Data-Driven Web APIs Recommendation System

Ciampi Giovanni[1]

[1]Università degli Studi di Salerno

g.ciampi5studenti.unisa.it

Abstract. *The world of Web APIs is growing at a very fast pace: a developer who intends to build a particular application can easily get lost in thousands of them. ProgrammableWeb is the main repository, and in 2019, it counts more than 18000 APIs. In recent years, many efforts have been put in order to build recommendation systems, that could help developers to find what they need. Unfortunately, these recommendation systems are not so straightforward to build, since the APIs are often not well documented, and there is not much information on the use that is being made of them. In 2015, Dojchinovski and Vitvar presented an online dataset containing, in addition to information on the semantic content of APIs, information on their usage to bouild mashups. In this work is presented an algorithm that tries to exploit this data in order to give accurate recommendations to the users. If used in its optimal way, the presented algorithm is able to give recommendations that are accepted, on average, on* $\frac{1}{3}$ *of the cases.*

Keywords*: Web APIs, Recommendation Systems, ProgrammableWeb.*

1. Introduction

The popularity of web APIs is growing everyday, since web services providers release them in order to allow developers to access their services. ProgrammableWeb is the main APIs repository, and to realize how fast the universe of APIs is growing, it is sufficient to notice that while in 2010 ProgrammableWeb hosted 3000 APIs, and in 2012 it hosted 5000 APIs, today it is hosting more than 18000 of them. Since the number of available APIs is becoming so big, it is becoming overwhelmingly difficult for developers to orientate in such world, and to find what they need. Although there are some features that try to make the research of the developer easier, such as the semantic description of APIs, or the informations about the usage that other developers make of them, the task still remains hard. Moreover, often these high-level information are not so helpful, because every user can post on the repository without much control, and for this reason the details could be inaccurate, for example lacking significant information. In 2015 Dojchinovski and Vitvar[1] made available online a dataset containing informations on many ProgrammableWeb's APIs and their usage. In particular, this work focuses on the relationship between mashups, tags and APIs. On ProgrammableWeb there are information not only on the APIs, but even on the mashups whose realization they have been used for. These information have been collected in the Linked Web APIs Dataset by Dojchinovski and Vitvar, and an attempt has been made in order to exploit them to provide recommendations to the users based on keywords that describe what they are attempting to build, and APIs they already decided to utilize.

[1]http://linked-web-apis.fit.cvut.cz/

Mashup Name	Mashup Tags	Mashup APIs
['rate-tour-guides']	['travel', 'tour', 'maps', 'tours', 'exotic', 'locations']	['google-maps-flash', 'google-maps', 'google-maps-data']
['global-flood-map']	['global', 'climate', 'map', 'warming', 'flood', 'change']	['google-maps-flash', 'google-maps']
['plocky.com-get-your-e-life-together']	['social', 'network', 'aggregation']	['facebook']

Table 1. Example of prepared data

2. Data Overview

As specified, all the data has been retrieved from the Linked Web APIs, collected from ProgrammableWeb by Dojchinovski and Vitvar in 2015. The full dataset provides information on the APIs and on a good amount of mashups in n-triples format, and it is fully accessible online via a SPARQL editor[2] that enables the user to query the dataset from the web browser. As regards the information provided, for many APIs semantic categories are specified, as well as the supported protocols and data formats. For what concerns the mashups, tags are specified for each one, that provide some high level information on their semantic content. The most interesting information provided by the dataset is pheraps the association between mashups and APIs, in particular, for each mashup, a list of APIs utilized for its realization are indicated. This is the main information that was tried to exploit in order to give recommendations to the users. A full specification of the dataset and its ontology are available on the dataset's webpage[3]

3. Proposed Solution

3.1. Environment and Setting

The presented software has been realized in the Python programming language (version 3.7.2), with the Jupyter Notebook editor. It has been written and tested on a machine equipped with an intel 8750H processor (i7 6 core 2.2GHz) and 16GB of RAM.

3.2. Data Preparation

The dataset that has been taken into account is available in RDF (Resource Description Framework) format; in particular, it is stored in n-triples. For this reason the data has been retrieved in two different csv files, one representing the *mashup-API* associations, and the other representing the *mashup-tag* associations.
The first step of the data-preparation process had as its aim a more concise representation of the data. In particular, for each atomic entity in the dataset (a specific tag, mashup or API), a link to its specification is provided, in place of the plain *name*. Consider, as an example, the information about the `wolfram-alpha` API: each entry in the dataset refers to it as `"http://linked-web-apis[...]/resource/wolfram-alpha_api"`. In order to get a clearer representation of the data, the first step of the data-preparation process consisted in getting rid of all the useless parts of the data, leaving just the names of the various entities. The second and final step consisted in the creation of a single Python list containing all the associations, in a table-like fashion. As an example, a three-rows table is presented in *Table 1*. The table implementation is showed in *Listing 1*.

[2]http://linked-web-apis.fit.cvut.cz/sparql
[3]http://linked-web-apis.fit.cvut.cz/ns/core/index.html

Listing 1. Structure of the table as implemented

```
table = [[[Name of Mashup #1], [Tags of Mashup #1], [APIs of
    Mashup #1]], [...], [[Name of Mashup #n], [Tags of Mashup #n
    ], [APIs of Mashup #n]], [...]]
```

4. Recommendation Algorithm

The recommendation algorithm is shown in *Listing 2*, implemented as the `get_recommendations` function. The function can be used in a variety of ways, that will be presented shortly. Here the main parameters of the function are illustrated: the first parameter is `dataset`, which represents the data on which the recommendations will be based. In particular, it has to have the form of a Python list structured as shown in *Listing 1*. It is necessary for the user to call the function with a non-empty dataset, otherwise the function won't run. The second parameter is `tag_list`; it represents a list of tags given in input by the user. Tags are keywords that describe the semantic content of the program that the user intends to realize. This parameter can be left empty (in order to get recommendations based only on the APIs list), but, for better performances, it is suggested to avoid this latter use-case. The parameter `input_api_list` represents a list of APIs that can be given as input by the user. This list should be filled with APIs that the user already decided to use for the mashup he intends to realize. The parameter `treshold` represents a value that indicates when a mashup in the dataset has to be considered similar enough to be used for the recommendations. The parameter `limit_results`, set to an integer $n > 0$ will set the function to return the first n suggestions, discarding the others.

Listing 2. The get_recommendations function

```
def get_recommendations(dataset, tag_list = [], input_api_list =
    [], include_synonyms=False, use_matching_treshold = False,
    treshold = 0.0001, limit_results = None, format_results =
    True):

    api_list = []
    number_of_tags = len(tag_list)
    similar_mashups_count = 0

    if include_synonyms:
        tag_list = get_list_synonyms(tag_list, vb)

    if not use_matching_treshold:
        treshold = 1

    for i in range(0, len(train_set)):

        # in order to remove repeated elements
        current_mashup_tags = list(set(dataset[i][1]))
        current_mashup_api = list(set(dataset[i][2]))

        matching_percentage = compute_matching_percentage(
            tag_list, current_mashup_tags, number_of_tags)

        # If the number of input tags is zero, the filtering is
            based only on the api.
        if matching_percentage >= treshold or (number_of_tags ==
            0 and len(input_api_list) != 0):

            to_add = internal_get_matching_apis(input_api_list,
                current_mashup_api)

            if to_add is not None:
                api_list.extend(to_add)
                similar_mashups_count = similar_mashups_count + 1

    if len(api_list) == 0:
        return "Impossible to give an advice"

    return get_output_api_list(api_list, similar_mashups_count,
        format_results)[:limit_results]
```

4.1. Use Cases

4.1.1. Recommendations based on tags list

The most immediate use-case of the function is the request for recommendations based on a full compatibility between tags. In this case, the user calls the function passing a non-empty list of tags as the only parameter (together with the dataset). An example of this usage is the following:
`get_recommendations(dataset, tag_list=['tag1', 'tag2', ...])`
In this case the parameters `include_synonyms` and `use_matching_treshold` will be automatically set to `False`. In order to compute the recommendations, the function will iterate through all the mashups contained in the dataset: for each mashup, the list of tags will be retrieved. Let T_U be the set of tags given in input by the user, and T_{m_i} be the set of tags of the i-*th* mashup in the dataset: if one between $T_U \subseteq T_{m_i}$ and $T_{m_i} \subseteq T_U$ holds, the APIs used to build the i-*th* mashup will be recommended to the user. Now let R be the full set of APIs that have been collected to be recommended to the user, let $M = \{m_1, m_2, ..., m_n\}$ be the set of mashups these APIs have been collected from (M is the set of mashups for which $T_U \subseteq T_{m_i}$ or $T_{m_i} \subseteq T_U$ held), let A_{m_i} be the set of APIs used for the mashup m_i and finally let $M_a = \{m_j \; s.t. \; m_j \in M \; and \; a \in A_{m_j}\}$: in the list of recommendations returned to the user, each API a will be paired with the value $\frac{|M_a|}{|M|}$, that indicates the percentage of *similar* mashups in which the API was used.

4.1.2. Recommendations based on tags list with matching treshold

This use-case lets the user choose the correlation that needs to exist between the tags he indicates and the tags of the mashups contained in the dataset in order to get recommendations. An example of function call for this usage is the following:
`get_recommendations(dataset, tag_list=['tag1', 'tag2', ...], use_matching_treshold = True, treshold = 0.5)`
A non-empty list of tags is indicated as well as the parameter `use_matching_treshold` set to `True` and the parameter `Treshold` set to `0.5` . The recommendations are computed as follows: the function iterates through all the mashups contained in the dataset; for each mashup the list of associated tag is retrieved. As before, let T_U be the set of tags given in input by the user, and T_{m_i} the tags of the i-*th* mashup in the dataset. The APIs from the mashup m_i are selected to be recommended only if the following holds: $\frac{T_U \cap T_{m_i}}{min(|T_U|,|T_{m_i}|)} \geq Treshold.$
The prosecution is identical to the previous use-case.

4.1.3. Recommendations based on tags list and synonyms

This use-case is very similar to the *recommendations based on tag list*, except for the fact that synonyms of the input tags are considered as well. Formally, in this case $T_U = \{$*Tags input by the user*$\} \cup \{$*Synonyms of tags input by the user*$\}$. In particular, we tested the function using the Vocabulary Python class[4] in order to get synonyms of the tags. An

[4]https://pypi.org/project/Vocabulary/

example of function call for this usage is the following:
`get_recommendations(dataset, tag_list=['tag1', 'tag2', ...], include_synonyms = True)`
Except for what specified, the execution is the same of the use-case presented in *section 4.1.1*.

4.1.4. Recommendations based on tags list, synonyms and treshold

This use case combines the one presented in *section 4.1.2* and the one presented in *section 4.1.3*. The set of tags input by the user is completed with the inclusion of synonyms; at the end, the treshold is used to decide if a mashup is *similar enough* to be used for recommendations. The details of this usage should be clear from *sections 4.1.2* and *4.1.3*, here is an example of function call:
`get_recommendations(dataset, tag_list=['tag1', 'tag2', ...], include_synonyms = True, use_matching_treshold = True, treshold = 0.5)`

4.1.5. Recommendations based on APIs list

This use-case lets the user ask for recommendations based on the correlation between the usage of some specified APIs and all the others that are present in the dataset. In particular, let A_U be the list of APIs input by the user. The function iterates through all the mashups in the dataset, retrieving the lists of APIs for each one: let A_{m_i} be set of APIs of the i-*th* mashup, let M be the set of mashups m_i such that $A_U \subseteq A_{m_i}$. The function returns the set $\{\bigcup_{m_i \in M} A_{m_i}\} - A_U$.
An example of function call for this usage is the following:
`get_recommendations(dataset, input_api_list=['api1', 'api2', ...])`

4.1.6. Recommendations based on tags list and APIs list

This use-case combines all the use-cases presented in *sections 4.1.1-4.1.4* with the one presented in *section 4.1.5*. If a function call from one of the use-cases from *sections 4.1.1-4.1.4* is extended with the inclusion of an APIs list, the execution runs as follows: a first-level filtering is executed without considering the APIs list; so everything is identical to the use-case that did not include it. Then, let M be the set of mashups that have passed the first level filtering (for example, if the function has been called with the parameters `dataset`, `tags_list` and `input_api_list`, $M = \{m_i | T_U \subseteq T_{m_i} \text{ or } T_{m_i} \subseteq T_U\}$), and let M' be the set of mashups m_j such that: $m_j \in M$ and $A_U \subseteq A_{m_j}$ (where A_U and A_{m_j} are as specified in the preceding subsection), the set of recommended APIs will be the set $\{\bigcup_{m_j \in M'} A_{m_j}\} - A_U$.

5. Performances

In order to test the performances of the proposed algorithm, the dataset has been split in a train set, the data used in order to give the recommendations, consisting of 3935 mashups,

and a test set, consisting of 1311 mashups. The two sets were of course disjoint.
To simulate a list of tags input by the user, for each mashup m_i in the test set, a subset of the tags of size *percentage_tags* $\times|T_{m_i}|$ has been sampled, this sampled subset will be referred to as T'_{m_i}. If $|T'_{m_i}| = 0$, because the percentage of tags to sample was too small compared to T_{m_i}, T'_{m_i} has been filled with a randomly selected element of T_{m_i}.
To simulate a list of APIs input by the user, for each mashup, where present, one of the APIs has been randomly selected; except for the testing of the use-case presented in *section 4.1.5*. The parameter `limit_results` has been set to 3 in order to get the three best recommendations. In the rest of this section the set R_{m_i} will represent the set of recommendations the algorithm gave for the mashup m_i.
For each test, the following results have been measured:
#Hits = the number of mashups in the test set with a non-empty set of APIs, for which at least one of the recommended APIs was present in the APIs list.
#Misses = the number of mashups in the test set with a non-empty set of APIs for which no recommended API was in the list of used APIs.
Average Accepted Recommendations = the quantity $\frac{\sum_{m_i \in TestSet} |R_{m_i} \cap A_{m_i}|}{\#m_i \in TestSet\ s.t. |A_{m_i}|, |T_{m_i}| > 0}$
The figures in the following pages summarize the results: in the charts, ***Hit frequency*** represents the value $\frac{\#Hits}{\#Hits + \#Misses}$, and ***AAR*** represents the *Average Accepted Recommendations*.

6. Considerations

From the results, it appears evident that the algorithm performs better when the parameter `treshold` is set between 0.35 and 0.5, and no API is given. Moreover, the usage of synonyms doesn't seem to significatively affect the results[5]. When set with the optimal parameters, almost $\frac{1}{3}$ of the APIs recommended by the proposed algorithm were among the APIs actually used for the realization of the mashup for which the recommendations were asked.
The performances are severely degraded when a list of APIs is given in input: this could be due to the dataset size, that may be too small in order to compute effective recommendations for such specific requests.

[5]This, of course, could be due to the software that has been used in order to provide synonyms.

Figure 1. Hit frequency for recommendations based on tags list

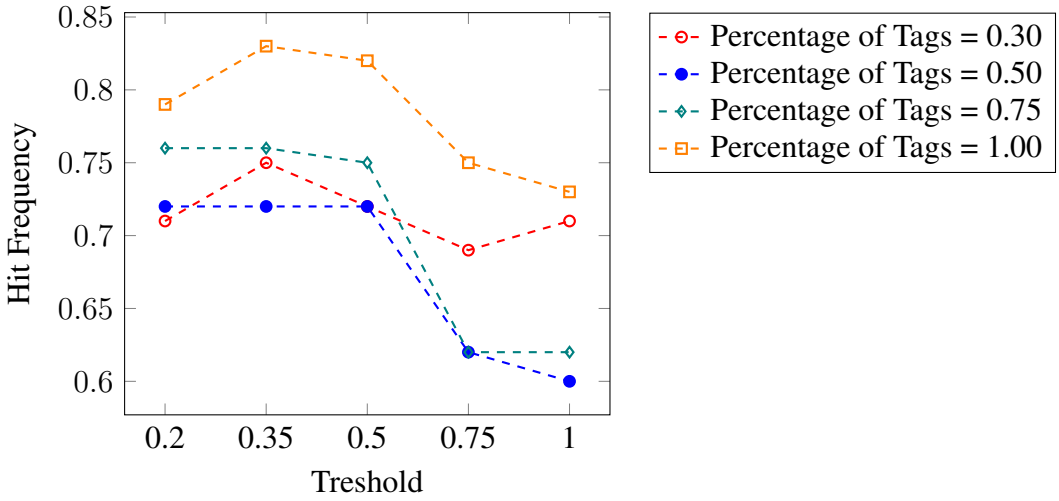

Figure 2. AAR for recommendations based on tags list

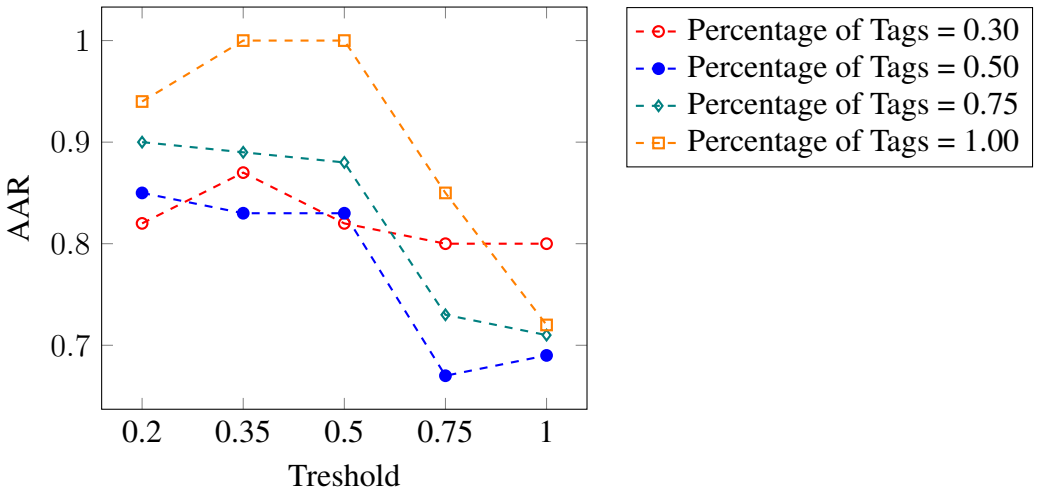

Figure 3. Hit frequency for recommendations based on tags list and synonyms

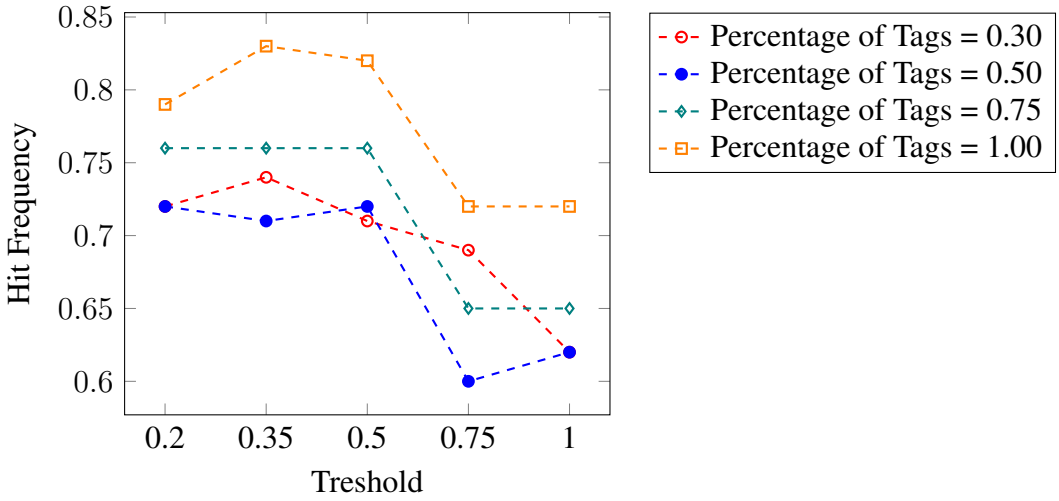

Figure 4. AAR for recommendations based on tags list and synonyms

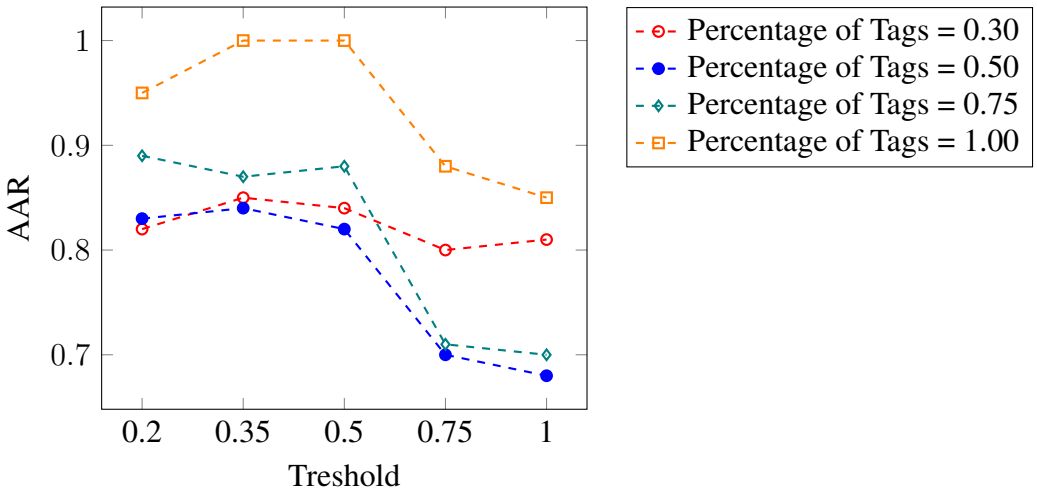

Figure 5. Hit frequency for recommendations based on tags list and API

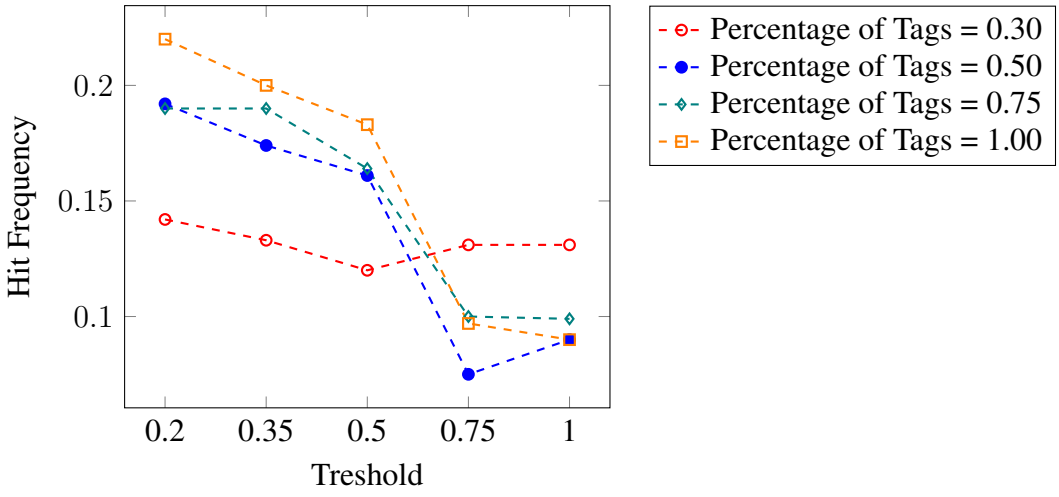

Figure 6. AAR for recommendations based on tags list and API

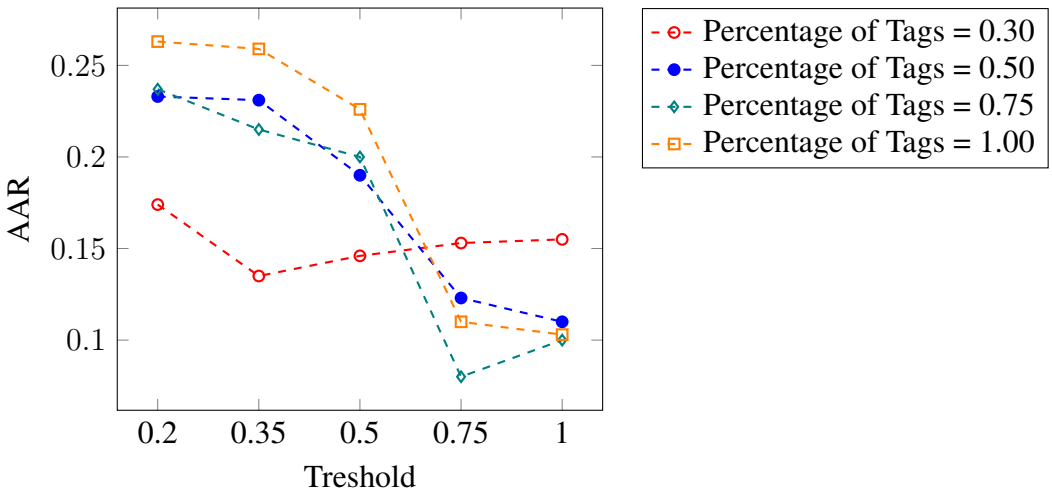

Figure 7. Hit frequency for recommendations based on tags list, synonyms and APIs

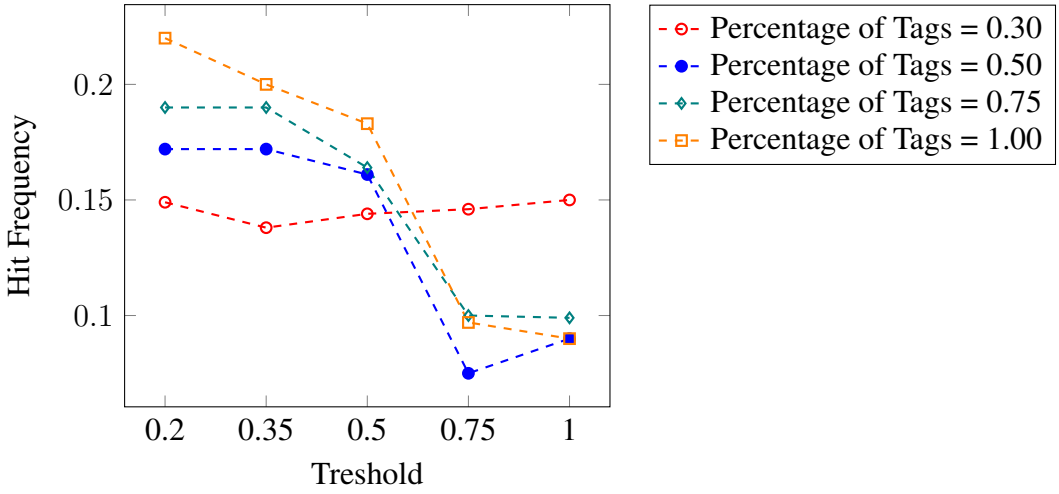

Figure 8. AAR for recommendations based on tags list, synonyms and APIs

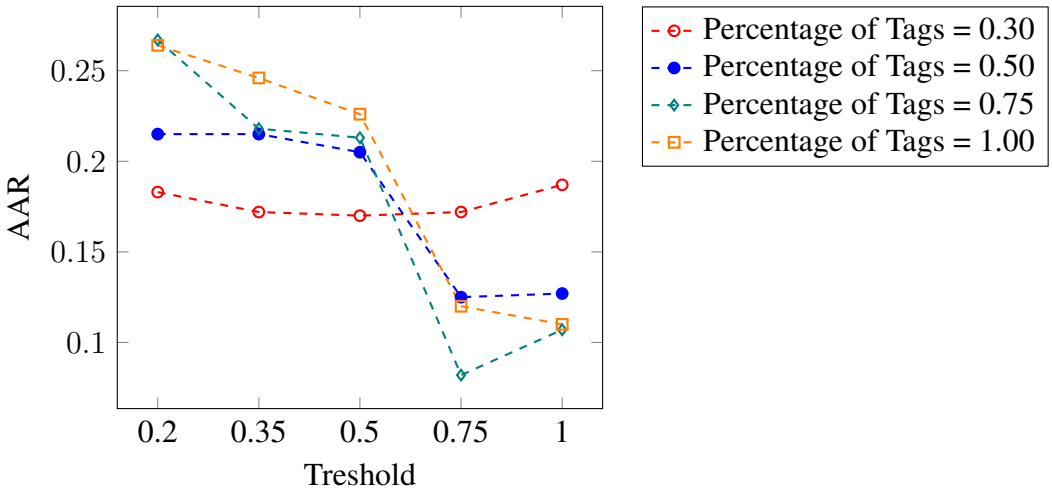

Figure 9. Hit Frequency and AAR for recommendations based on APIs only