

# L1: Information retrieval

In this lab you will apply basic techniques from information retrieval to implement the core of a minimalistic search engine. The data for this lab consists of a collection of app descriptions scraped from the [Google Play Store](#). From this collection, your search engine should retrieve those apps whose descriptions best match a given query under the vector space model.

## Data set

The app descriptions come in the form of a compressed [JSON](#) file. Start by loading this file into a [Pandas DataFrame](#).

In [211...

```
import bz2
import pandas as pd

with bz2.open('app-descriptions.json.bz2') as source:
    df = pd.read_json(source)
```

In Pandas, a DataFrame is a table with indexed rows and labelled columns of potentially different types. Data in a DataFrame can be accessed in various ways, including by row and by column. To give an example, the code in the next cell shows rows 200–204:

In [212...

```
df[200:205]
```

Out[212...

	name	description
200	Brick Breaker Star: Space King	Introducing the best Brick Breaker game that e...
201	Brick Classic - Brick Game	Classic Brick Game!\n\nBrick Classic is a popu...
202	Bricks Breaker - Glow Balls	Bricks Breaker - Glow Balls is a addictive and...
203	Bricks Breaker Quest	How to play\n- The ball flies to wherever you ...
204	Brothers in Arms® 3	Fight brave soldiers from around the globe on ...

As you can see, there are two labelled columns: `name` (the name of the app) and `description` (a textual description). The code in the next cell shows how to access fields from the description column.

In [213...

```
df['description'][200:205]
```

Out[213...

```
200    Introducing the best Brick Breaker game that e...
201    Classic Brick Game!\n\nBrick Classic is a popu...
202    Bricks Breaker - Glow Balls is a addictive and...
203    How to play\n- The ball flies to wherever you ...
204    Fight brave soldiers from around the globe on ...
Name: description, dtype: object
```

## Problem 1: Preprocessing

Your first task is to implement a preprocessor for your search engine. In the vector space model, *preprocessing* refers to any kind of transformation that is applied before a text is vectorized. Here you can restrict yourself to a very simple preprocessing: tokenization, stop word removal, and lemmatization.

To implement your preprocessor, you can use [spaCy](#). Make sure that you read the [Linguistic annotations](#) section of the spaCy 101; that section contains all the information that you need for this problem (and more).

Implement your preprocessor by completing the skeleton code in the next cell, adding additional code as you feel necessary.

```
In [25]: import spacy

nlp = spacy.load("en_core_web_sm", disable=["tagger", "parser"])

def preprocess(text):

    doc = nlp(text)
    #t=list()
    #for token in doc:
    #    if token.is_stop== False and token.is_alpha==True:
    #        t.append(token.lemma_)
    return [token.lemma_ for token in doc if token.is_stop == False and token.is_alpha == True]
```

Your implementation should conform to the following specification:

**preprocess** (*text*)

Preprocesses given text by tokenizing it, removing any stop words, replacing each remaining token with its lemma (base form), and discarding all lemmas that contain non-alphabetical characters. Returns the list of remaining lemmas (represented as strings).

**Tip:** To speed up the preprocessing, you can disable loading those spaCy components that you do not need, such as the part-of-speech tagger, parser, and named entity recognizer. See [here](#) for more information about this.

Test your implementation by running the following cell:

```
In [26]: preprocess('Apple is looking at buying U.K. startup for $1 billion')
```

```
Out[26]: ['Apple', 'look', 'buy', 'startup', 'billion']
```

This should give the following output:

```
['Apple', 'look', 'buy', 'startup', 'billion']
```

## Problem 2: Vectorizing

Your next task is to vectorize the data – and more specifically, to map each app description to a tf-idf vector. For this you can use the `TfidfVectorizer` class from [scikit-learn](#). Make sure to specify your preprocessor from the previous problem as the `tokenizer` – not the `preprocessor`! – for the vectorizer. (In scikit-learn parlance, the `preprocessor` handles string-level preprocessing.)

In [171]...

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

des = df['description']
# to vectorize the data, map each descrip to a ti-idf vectore ( term freq -

tfidf_vectorizer=TfidfVectorizer(use_idf = True,tokenizer = preprocess)
tfidf_mat=tfidf_vectorizer.fit_transform(des)

X = tfidf_mat
```

Test your implementation by running the following cell:

In [30]:

```
X.shape
```

Out[30]: (1614, 20681)

This should show the dimensions of the matrix `X` to be  $1614 \times 20669$ .

## Problem 3: Retrieving

To complete the search engine, your last task is to write a function that returns the most relevant app descriptions for a given query. An easy way to do solve this task is to use scikit-learn's [NearestNeighbors](#) class. That class implements unsupervised nearest neighbours learning, and allows you to easily find a predefined number of app descriptions whose vector representations are closest to the query vector.

In [239]...

```
from sklearn.neighbors import NearestNeighbors

def search(query):
    nbrs = NearestNeighbors(n_neighbors=10).fit(tfidf_mat)

    X = tfidf_vectorizer.transform([query])

    ind = nbrs.kneighbors(X, return_distance=False)

    return df.loc[ind[0]][:10]
```

Your implementation should conform to the following specification:

**search** (*query*)

Returns the 10 app descriptions most similar (in terms of cosine similarity) to the


given query as a Pandas DataFrame.

Test your implementation by running the following cell:

In [232...

```
search('dodge trains')
```

Out[232...

	name	description
1301	Subway Surfers	DASH as fast as you can! \nDODGE the oncoming ...
1300	Subway Princess Runner	Subway princess runner, Bus run, forest rush w...
1428	Train Conductor World	Master and manage the chaos of international r...
998	No Humanity - The Hardest Game	2M+ Downloads All Over The World!\n\n* IGN Nom...
1394	Tiny Rails	All aboard for an adventure around the world!\n...
1429	Train for Animals - BabyMagica free	 BabyMagica "Train for Animals" is a educatio...
1168	Rush	Are you ready for a thrilling ride?\n\nRush th...
1077	Polar Flow – Sync & Analyze	Polar Flow is a sports, fitness, and activity ...
1286	Strava: Track Running, Cycling & Swimming	Track your fitness with Strava activity tracke...
1465	Virus War - Space Shooting Game	Warning! Virus invasion! Destroy them with you...

The top hit in the list should be *Subway Surfers*.

## Problem 4: Finding terms with low/high idf

Recall that the inverse document frequency (idf) of a term is the lower the more documents from a given collection the term appears in. To get a better understanding for this concept, your next task is to write code to find out which terms have the lowest/highest idf with respect to the app descriptions.

Start by sorting the terms in increasing order of idf, breaking ties by falling back on alphabetic order.

In [308...

```
np.argsort(tfidf_vectorizer.idf_)
```

Out[308...

```
array([ 5832, 11485,  5194, ..., 10869, 10855, 20680])
```

In [306...

```
# TODO: Replace the next line with your own code.
```

```
terms = np.array(tfidf_vectorizer.get_feature_names())[np.argsort(tfidf_vect
```

Out[306...

```
array(['game', 'play', 'feature', ..., 'overhaul', 'outworld', 'flying'],
      dtype='<U44')
```

Now, print the 10 terms with the lowest/highest idf. How do you explain the results?

In [309...

```
## the lowest idf
print(terms[:10])
```

```
['game' 'play' 'feature' 'free' 'new' 'world' 'time' 'app' 'fun' 'well']
```

*TODO: Enter your explanation here*

The idf score is lower if the term appears in more documents frequently, so the result seems quite reasonable here, because above words are normally will be appeared the most in the description of a game.

## Problem 5: Keyword extraction

A simple method for extracting salient keywords from a document is to pick the  $k$  terms with the highest tf-idf value. Your last task in this lab is to implement this method. More specifically, we ask you to implement a function `keywords` that extracts keywords from a given text.

In [327...

```
def keywords(text, n=10):  
    features = np.array(tfidf_vectorizer.get_feature_names())  
    input_texts= tfidf_vectorizer.transform( [text])  
    sort_tfidf= np.argsort(response.toarray()).flatten()[::-1] #highest  
    return features[tfidf_sorting][:n]
```

Your implementation should conform to the following specification:

**keywords** (*text*, *n* = 10)

Returns a list with the *n* (default value: 10) most salient keywords from the specified text, as measured by their tf-idf value relative to the collection of app descriptions.

Test your implementation by running the following cell:

In [328...

```
print(keywords(df['description'][1428]))  
['train' 'railroad' 'railway' 'rail' 'chaos' 'crash' 'haul' 'tram'  
 'locomotive' 'overcast']
```

This should give the following output:

```
['train', 'railway', 'railroad', 'rail', 'chaos', 'crash', 'overcast', 'locomotive', 'timetable', 'railyard']
```

Please read the General information section on the lab web page before submitting this notebook!