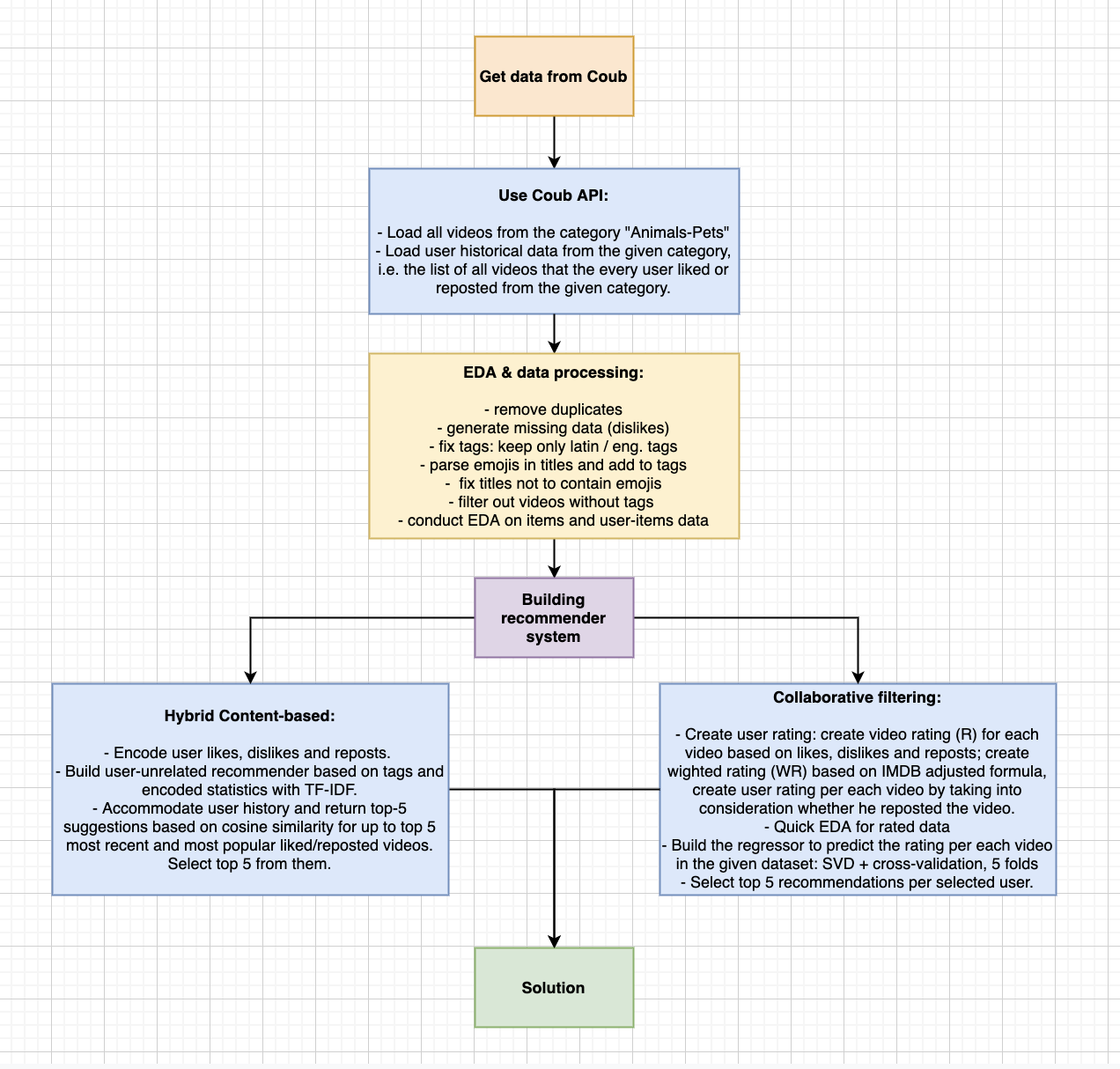
Coub recommender system demo. Final report.

The purpose of this project has been to conduct research and build the prototype recommender system to suggest new coub videos, which are likely to fit their interests and previous history. The category chosen for the current demo is **Animals-Pets.** The flow of the project is presented below:

**Figure 1 – Project flow**



**Data Load**

In the first place we need to download the input data from Coub. This data includes two data sources:

* The list of all videos from the selected category Animals-Pets along with relevant metadata, like title, date of creation, likes and dislikes, reposts, tags, URL, etc.
* The list of all user history data with respect to the given videos.

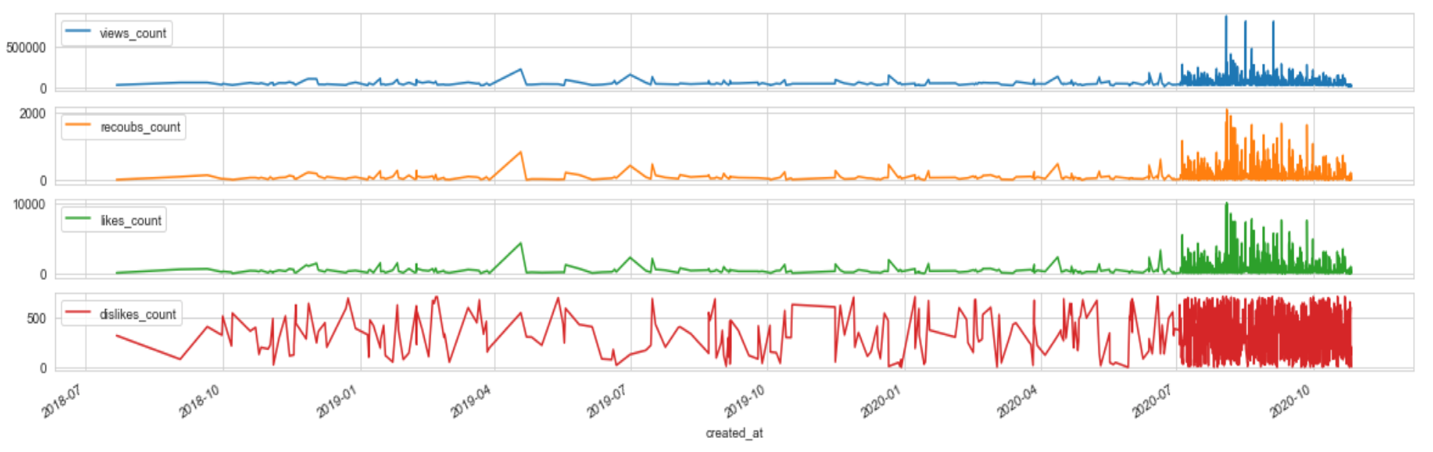
Once data is loaded, preprocessing starts. Filter out all non-Latin tags from the tag-list, parse the emojis that might be present in the title, remove them from the title along with the all non-alpha-numeric and non-Latin characters. Remove duplicates. It appeared that for the same videos we had multiple entries, which differed only by number of views. Thus, I kept only the most recent – the bigger number. The result has been that data has shrunk by 2.5 times.

Example of processing the emojis string:

'Little #kitty cat is so funny 😺😂😂' 🡪 ['little', 'cat', 'kitty', 'funny', 'grinning', 'face', 'joy', 'tears']

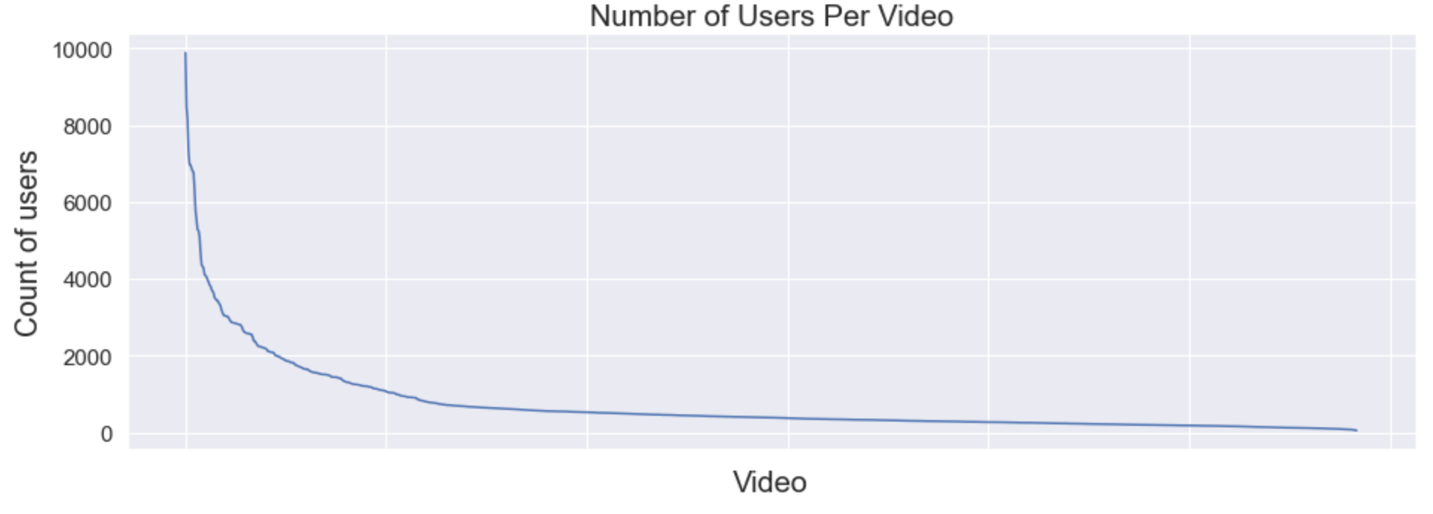
Let’s see what the EDA has to say us about our data.

**Figure 2 – Temporal distribution of likes, reposts and dislikes**



It seems that most of the activity in the current category is quite recent and gained momentum in July 2020. Since we didn’t have the real dislikes data, I randomly generated them.

**Figure 3 – Number of users per video**



The figure shows most of our users didn’t watch too many movies, up to 500 maybe, but some videos are just top tier performers with more than 10000 unique users.

More details on the EDA results are in the research notebook.

**Building Recommender for our videos**

1. **Content-Based System**

At first, I went with the item content-based approach. We have abundant metadata available, so I used it to build impersonalized recommendations based on the item similarity only. I encoded the likes, dislikes and reposts columns by dividing them into low, middle and high ranges. This text representation, along with the title and tags, has been concatenated into a string to be analyzed through the TF-IDF vectorizer and comparing the cosine similarity of the given text string. But before using TF-IDF, the lemmatization of words has been performed to keep only the root of the word.

The **TD-IDF** algorithm is used to weigh a keyword in any document and assign the importance to that keyword based on the number of times it appears in the document. Put simply, the higher the TF-IDF score (weight), the rarer and more important the term, and vice versa.

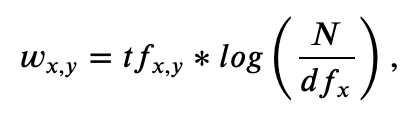
Each word or term has its respective TF and IDF score. The product of the TF and IDF scores of a term is called the TF-IDF weight of that term.

The **TF (term frequency)** of a word is the number of times it appears in a document. When you know it, you’re able to see if you’re using a term too often or too infrequently.

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).

The **IDF (inverse document frequency)** of a word is the measure of how significant that term is in the whole corpus.

IDF(t) = log\_e(Total number of documents / Number of documents with term t in it).



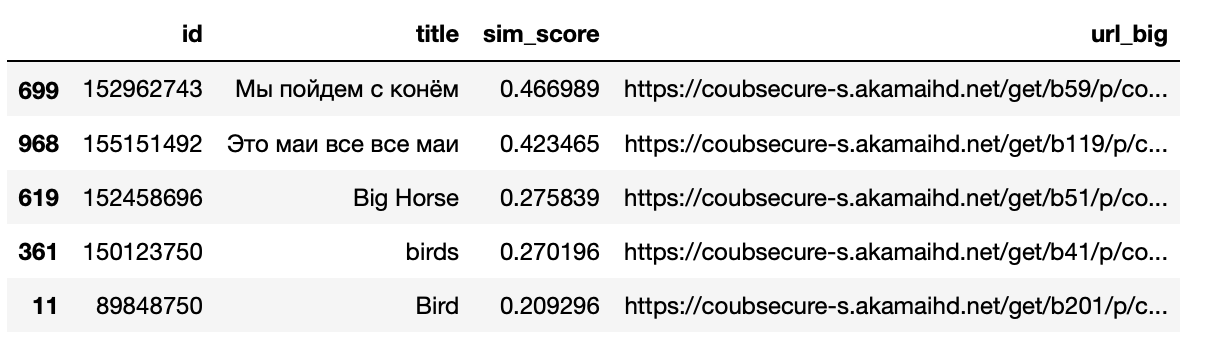
where:

* tf\_{x,y} - frequency of x in y
* df\_x - number of documents containing x
* N - total number of documents

This is a good way to represent text strings and based on the cosine similarity – define most similar items.

However, at present no previous user history is taken into consideration. Thus, I built for every user I sorted his up to top 5 most recent and most popular items, and against every one of them ran the content-based recommender defined above. As a result, I ended up with up to 25 movies, of which based on the cosine similarity the algorithm returns only top-5. But if we want to specify the user and the exact video to perform recommendation for and return top-5 similar items, this option is available too. Finally if you want to visually inspect the result, open up the link specified in the url\_big field.

**Figure 4 – Sample output from the recommender engine**

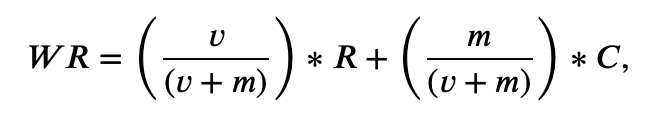


1. **Collaborative filtering**

Next, I decided to check out how the collaborative filtering would work on our data. But in order to proceed I needed to create the rating for each video, since it wasn’t initially available to us. I had two options – randomly create it or try to integrate some logic while using the existing statistics on the very same likes, dislikes and reposts information which we already have. So, I went with the second option. Firstly, I introduced the rating per movie following the next logic: I binarize the mentioned above columns into three bins each and assign the value from 2 to 4 with respect to where specific value falls into except for dislikes, which amount to 1 to 3 - we don't want to penalize that much.

**R = (views\_count(score) + recoubs\_count(score) + likes\_count(score) - dislikes\_count(score))**

In this way we incorporated the statistics of interest. But it’s not quite weighted yet, since different users differently assess the video, and number of assessments also varies. Therefore, I adjusted the following IMDB formula to account for this:



where:

* v - the number of votes for the movie;
* m - the minimum votes required to be listed in the chart;
* R - the average rating of the movie;
* C - the mean vote across the whole report.

Since we don't have the number of votes, let's create them based on the assumption:

**#votes = int(0.10 \* #views)**;

The m, or minimal number of votes required to be listed is set to be 90% percentile of v;

C is easy to obtain by taking an average accross all ratings in the dataframe

Once the weighted average rating is computed, we can calculate the individual rating of the coub using the following logic:

* if the user only liked the coub, the result of WR is multiplied by 1, i.e. equals the WR;
* else if he reposted the coub onto his page, the WR is multiplied by 1.25.
* I limit the votes to 5 points at the most to avoid very imbalanced rating distribution (it appeared in our set ratings with >5 were very underrepresented, so I decided to put them into one category)

Okay, done with this. Here’s how the rating distribution looks like:

