# 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.

**Before starting with this lab, make sure that you have read the Rules for hand-in assignments and the Policy on cheating and plagiarism.**

## Data set

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

In [10]:
```
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 [11]:
```
df[200:205]
```

Out[11]:

| | 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 acess fields from the description column.

In [12]:
```
df['description'][200:205]
```

Out[12]:
```
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 [42]:
```
import spacy
import en_core_web_sm

def preprocess(text):
    nlp = en_core_web_sm.load()
    res = [token.lemma_  for token in nlp(text) if token.is_stop == False and token.
    return res
```

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 parser, and named entity recognizer. See here for more information about this.

Test your implementation by running the following cell:

In [44]:
```
preprocess('Apple is looking at buying U.K. startup for $1 billion')
```

Out[44]:
```
['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 [52]:
```
df['description'][:10]
```

Out[52]:
```
0    10000000 is a Dungeon Crawling Puzzle RPG Mat...
1    I 1177 Vårdguidens app får du tillgång till 11...
2    Need counting games for kids & drawing for tod...
```

```
3    Beautiful and simple music application for tod...
4    A Fun and intuitive numbers game for your baby...
5    Animal Sounds is a very good entertainment gam...
6    The classic, long-run shooting game from the 9...
7    Duel your friends in a variety of 4 player min...
8    2048 is an addictive number puzzle game based ...
9    This is a fun shooter with a view from above. ...
Name: description, dtype: object
```

In [61]:
```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
# TODO: Replace the next line with your own code.

vectorizer = TfidfVectorizer(tokenizer = preprocess)
X = vectorizer.fit_transform(df['description'])

#X = np.zeros((len(df), 1))
```

Test your implementation by running the following cell:

In [62]:
```
X.shape
```

Out[62]:
```
(1614, 21679)
```

This should show the dimensions of the matrix `X` to be 1614 × 21427.

# 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 [65]:
```
from sklearn.neighbors import NearestNeighbors

neigh = NearestNeighbors(n_neighbors=10)
neigh.fit(X)
```

Out[65]:
```
NearestNeighbors(n_neighbors=10)
```

In [76]:
```
def search(query):
    vector = vectorizer.transform([query])
    res = neigh.kneighbors(vector, return_distance=False)
    # TODO: Replace the next line with your own code.
    return df.iloc[res.flatten()]
```

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 [77]:
```
search('dodge trains')
```

Out[77]:

| | name | description |
|---|---|---|
| **998** | No Humanity - The Hardest Game | 2M+ Downloads All Over The World!\n\n* IGN Nom... |
| **1300** | Subway Princess Runner | Subway princess runner, Bus run, forest rush w... |
| **1168** | Rush | Are you ready for a thrilling ride?\n\nRush th... |
| **1301** | Subway Surfers | DASH as fast as you can! \nDODGE the oncoming ... |
| **1465** | Virus War - Space Shooting Game | Warning! Virus invasion! Destroy them with you... |
| **1153** | Road Riot | Road Riot is the global sensation that defined... |
| **360** | Dancing Road: Color Ball Run! | Try out the most exciting Running - Sliding - ... |
| **757** | Kids Numbers and Math | Wouldn't it be just wonderful if there was a s... |
| **384** | Dialog Mega Run | Run around the iconic places of Sri Lanka !\nD... |
| **584** | Galaxiga - Classic 80s Arcade | Galaxiga is a best space shooting game that ma... |

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 [96]:
```
# TODO: Replace the next line with your own code.
idf_df = pd.DataFrame(vectorizer.idf_, index=vectorizer.get_feature_names(),columns=
terms = idf_df.sort_values(by='idf_weights',  ascending=True).index
```

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

In [98]:
```
print(terms[:10])
print(terms[-10:])
```

```
Index(['game', 'play', 'feature', 'free', 'new', 'world', 'time', 'app', 'fun',
       'use'],
      dtype='object')
Index(['lynx', 'lyon', 'lyrica', 'lyssningar', 'lãnh', 'lão', 'lägg', 'lägger',
       'lydia', 'flye'],
      dtype='object')
```

# 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 [127…
```
def keywords(text, n=10):
```

```
    vector = vectorizer.transform([text]).toarray()
    idf_df =  pd.DataFrame({"tfidf" : vector.flatten(), "names" : vectorizer.get_fea
    return idf_df.sort_values(by=['tfidf', "names"],  ascending=False)["names"].toli
```

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 [128…

```
print(keywords(df['description'][1428]))
```

```
['train', 'railway', 'railroad', 'rail', 'chaos', 'crash', 'tram', 'timetable', 'rai
lyard', 'overcast']
```

This should give the following output:

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

# Reflection questions

The following reflection questions are questions that you could be asked in the oral exam. Try to answer each of them in the form of a short text and put it in the cell below. You will get feedback on your answers from your lab assistant.

**RQ 1.1:** Why do we remove common stop words and lemmatise the text? Can you give an example of a scenario where, in addition to common stopwords, there are also *domain-specific* or *application-specific* stop words?

**RQ 1.2:** In Problem 2, what do the dimensions of the matrix `X` correspond to? This matrix gives rise to different types of *representations*. Explain what these representations are. What information from the data is preserved, what information is lost in these representations?

**RQ 1.3:** What does it mean that a term has a high/low idf value? Based on this, how can we use idf to automatically identify stop words? Why do you think is idf not used as a term weighting on its own, but always in connection with term frequency (tf–idf)?

**RA 1.1:**

With respect to lemmatization, it's usually a necessary practise in tokenization. If we don't perform this technique we would generate continuously new vocabulary for our app whose actual equal meaning is the same as their respective lemma. Therefore, it's a more efficient task to just account for the lemma when tokenizing.

When it comes to the stop-words, they appear in the language corpus so many times that they lose importance in the process of tokenizing or generating vocabulary. Generally it is better to exclude them because they don't add strict important information other than little nuances, as in the case of derived words compared to their lemma.

Domain-or-app-specific stop-words act like common stop-words under some context or scenarios. Eg: In the Environmental Sciences words such as "Nature" or "Ecosystem" are overly

repeated until the point they lose some meaning in the context and add no additional relevant information.

**RA 1.2:**

rows are data, columns are unique words. every document is represented as a vector of tfidf values. so we keep information of word frequences in document (tf) and word frequency in all documents (idf). we loose linguistic relations to other words (POS TAG DEP SHAPE)

**RA 1.3:**

high idf - word is not frequent in all documents, low - opposite. stop words are frequent so idf will be low, so n words with lowest idf will be stop words. idf is not used alone because it doesnt say anything about the term frequency in the document (if term is frequent in a document - it should more important).

**Congratulations on finishing L1!** 👍