EDAN20 Language Technology
-Lab 1.

Nils Romanus

September 2022

1 Creating an Indexer

Using the regex library coupled with baseline python text operations one may create an indexer for novels written by Selma Lagerlöf.

1.1 Creating a Tokenizer

```
# Write your regex here
regex = r'\p{L}+'
# Write your code here
def tokenize(text: str) -> list:
    return re.findall( r'\p{L}+', text)
```

Listing 1: Tokenizer

By using regex the regex pattern in code snippet 1 coupled with the *findall* call one may create a tokenizer that returns a list of all the words in a text.

1.2 Extracting indices

```
# Write your code here
from typing import Dict, List
def text_to_idx(words: List[str], text: str) -> Dict[str, List[int
    ]]:
    # get unique words
    words = set(words)
    idxs = {}

    for word in words:
        word_delimited = r'\b' + word + r'\b'
        for match in re.finditer(word_delimited, text):
            if word not in idxs.keys():
                idxs[word] = [match.start()]
            else:
                idxs[word] += [match.start()]

    return idxs
```

Listing 2: Indexer

By using the tokenizer from code snippet 1 one may create the indexer displayed in code snippet 2. This is done by using the *finditer* regex function with the regex pattern for every word in the list of word extracted from the tokenizer. For every word its indices are generated by using the *start* function on each match. The results are subsequently stored in a map structure where the words constitutes the keys and the list of indices the corresponding values. This function can subsequently be applied to a tokenized version of an entire file, thus generating an index for the entire novel found in that file.

1.3 Creating a Master Index

```python
# write your code here
def get_master_index(file_names):
    # find full vocabulary across documents
    file_tokens = {} # map file name to tokens
    master_tokens = set()
    for file_name in file_names:
        with open("Selma/" + file_name, "r") as f:
            tokens = set(tokenize(f.read().lower()))

            master_tokens = master_tokens.union(tokens)

            # save tokens for later
            file_tokens[file_name] = tokens

    # create index for each document
    file_to_index = {} # {file_name: {token: idx}}
    for file_name in file_names:
        with open("Selma/" + file_name, 'r') as f:
            tokens = file_tokens[file_name]
            file_to_index[file_name] = text_to_idx(tokens, f.read()
    .lower())

    # create master index for each word
    _master_index = {
        token: {file_name: idx[token]
                for file_name, idx in file_to_index.items()
                if token in idx}
        for token in master_tokens
    }

    return _master_index
```

Listing 3: Master Indexer

By using the indexer on each file in the set of files in the *Selma* corpus one may create a master index covering the entire corpus. As code snippet 3 shows this is done by iterating through all files, generating all tokens for each file and calling the indexer function displayed in code snippet 2. The results are subsequently stored in a nested map structure with the token as primary with its value being a map with the file names as keys and list of indices as values.

1.4 Generating Concordances

```python
# Write your code here
def concordance(word, master_index, window) -> None:
    assert word in master_index

    for file_name, indices in master_index[word].items():
        print(file_name)

        with open("Selma/" + file_name, 'r') as f:
            text = f.read().lower().replace("\n", " ")
            for start in indices:
                print("\t", text[start - window:start + window])
```

Listing 4: Concordances

The master index can subsequently be used to generate all concordances of a word in the entire corpus. This is done by using the index of the word in the master index map and printing the slice of the text where the word occurs including the *window* number of characters before and after.

1.5 Representing Documents with *tf-idf*

```
# Write your code here
from collections import Counter
import numpy as np
import itertools

def get_tfidf(file_names):
    # create index for each document
    file_to_count = {}
    all_tokens = set()
    for file_name in file_names:
        with open("Selma/" + file_name, 'r') as f:
            counter = Counter(tokenize(f.read().lower()))

            file_to_count[file_name] = counter # {file_name: {token
    : count}}
            all_tokens = all_tokens.union(counter.keys())

    # Create tf
    tf = {file_name: {} for file_name in file_names} # {file_name:
    {token: tf}}
    for file_name, token_to_count in file_to_count.items():
        for token, count in token_to_count.items():
            tf[file_name].update({token : count/sum(list(
    token_to_count.values()))})

    # Compute idf
    idf = {} # {token: idf}
    N = len(file_names)
    for token in all_tokens:
        idf[token] = np.log10(N / len([_
                                       for file_name in file_names
                                       if token in file_to_count[
    file_name]]))

    # combine into tfidf
    tfidf = {file_name : {} for file_name in file_names}
    for all_token in all_tokens:
        for file_name, token_to_tf in tf.items():
            value = token_to_tf[all_token]*idf[all_token] if
    all_token in token_to_tf else 0.0

            tfidf[file_name].update({all_token: value})

    return tfidf
```

Listing 5: tf-idf

In order to compare documents one may use the *tf-idf*. This metric can be generated by using the function displayed in code snippet 5. The functions work

by firstly creating a nested map with file name as the primary key and a value consisting of a mapping between the tokens in the text and their occurrence counts. The relative term frequency (tf) can subsequently be computed and stored in a similar nested map structure by using token counts. Similarly, the inverse document frequency (idf) can be computed by iterating through all tokens in the full corpus and checking which documents they occur in. The tf and idf can is subsequently combined to form the tf-idf which is stored in a nested map structure using the file name as the primary key and token as secondary key.

1.6 Comparing Documents

```
# Write your code here
from scipy.spatial.distance import cosine
import pandas as pd

def cosine_similarity(document1, document2, tfidf=tfidf, file_names
    =get_files('Selma', 'txt')): # e.g. troll.txt
    if not tfidf:
        tfidf = get_tfidf(file_names)

    return np.dot(list(tfidf[document1].values()), list(tfidf[
    document2].values())) \
                / np.linalg.norm(list(tfidf[document1].values()))
     \
                / np.linalg.norm(list(tfidf[document2].values()))
```

Listing 6: Cosine Similarity

The tf-idf metric is useful for comparing documents. By using the normalized dot product (cosine similarity) of two documents tf-idf one may compute a similarity score between a pair of documents as shwon in code snippet 6. One may perform this operation for all documents in the corpus and thereby generate the similarity matrix displayed in table **??**.

	troll.txt	kejsaren.txt	marbacka.txt	herrgard.txt	nils.txt	osynliga.txt	jerusalem.txt	bannlyst.txt	gosta.txt
troll.txt	1.000000	0.181284	0.147154	0.004074	0.188475	0.192597	0.007057	0.088621	0.195738
kejsaren.txt	0.181284	1.000000	0.071121	0.000740	0.049663	0.051108	0.001834	0.024009	0.048018
marbacka.txt	0.147154	0.071121	1.000000	0.003614	0.084741	0.093168	0.004873	0.036810	0.080168
herrgard.txt	0.004074	0.000740	0.003614	1.000000	0.005068	0.004826	0.370689	0.000949	0.003110
nils.txt	0.188475	0.049663	0.084741	0.005068	1.000000	0.110574	0.004539	0.050982	0.104826
osynliga.txt	0.192597	0.051108	0.093168	0.004826	0.110574	1.000000	0.028300	0.052055	0.124755
jerusalem.txt	0.007057	0.001834	0.004873	0.370689	0.004539	0.028300	1.000000	0.006460	0.004321
bannlyst.txt	0.088621	0.024009	0.036810	0.000949	0.050982	0.052055	0.006460	1.000000	0.049037
gosta.txt	0.195738	0.048018	0.080168	0.003110	0.104826	0.124755	0.004321	0.049037	1.000000

Table 1: Similarity Matrix

2 Google Indexing

Jeff Dean's presentation on Google indexing techniques covers Google's indexing techniques and the infrastructure around them. Slide 14 titled *Research Project*

1997 displays an indexing architecture that is similar to the one used in the lab. In the indexing script, one does not really have a doc server however the *Selma* directory has the same function. The `open(Selma/ + file_name, r)` constitutes a query to our so-called 'docs server' and the only index shard is our master index map. Comparing this to slide 37 *Dealing with growth* one may note that our program does not make use of a cache server functionality for common queries. We also do not use any encodings for the document, instead they are 'encoded' with simply their file name, e.g 'troll.txt'. On the other hand the encoding technique is, in a way, similar to the one displayed on slide 45. In 1997 Google also used a map structure where 'docid' corresponds to a list of hit mappings. Similar to in our program the hits contain the index, however, it also contains other information such as font type or size. This information is absent in our index mapping. Also, we do not store the number of occurrences of a certain hit, this is instead accessed through computing the length of the list containing the indices of the hit. We also do not make use of index shard replicas as shown in the slide. This works for the simple application of indexing a few of Selma Lagerlöfs novels however as Dean notes in his presentation it also comes with some problems. One benefit of our program is that it is easy to set up and keep track of. On the other hand, our program is computationally expensive and vulnerable to machine failure.