Natural Language processing

TP 5 – Text Vectorization Using Traditional Methods

- Numerous methods exist for converting text to numeric representations.
- Linguists traditionally manually annotate text using self-defined linguistic properties, which are easily converted into numeric values, facilitating text vectorization.
- In statistical language processing, the focus is on automating text vectorization to minimize manual effort in annotation.
- This tutorial explores the widely-used bag-of-words method for text vectorization in machine learning NLP.

Exercice 1 – Frequency Based embeddings

1. Pre treatment

1. Import dependencies

```
# import warnings
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib
import matplotlib.pyplot as plt

## Default Style Settings
matplotlib.rcParams['figure.dpi'] = 150
pd.options.display.max_colwidth = 200
```

2. Sample Corpus of Text Documents

```
corpus = [
    'The sky is blue and beautiful.', 'Love this blue and beautiful sky!',
    'The quick brown fox jumps over the lazy dog.',
    "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
    'I love green eggs, ham, sausages and bacon!',
    'The brown fox is quick and the blue dog is lazy!',
    'The sky is very blue and the sky is very beautiful today',
    'The dog is lazy but the brown fox is quick!'
]
labels = [
    'weather', 'weather', 'animals', 'food', 'food', 'animals', 'weather',
    'animals'
]

corpus = np.array(corpus) # np.array better than list
corpus_df = pd.DataFrame({'Document': corpus, 'Category': labels})
corpus_df
```

Text :

corpus = [
'The sky is blue and beautiful.', 'Love this blue and beautiful sky!',
'The quick brown fox jumps over the lazy dog.',
"A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
'I love green eggs, ham, sausages and bacon!',
'The brown fox is quick and the blue dog is lazy!',
'The sky is very blue and the sky is very beautiful today',
'The dog is lazy but the brown fox is quick!'

```
]
labels = [
  'weather', 'weather', 'animals', 'food', 'food', 'animals', 'weather',
  'animals'
]
```

3. Text Preprocessing

```
wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')


def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I | re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc


normalize_corpus = np.vectorize(normalize_document)
```

4. Print results

```
norm_corpus = normalize_corpus(corpus)
print(corpus)
print("="*50)
print(norm_corpus)
```

2. Frequency-based embeddings

5. BOW modelling

```
from sklearn.feature_extraction.text import CountVectorizer
# get bag of words features in sparse format
cv = CountVectorizer(min_df=0., max_df=1.)
cv_matrix = cv.fit_transform(norm_corpus)
cv_matrix
```

6. View dense representation

```
cv_matrix = cv_matrix.toarray()
cv_matrix
```

7. Get unique words in the corpus

```
vocab = cv.get_feature_names_out()
# show document feature vectors
pd.DataFrame(cv_matrix, columns=vocab)
```

In `CountVectorizer()`, we can utilize its parameters:

- `max_df`: When building the vocabulary, the vectorizer will ignore terms that have a **document frequency** strictly higher than the given threshold (corpus-specific stop words). `float` = the parameter represents a proportion of documents; `integer` = absolute counts.

- `min_df`: When building the vocabulary, the vectorizer will ignore terms that have a **document frequency** strictly lower than the given threshold. `float` = the parameter represents a proportion of documents; `integer` = absolute counts.
- `max_features` : Build a vocabulary that only consider the top `max_features` ordered by term frequency across the corpus.
- `ngram_range` : The lower and upper boundary of the range of n-values for different word n-grams. `tuple` (min_n, max_n), default=(1, 1).
- `token_pattern`: Regular expression denoting what constitutes a "token" in vocabulary. The default regexp select tokens of 2 or more alphanumeric characters (Note: **punctuation** is completely ignored and always treated as a token separator).

8. N-gram BOW text representation

```
# you can set the n-gram range to 1,2 to get unigrams as well as bigrams
bv = CountVectorizer(ngram_range=(2, 2))
bv_matrix = bv.fit_transform(norm_corpus)

bv_matrix = bv_matrix.toarray()
vocab = bv.get_feature_names_out()
pd.DataFrame(bv_matrix, columns=vocab)
```

9. Import TF-IDF transformer implementation

```
from sklearn.feature_extraction.text import TfidfTransformer

tt = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True)
tt_matrix = tt.fit_transform(cv_matrix)

tt_matrix = tt_matrix.toarray()
vocab = cv.get_feature_names_out()
pd.DataFrame(np.round(tt_matrix, 2), columns=vocab)
```

10. Import TF-IDF vectorizer implementation

```
from sklearn.feature_extraction.text import TfidfVectorizer

tv = TfidfVectorizer(min_df=0.,
                     max_df=1.,
                     norm='l2',
                     use_idf=True,
                     smooth_idf=True)
tv_matrix = tv.fit_transform(norm_corpus)
tv_matrix = tv_matrix.toarray()

vocab = tv.get_feature_names_out()
pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```

11. Create vocabulary dictionary of the corpus

```
# get unique words as feature names
unique_words = list(
    set([word for doc in [doc.split() for doc in norm_corpus]
         for word in doc]))

# default dict
def_feature_dict = {w: 0 for w in unique_words}

print('Feature Names:', unique_words)
print('Default Feature Dict:', def_feature_dict)
```

12. Create Document-Word Matrix (Bag-of-Word Frequencies)

```
from collections import Counter
# build bag of words features for each document - term frequencies
bow_features = []
for doc in norm_corpus:
    bow_feature_doc = Counter(doc.split())
    # initialize default corpus dictionary
    all_features = Counter(def_feature_dict)

    # update default dict with current doc words
    bow_feature_doc.update(all_features)

    # append cur doc dict
    bow_features.append(bow_feature_doc)

bow_features = pd.DataFrame(bow_features)
bow_features
```

13. Compute Document Frequency of Words

```
import scipy.sparse as sp
feature_names = list(bow_features.columns)

# build the document frequency matrix
df = np.diff(sp.csc_matrix(bow_features, copy=True).indptr)
# `csc_matrix()` compress `bow_features` into sparse matrix based on columns
# `csc_matrix.indices` stores the matrix value indices in each column
# `csc_matrix.indptr` stores the accumulative numbers of values from column-0 to the right-most column

df = 1 + df  # adding 1 to smoothen idf later

# show smoothened document frequencies
pd.DataFrame([df], columns=feature_names)
```

14. Create Inverse Document Frequency of Words

```
# compute inverse document frequencies for each term
total_docs = 1 + len(norm_corpus)
idf = 1.0 + np.log(float(total_docs) / df)

# show smoothened idfs
pd.DataFrame([np.round(idf, 2)], columns=feature_names)
```

15. Compute Raw TF-IDF for Each Document

```
# compute tfidf feature matrix
tf = np.array(bow_features, dtype='float64')
tfidf = tf * idf  ## `tf.shape` = (8,20), `idf.shape`=(20,)
# view raw tfidf feature matrix
pd.DataFrame(np.round(tfidf, 2), columns=feature_names)
```

16. Get L2 Norms of TF-IDF

```
from numpy.linalg import norm
# compute L2 norms
norms = norm(tfidf, axis=1)  # get the L2 forms of tfidf according to columns

# print norms for each document
print(np.round(norms, 3))
```

17. Compute Normalized TF-IDF for Each Document

```
# compute normalized tfidf
norm_tfidf = tfidf / norms[:, None]

# show final tfidf feature matrix
pd.DataFrame(np.round(norm_tfidf, 2), columns=feature_names)
```

18. Testing

```
new_doc = 'the sky is green today'

pd.DataFrame(np.round(tv.transform([new_doc]).toarray(), 2),
             columns=tv.get_feature_names_out())
```

Exercise 2 – Document similarity

19. Import libraries

```
from sklearn.metrics.pairwise import manhattan_distances, euclidean_distances, cosine_similarity
```

20. Pairwise similarity computation

```
similarity_doc_matrix = cosine_similarity(tv_matrix)
similarity_doc_df = pd.DataFrame(similarity_doc_matrix)
similarity_doc_df
```

21. Clustering documents using similarity features

```
from scipy.cluster.hierarchy import dendrogram, linkage

Z = linkage(similarity_doc_matrix, 'ward')
```

22. Convert hierarchical cluster into a flat cluster structure

```
from scipy.cluster.hierarchy import fcluster
max_dist = 1.0

cluster_labels = fcluster(Z, max_dist, criterion='distance')
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```

23. Clustering Words Using Similarity Features

```
similarity_term_matrix = cosine_similarity(np.transpose(tv_matrix))
similarity_term_df = pd.DataFrame(similarity_term_matrix,
                                  columns=feature_names,
                                  index=feature_names)
similarity_term_df
```

24. Running clustering and plotting

```
Z2 = linkage(similarity_term_matrix, 'ward')
plt.figure(figsize=(7, 4))
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data point')
plt.ylabel('Distance')
dendrogram(Z2, labels=feature_names, leaf_rotation=90)
plt.axhline(y=1.0, c='k', ls='--', lw=0.5)
```