 'deep', 'rich', 'knowledge', 'text', 'analysis', 'techniques', 'applications', 'including', 'sentiment', 'analysis', 'opinion', 'mining', 'information', 'access', 'text', 'mining', 'document', 'classification', 'topic', 'extraction', 'techniques', 'applications', 'using', 'data', 'cases']

Bag of word (bow) representations and Time Frequency (tf)

Texts cannot be implemented directly into machine learning models. Bow is one of the necessary preprocessing steps in nlp which turns regular texts to 'bag-of-words'. Bow has a dictionary like structure, and it has basically two information: (a) words in the raw text and (b) frequency of occurrence of words. Bows are created in three steps in this study. First a 'corpus', the combination of all raw texts, was created.

```
# Creating corpus
corpus = [raw_tromso, raw_oslo, raw_nlp]
```

A bow generation function was created by using the newly created corpus. The ‘Counter’ container which holds words as a dictionary key is the main part of this function.

```
# Creating general bow function
tokens = token_pro(''.join(corpus))
def gen_bow(tokens):
    bow = Counter(tokens)
    return bow
```

At the last step three bows were created by the general ‘gen_bow’ function.

```
# Generating bows for each token
bow_tromso = gen_bow(token_tromso)
bow_oslo = gen_bow(token_oslo)
bow_nlp = gen_bow(token_nlp)
```

The resulting bow contains information about words for each bow and their frequency in the corpus:

bow_tromso: Counter({'city': 3, 'tromsø': 1, 'beautiful': 1, 'fjords': 1, 'islands': 1, 'mountains': 1, 'visible': 1, 'past': 1, 'fascinating': 1, 'history': 1, 'lively': 1, 'colourful': 1, 'centre': 1, 'inclusive': 1, 'nightlife': 1, 'numerous': 1, 'attractions': 1, 'use': 1, 'base': 1, 'foray': 1, 'arctic': 1, 'wilderness': 1, 'chasing': 1, 'midnight': 1, 'sun': 1, 'northern': 1, 'lights': 1, '01': 1, '02': 1})
bow_oslo: Counter({'oslo': 3, 'considered': 1, 'global': 1, 'city': 1, 'major': 1, 'norwegian': 1, 'hub': 1, 'trading': 1, 'shipping': 1, 'banking': 1, 'location': 1, 'positioned': 1, 'northernmost': 1, 'end': 1, 'oslofjord': 1, 'occupies': 1, 'around': 1, '40': 1, 'big': 1, 'small': 1, 'islands': 1, 'within': 1, 'limits': 1, 'climate': 1, 'region': 1, 'temperate': 1, 'humid': 1})
bow_nlp: Counter({'analysis': 4, 'techniques': 3, 'course': 2, 'information': 2, 'text': 2, 'applications': 2, 'mining': 2, 'aim': 1, 'introduce': 1, 'students': 1, 'concepts': 1, 'natural': 1, 'languages': 1, 'processing': 1, 'unstructured': 1, 'management': 1, 'better': 1, 'decisionmaking': 1, 'deriving': 1, 'valuable': 1, 'insights': 1, 'enterprise': 1, 'content': 1, 'regardless': 1, 'source': 1, 'format': 1, 'provides': 1, 'deep': 1, 'rich': 1, 'knowledge': 1, 'including': 1, 'sentiment': 1, 'opinion': 1, 'access': 1, 'document': 1, 'classification': 1, 'topic': 1, 'extraction': 1, 'using': 1, 'data': 1, 'cases': 1})

A time frequency vector (tf) is also created based upon bows.

```python
# Term frequency vector generation
def print_bow_corpus(bow,query_str):
    times_query_appears = Counter(query_str)
    tf_vec = []
    for word in bow:
        if word in query_str:
            tf = times_query_appears[word]
        else:
            tf=0
        tf_vec.append(tf)

    return tf_vec
```

Time frequency (tf) vectors shows number of times a word appears in a bow. Tf vectors were created by newly created 'print_bow_corpus' function, bows, and individual tokens.

bow = gen_bow(tokens)

But you're not using 'relative TF' (divide count by document length to adjust for variable length docs)

tf_tromso = print_bow_corpus(bow,token_tromso)

tf_oslo = print_bow_corpus(bow,token_oslo)

tf_nlp = print_bow_corpus(bow,token_nlp)

Tf vectors represent absolute numbers which a word appears in a bow and this is not very ideal. In a tf vector, it is not easy to understand weight of a word in a text. It is not the same appearing 5 times in a text with 100 words and another one with 200 words.

GOOD INSIGHT ON RAW COUNTS

tf_tromso: [1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
tf_oslo: [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
tf_nlp: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1, 1, 1, 3, 1, 1, 1, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1]

Creating TF-IDF Vectors and Discovering Similarities between Texts

The main aim of this study is to find two most similar text to the new text given in the task 5. There are two approaches in this part of this study. First distances between texts were discovered by subtracting of texts. But there is no way to subtract two texts directly. Therefore, first token, bow, and tf vector were created. At the last step the first tf vectors subtracted from the last one after conversion to numpy array:

```
dtm = sum(np.abs(np.asarray(tf_tromso)-np.asarray(tf_mlnlp)))

dom = sum(np.abs(np.asarray(tf_oslo)-np.asarray(tf_mlnlp)))

dnm = sum(np.abs(np.asarray(tf_nlp)-np.asarray(tf_mlnlp)))
```

The results show that the difference between tf_nlp and tf_mlnlp (the last text) is the minimal (56) and it can be assumed these two texts are more similar to each other.

The second and more advanced alternative of finding similarities between texts is to develop TF-IDF vector. TF-IDF is an acronym of Term Frequency – Inverse Document Frequecy (Chakravarthy, 2020) and it is a weighting factor which describes importance of features in a document (corpus) (Sanjeevi, 2017). TF-IDF is developed as a ranking metric for search engines. TF is described as: TF = (Number of times a word appears in a document) / (Number of words in the document). IDF is described as: IDF = log_e(Number of documents/ Number of documents in which the term appears). The main function of TF-IDF is to increase value of tokens which are important in a text or document (Chakravarthy, 2020).

For developing TF-IDF vector, first a new corpus which encompasses all four tokens was created.

```
new_corpus = [' '.join(token_tromso),' '.join(token_oslo),' '.join(token_nlp),' '.join(token_mlnlp)]
```

After this step a CountVectorizer matrix (cv_matrix) and a CountVectorizer array (cv_array) were created:

```
cv = CountVectorizer(min_df=0, max_df=1.)

cv_matrix = cv.fit_transform(new_corpus)

cv_array = cv_matrix.toarray()
```

A TF-IDF vector is created at the next step by 'TfidfVectorizer'. TfidfVectorizer transforms text to features and can be used as an input estimator.

```
vector = TfidfVectorizer(norm = 'l2', use_idf = True)
tf_idf = vector.fit_transform(new_corpus)
```

At the last step 'pairwise_similarity' values were calculated by multiplication of TF-IDF and transpose TF-IDF values.

```
pairwise_similarity = tf_idf * tf_idf.T
```

Pairwise similarity shows that token_nlp and token mlnlp are the most similar token with 0.2296 value. In the result, '2' (token_nlp) represents the third token (token_mlnlp) and '3' represents the fourth token. This number shows that almost 23% of texts are the same and the other similarity values are very low.

```
Similarity
  (0, 1)        0.07414700741193253
  (0, 3)        0.005976405238858923
  (0, 0)        0.9999999999999993
  (1, 0)        0.07414700741193253
  (1, 1)        0.9999999999999989
  (2, 3)        0.22965674708615336
  (2, 2)        1.0
  (3, 0)        0.005976405238858923
  (3, 2)        0.22965674708615336
  (3, 3)        1.0

Similarity Array
[[1.         0.07414701 0.         0.00597641]
 [0.07414701 1.         0.         0.        ]
 [0.         0.         1.         0.22965675]
 [0.00597641 0.         0.22965675 1.        ]]
```

Conclusions and Recommendations

TF-IDF model is a quick way to acquire information about texts and documents which allows get meaning from texts and compare documents in a mathematical way. It is very easy to implement the code and the method is not computationally expensive. Besides the advantages it also has some disadvantages. It computes document (text) similarity based on only word-count (bag-of-words) and it does not take into account word's position in the text, semantics and co-occurrences (Ahuja,

2020). TF-IDF method based on word count gives enough information about text similarity which is not always the case. It may be slow to implement this method for large text.

There are some alternatives for TF-IDF method which provide improvements in text analysis. I suggest some methods which take into account not only word count but also context and semantics. One of them is word2vec algorithm which uses neural network model to learn associations in a large corpus (Wikipedia 2021). It can detect synonyms and suggest words for partial sentences. It uses word-embedding very efficiently. Therefore, it is used very efficiently by companies like Ali express, AIR BNB, Spotify, etc (Alammar, 2019). Another efficient alternative NLP model to TF-IDF is BERT (Bidirectional Encoder Representations from Transformers). This method is released in 2018 and stirred NLP community by its highly accurate results in different NLP tasks (Horev, 2018). BERT uses a transformer, an attention mechanism, that learns contextual relationships between words. The transformer has two mechanisms – an encoder which reads texts and a decoder which produces predictions for the text. Directional models read text sequentially, from left to right or right to left. But BERT reads text in both directions; left-to-right and right-to-left. This specialty allows BERT to learn better from words surroundings, from both left and right. One of the most used methods in both industrial and research purposes is ELMo. ELMo models both complex characteristics of word use (syntax, semantics, etc.) and how words are used in linguistic contexts. The internal states of bidirectional language model (biLM), pre-trained in corpus, is used to create word vectors. These vectors can be added to existing model and improve results significantly in different NLP works such as question answering or sentiment analysis (Peters et al. 2018).

References

Jay Alammar (March 27, 2019). The illustrated Word2vec. https://jalammar.github.io/illustrated-word2vec/

Madhu Sanjeevi (2017). Chapter 9.1: NLP - Word vectors. https://medium.com/deep-math-machine-learning-ai/chapter-9-1-nlp-word-vectors-d51bff9628c1.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer. Deep contextualized word representations. NAACL 2018.

Pallavi Ahuja, (2020). How Good (or Bad) is Traditional TF-IDF Text Mining Technique? https://medium.com/analytics-vidhya/how-good-or-bad-is-traditional-tf-idf-text-mining-technique-304aec920009

Rani Horev (November 10, 2018). Best explained: State of the art language model for NLP. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

Srinivas Chakravarthy (2020). Simple Word Embedding for Natural Language Processing. https://towardsdatascience.com/simple-word-embedding-for-natural-language-processing-5484eeb05c06.

Wikipedia (2021). Word2vec. https://en.wikipedia.org/wiki/Word2vec. Accessed October 29, 2021.

Good stuff. This is the type of analysis report we are looking for.