20BCE529

Naman Thaker

Data Mining

Practical 9

Apply suitable unsupervised technique and clustered a newspaper article(Prerequisite: Knowledge of TF-IDF term frequency-inverse document frequency is required). Using suitable accuracy parameters evaluate prediction accuracy.

```
import re
import string
import pandas as pd
from functools import reduce
from math import log
```

Simple Implementation of TF-IDF without library

Simple example of TF-IDF

1. Example of corpus
2. Preprocessing and Tokenizing
3. Calculating bag of words
4. TF
5. IDF
6. TF-IDF

```
#1
corpus = """
Simple example with Cats and Mouse
Another simple example with dogs and cats
Another simple example with mouse and cheese
""".split("\n")[1:-1]
```

```
#2
l_A = corpus[0].lower().split()
l_B = corpus[1].lower().split()
l_C = corpus[2].lower().split()

print(l_A)
print(l_B)
print(l_C)
```

```
['simple', 'example', 'with', 'cats', 'and', 'mouse']
```

```
['another', 'simple', 'example', 'with', 'dogs', 'and', 'cats']
['another', 'simple', 'example', 'with', 'mouse', 'and', 'cheese']
```

```
#3
word_set = set(l_A).union(set(l_B)).union(set(l_C))
print(word_set)
```

```
{'example', 'dogs', 'cheese', 'simple', 'with', 'mouse', 'another', 'cats', 'and'}
```

```
word_dict_A = dict.fromkeys(word_set, 0)
word_dict_B = dict.fromkeys(word_set, 0)
word_dict_C = dict.fromkeys(word_set, 0)

for word in l_A:
    word_dict_A[word] += 1

for word in l_B:
    word_dict_B[word] += 1

for word in l_C:
    word_dict_C[word] += 1

pd.DataFrame([word_dict_A, word_dict_B, word_dict_C])
```

	example	dogs	cheese	simple	with	mouse	another	cats	and
0	1	0	0	1	1	1	0	1	1
1	1	1	0	1	1	0	1	1	1
2	1	0	1	1	1	1	1	0	1

#4 tf - term frequency

In the case of the term frequency $tf(t,d)$, the simplest choice is to use the raw count of a term in a string.

$$\mathrm{tf}(t,d) = \frac{n_t}{\sum_k n_k}$$

where n_t is the number of occurrences of the word t in the string, and in the denominator - the total number of words in this string.

```
def compute_tf(word_dict, l):
    tf = {}
    sum_nk = len(l)
    for word, count in word_dict.items():
        tf[word] = count/sum_nk
    return tf
```

```
tf_A = compute_tf(word_dict_A, l_A)
tf_B = compute_tf(word_dict_B, l_B)
```

```
tf_C = compute_tf(word_dict_C, l_C)
```

#5 idf - inverse document frequency

idf is a measure of how much information the word provides

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

- N: total number of strings in the corpus $N = |D|$
- $|\{d \in D : t \in d\}|$: number of strings where the term t appears (i.e., $\mathrm{tf}(t, d) \neq 0$). If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the denominator to $1 + |\{d \in D : t \in d\}|$.

```
def compute_idf(strings_list):
    n = len(strings_list)
    idf = dict.fromkeys(strings_list[0].keys(), 0)
    for l in strings_list:
        for word, count in l.items():
            if count > 0:
                idf[word] += 1

    for word, v in idf.items():
        idf[word] = log(n / float(v))
    return idf
```

```
idf = compute_idf([word_dict_A, word_dict_B, word_dict_C])
```

6 tf-idf

Then tf−idf is calculated as

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

```
def compute_tf_idf(tf, idf):
    tf_idf = dict.fromkeys(tf.keys(), 0)
    for word, v in tf.items():
        tf_idf[word] = v * idf[word]
    return tf_idf
```

```
tf_idf_A = compute_tf_idf(tf_A, idf)
tf_idf_B = compute_tf_idf(tf_B, idf)
tf_idf_C = compute_tf_idf(tf_C, idf)
```

```
pd.DataFrame([tf_idf_A, tf_idf_B, tf_idf_C])
```

	example	dogs	cheese	simple	with	mouse	another	cats	and
0	0.0	0.000000	0.000000	0.0	0.0	0.067578	0.000000	0.067578	0.0
1	0.0	0.156945	0.000000	0.0	0.0	0.000000	0.057924	0.057924	0.0

For clustering we must use tf-idf weights

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
```

Full text for clusterring

This corpus contain some news strings about Google and some strings about TF-IDF from Wikipedia. Just for example

```
all_text = """
Google and Facebook are strangling the free press to death. Democracy is the loser
Your 60-second guide to security stuff Google touted today at Next '18
A Guide to Using Android Without Selling Your Soul to Google
Review: Lenovo’s Google Smart Display is pretty and intelligent
Google Maps user spots mysterious object submerged off the coast of Greece - and no-one kn
Android is better than IOS
In information retrieval, tf-idf or TFIDF, short for term frequency–inverse document frequ
is a numerical statistic that is intended to reflect
how important a word is to a document in a collection or corpus.
It is often used as a weighting factor in searches of information retrieval
text mining, and user modeling. The tf-idf value increases proportionally
to the number of times a word appears in the document
and is offset by the frequency of the word in the corpus
""".split("\n")[1:-1]
```

Preprocessing and tokenizing

Firstly, we must bring every chars to lowercase and remove all punctuation, because it's not important for our task, but is very harmful for clustering algorithm. After that, we'll split strings to array of words.

```
def preprocessing(line):
    line = line.lower()
    line = re.sub(r"[{}]".format(string.punctuation), " ", line)
    return line
```

Now, let's calculate tf-idf for this corpus

```
tfidf_vectorizer = TfidfVectorizer(preprocessor=preprocessing)
tfidf = tfidf_vectorizer.fit_transform(all_text)
```

And train simple kmeans model with k = 2

```
kmeans = KMeans(n_clusters=2).fit(tfidf)
```

Predictions

```
lines_for_predicting = ["tf and idf is awesome!", "some androids is there", "how important
kmeans.predict(tfidf_vectorizer.transform(lines_for_predicting))
```

```
array([1, 0, 1, 1], dtype=int32)
```

✓ 0s completed at 1:23 AM