ElasticSearch 2019

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

15 DICIEMBRE

Information repositories – Uniovi 19-20
Lab group L.I-2
Written by: Daniel Finca Martínez (UO264469)
Óscar Sánchez Campo (UO265078)

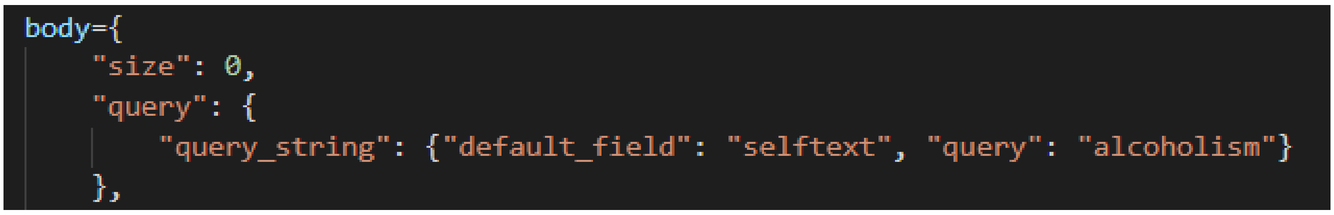
Prologue

For this project, the provided script “bulkIndexer4.py” has been used to generate the index. Process such as stemming, removing stopwords and n-grams generation are used in such script.
Note: Lowercase operation has been commented out and the reason why this has been done will be explained later on (Exercise 3).

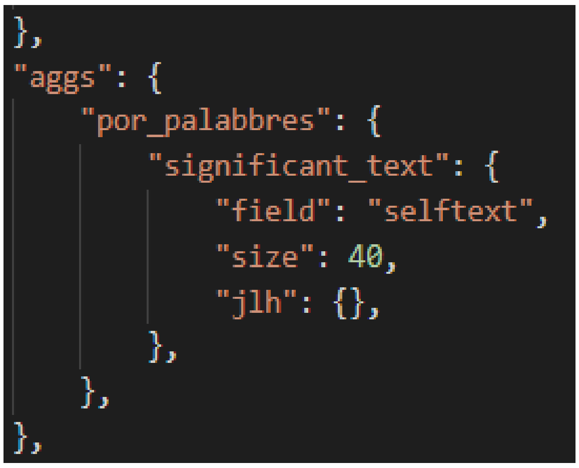
Exercises

Exercise 1

In this exercise “alcoholism” topic has been chosen. As an initial query to retrieve significant terms, a simple structure has been used as shown here:



```
body={
    "size": 0,
    "query": {
        "query_string": {"default_field": "selftext", "query": "alcoholism"}
    },
```



```
},
"aggs": {
    "por_palabbres": {
        "significant_text": {
            "field": "selftext",
            "size": 40,
            "jlh": {},
        },
    },
},
```

To expand the results provided with such request, Elasticsearch query language aggregations feature is added to above body as stated on the image to the right. Using this, most common words used in posts retrieved from above results are grouped by frequency. These new outcomes then serve as the expanded set of terms about the topic chosen for the theme of the exercise.

In addition to above aggregation snippet, different similarity metrics have been used. These are gnd, chi_square and jlh. Different aggregation sizes have been tested to assess their performance. Jlh metric was used with above query. This will be explained later on.

A table can be found below with the relation of the metrics used to the number of words obtained to query the index. To our criteria, jlh has one of the highest precision rates obtained from execution experience.

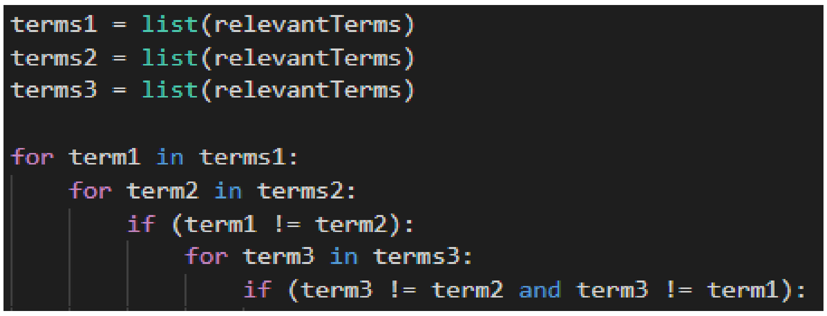
Similarity metric *to* Aggregation size	**GND**	**JLH**	**Chi-square**
20	7 words – 85%	7 words – 100%	7 words – 100%
30	8 words – 95%	8 words – 100%	9 words – 100%
40	13 words – 100%	14 words – 100%	13 words – 95%
50	16 words – 95%	17 words – 95%	17 words – 95%

As mentioned above, in jlh metric has been used to obtain the results (cell marked). These results were obtained considering the first 20 posts which have the highest score. This is to emulate a normal user's search; on Google for instance. Further results would be discarded as they don't seem to be important (user's judge).

From the exercise statement, it can be read that the set of posts related to the chosen topic must be obtained and extracted to a file. To do this, these steps have been followed:

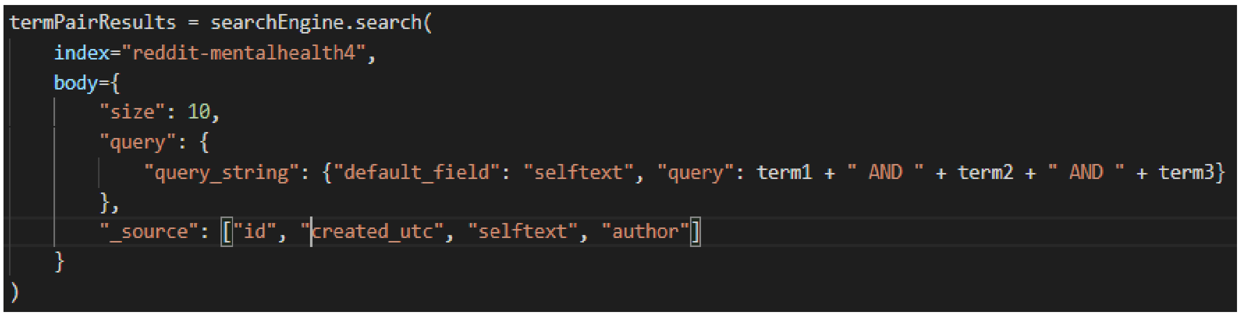
1. Significant terms are taken out from the aggregation previously shown.

2. Then, these are combined in triplets to query the index. In this way, the posts that talk about such combination of terms are retrieved. These tuples are created using three for-loops which avoid same term being queried at the same time, hence being always different.



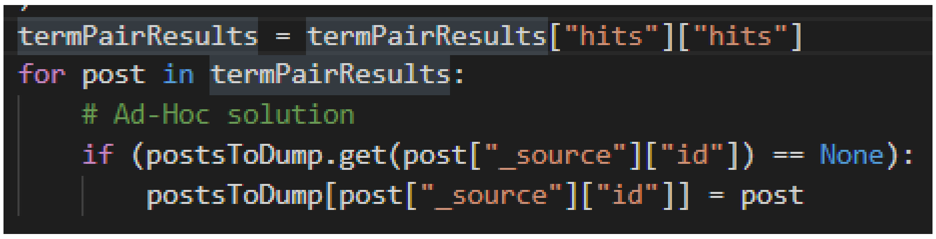
```python
terms1 = list(relevantTerms)
terms2 = list(relevantTerms)
terms3 = list(relevantTerms)

for term1 in terms1:
    for term2 in terms2:
        if (term1 != term2):
            for term3 in terms3:
                if (term3 != term2 and term3 != term1):
```

3. As triplets are generated, they are fed to an Elasticsearch query that looks like this:



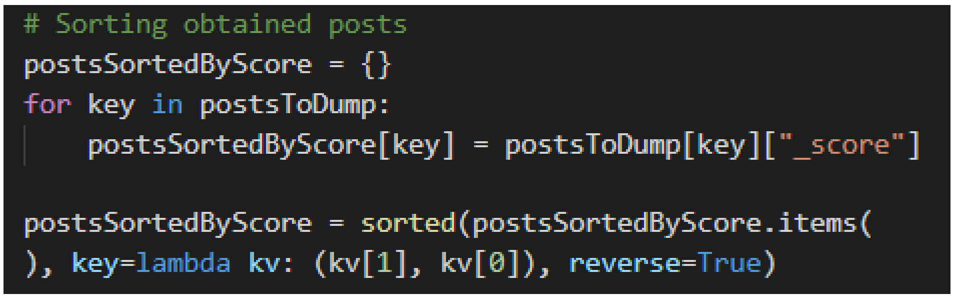
```python
termPairResults = searchEngine.search(
    index="reddit-mentalhealth4",
    body={
        "size": 10,
        "query": {
            "query_string": {"default_field": "selftext", "query": term1 + " AND " + term2 + " AND " + term3}
        },
        "_source": ["id", "created_utc", "selftext", "author"]
    }
)
```

4. As different combinations might generate repeated posts as outcome, this part of the script is used to avoid such issue. Now a dictionary with id to post relation is on our hands to process.



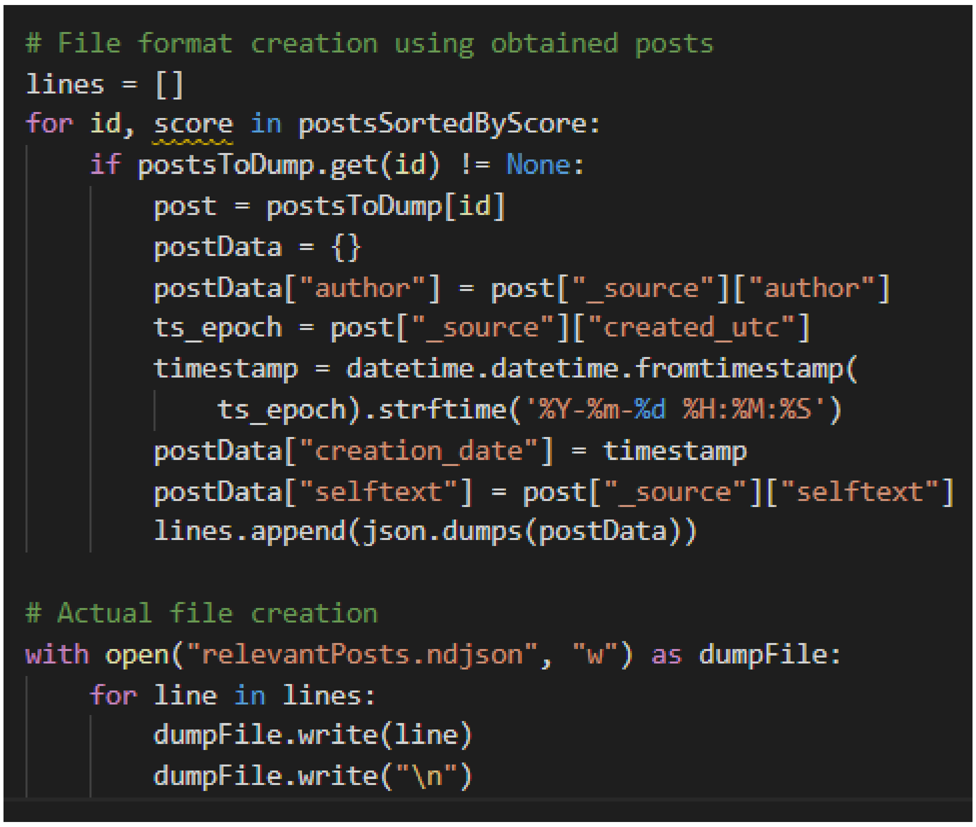
```python
termPairResults = termPairResults["hits"]["hits"]
for post in termPairResults:
    # Ad-Hoc solution
    if (postsToDump.get(post["_source"]["id"]) == None):
        postsToDump[post["_source"]["id"]] = post
```

5. Once the previous relation is obtained, the posts are sorted based on their relevance score; using yet again a fresh new dictionary. This can be used whenever the nth most relevant posts want to be obtained.



```python
# Sorting obtained posts
postsSortedByScore = {}
for key in postsToDump:
    postsSortedByScore[key] = postsToDump[key]["_score"]

postsSortedByScore = sorted(postsSortedByScore.items(
), key=lambda kv: (kv[1], kv[0]), reverse=True)
```

6. Once sorted, format and output of the posts can be produced. Author, selftext and creation date have been added to the output to comply with the statement of the exercise. This output has been written to an ndJson file. Each line represents a post in json format obtained from stringifying python dictionary objects; by means of json package.



```python
# File format creation using obtained posts
lines = []
for id, score in postsSortedByScore:
    if postsToDump.get(id) != None:
        post = postsToDump[id]
        postData = {}
        postData["author"] = post["_source"]["author"]
        ts_epoch = post["_source"]["created_utc"]
        timestamp = datetime.datetime.fromtimestamp(
            ts_epoch).strftime('%Y-%m-%d %H:%M:%S')
        postData["creation_date"] = timestamp
        postData["selftext"] = post["_source"]["selftext"]
        lines.append(json.dumps(postData))

# Actual file creation
with open("relevantPosts.ndjson", "w") as dumpFile:
    for line in lines:
        dumpFile.write(line)
        dumpFile.write("\n")
```

Exercise 2

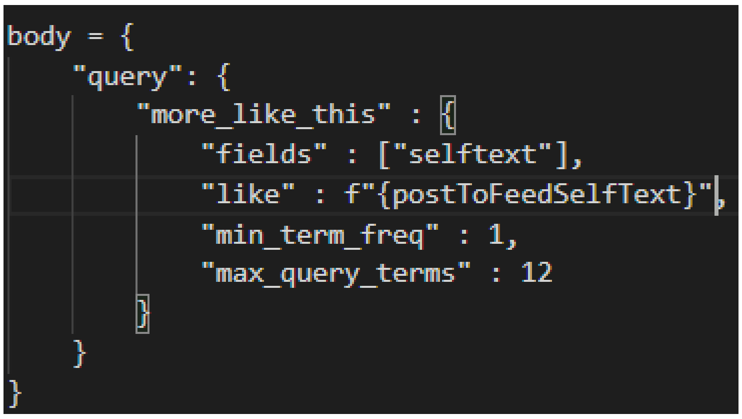
In this exercise, the requested task asks for a suggestion on how to implement "*More like this*" queries. This should pretty much simulate the behaviour of gnd similarity metric used in exercise 1. Somewhat alike to what google does when searching any term, suggesting new complete searches like the one on the input.

On top of the terms found from a query to expand relevant terms to our topic, we can feed in the text of the posts found to obtain related ones. This would be essentially the same as doing it using the terms to feed into the "like" parameter. Anyways, here is an explanation of kind of a pseudocode that might be used.

These are the series of steps followed:

1. To obtain relevant documents for the selected topic a single and normal search query is used. To limit the results, the nth most relevant ones might be extracted.

2. Once done, they can be fed into an Elasticsearch request that uses more_like_this dsl's feature. Where the text of each obtained post from step 1 is introduced into the "like" parameter. The syntax was obtained from the links provided in some document from the assignment material. These queries are easy to create using python f"" formatting syntax as shown in the picture included. Where "postToFeedSelfText" could be a loop iterator taking out text from previously retrieved posts.

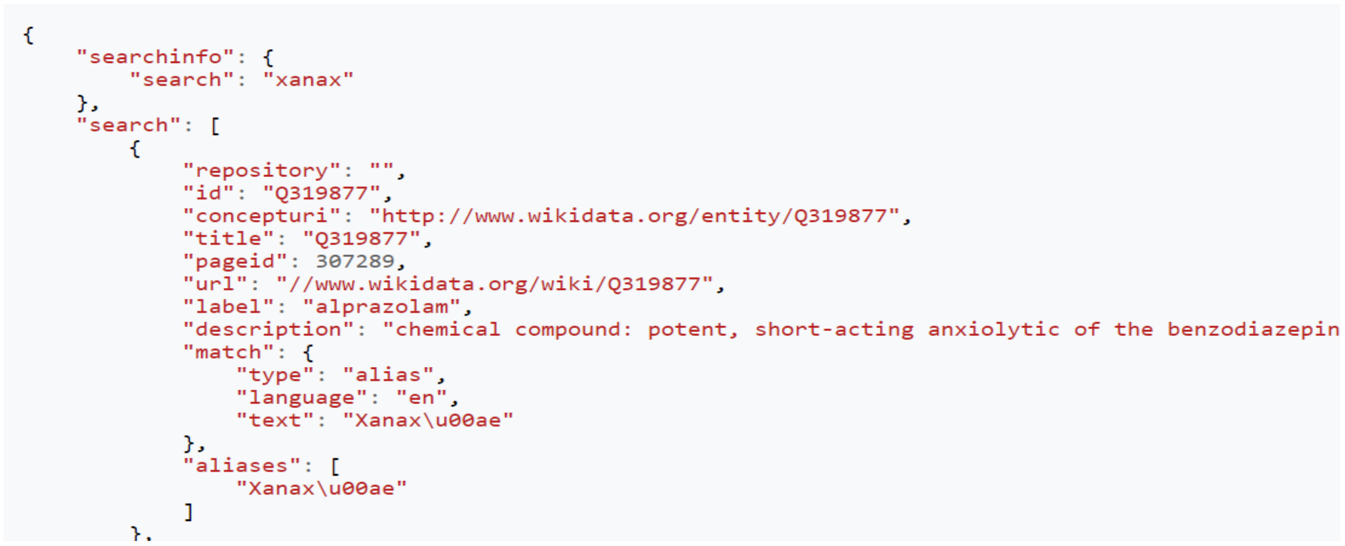


```
body = {
    "query": {
        "more_like_this" : {
            "fields" : ["selftext"],
            "like" : f"{postToFeedSelfText}",
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

Exercise 3

In this exercise, the problem to solve talks about obtaining a list of medications used and/or mentioned by users of reddit who appear in our post collection. In this case Wikidata has been used as a source of information to distinguish medications from other terms that might get retrieved on accident. The information is retrieved using json format from these two URLs:

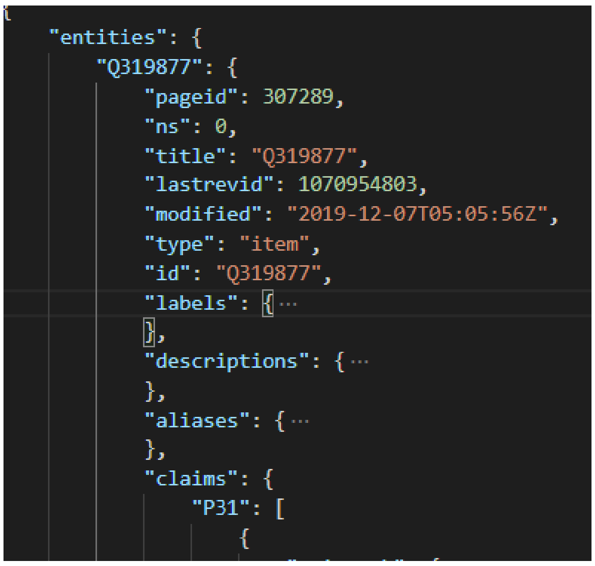
- https://www.wikidata.org/w/api.php?action=wbsearchentities&search=%SEARCH%&language=en&format=json
    - To search for the actual word retrieved by the first issued query which will be mentioned later. This URL allows us to include a search word or phrase as a parameter.
    - It is also configured with a “format” parameter to transform the output to json format.
    - “%SEARCH%” string has been placed there to allow the script to later on the code replace it by currently iterated term to check. It returns a list of possible registered items in their databases that refer to the provided search string.
    - These items in such list contain a code that will be used later to verify its precedence using second mentioned query.



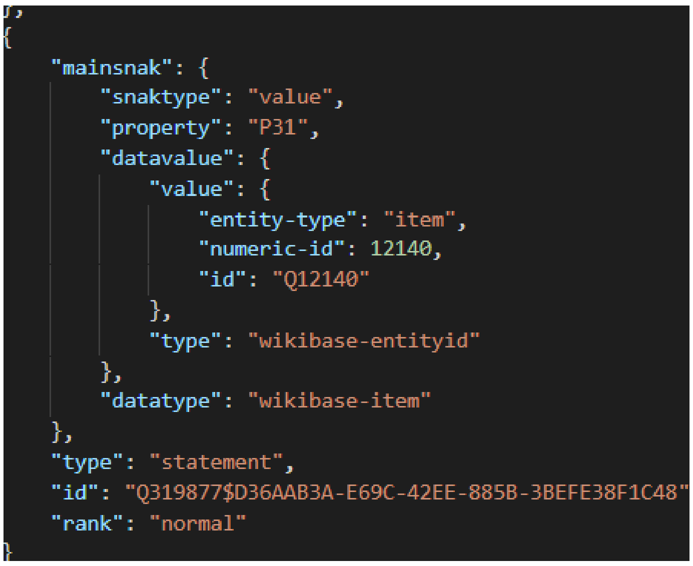
```
{
    "searchinfo": {
        "search": "xanax"
    },
    "search": [
        {
            "repository": "",
            "id": "Q319877",
            "concepturi": "http://www.wikidata.org/entity/Q319877",
            "title": "Q319877",
            "pageid": 307289,
            "url": "//www.wikidata.org/wiki/Q319877",
            "label": "alprazolam",
            "description": "chemical compound: potent, short-acting anxiolytic of the benzodiazepin
            "match": {
                "type": "alias",
                "language": "en",
                "text": "Xanax\u00ae"
            },
            "aliases": [
                "Xanax\u00ae"
            ]
        },
```

- https://www.wikidata.org/wiki/Special:EntityData/%ITEM_ID%.json
    - Given an item id (“Q[0-9]+”), this query returns the structured data for the associated item in json format. These ids are obtained from the previous URL.
    - This structure also includes the Wikidata classes from which such item is a subclass. We look for P31 property in this output file which retains the “instance of” information. Object properties information is held inside “claims” key in obtained json file. We are interested in a specific class which is “medication”. Its id is: “Q12140”

- Same as in previous query, “%ITEM_ID%” string is used as an anchor to replace by the id of the item to request to Wikidata.



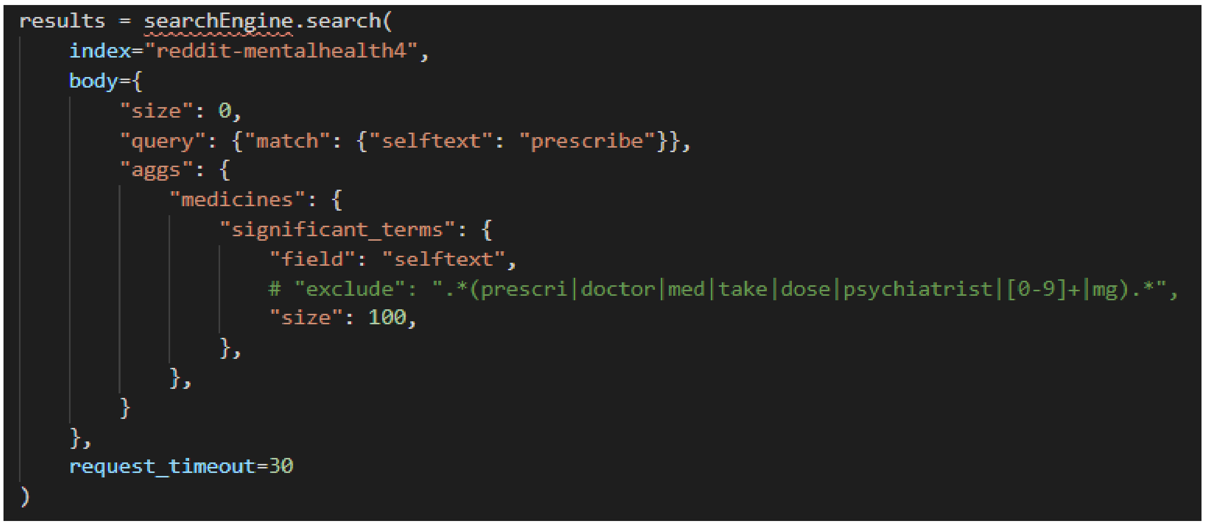
```
{
    "entities": {
        "Q319877": {
            "pageid": 307289,
            "ns": 0,
            "title": "Q319877",
            "lastrevid": 1070954803,
            "modified": "2019-12-07T05:05:56Z",
            "type": "item",
            "id": "Q319877",
            "labels": {···
            },
            "descriptions": {···
            },
            "aliases": {···
            },
            "claims": {
                "P31": [
                    {
```



```
},
{
    "mainsnak": {
        "snaktype": "value",
        "property": "P31",
        "datavalue": {
            "value": {
                "entity-type": "item",
                "numeric-id": 12140,
                "id": "Q12140"
            },
            "type": "wikibase-entityid"
        },
        "datatype": "wikibase-item"
    },
    "type": "statement",
    "id": "Q319877$D36AAB3A-E69C-42EE-885B-3BEFE38F1C48"
    "rank": "normal"
}
```

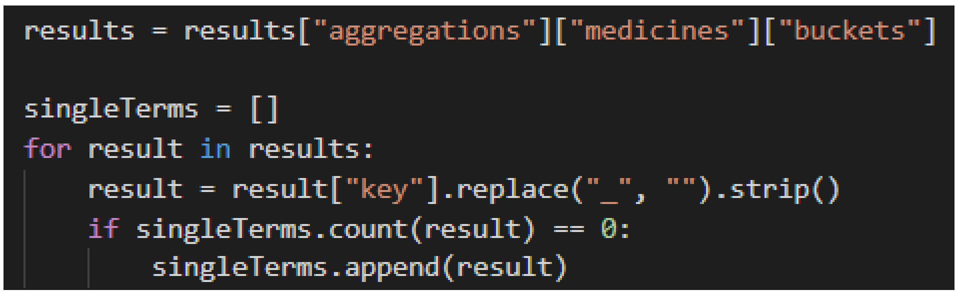
To solve this exercise, these steps are followed:

1. The process starts by issuing the query included below. The starting word to retrieve posts from the index is “prescribe”. Although it is not any piece of expert knowledge, it would propose several posts with the context of prescribing medicine/medications and would so represent a solid starting point. There could also be more terms to include in the search, but for the sake of the demonstration, this is enough. It can also be seen that an aggregation has been included to group the most common words in such posts. These might contain actual medications but also terms that are not actual medicines. Which will be filtered down in the script; more on that later.



```
results = searchEngine.search(
    index="reddit-mentalhealth4",
    body={
        "size": 0,
        "query": {"match": {"selftext": "prescribe"}},
        "aggs": {
            "medicines": {
                "significant_terms": {
                    "field": "selftext",
                    # "exclude": ".*(prescri|doctor|med|take|dose|psychiatrist|[0-9]+|mg).*",
                    "size": 100,
                },
            },
        }
    },
    request_timeout=30
)
```

2. Once we obtained such terms, we need to obtain unique terms because they might be repeated inside the results from above query.



```
results = results["aggregations"]["medicines"]["buckets"]

singleTerms = []
for result in results:
    result = result["key"].replace("_", "").strip()
    if singleTerms.count(result) == 0:
        singleTerms.append(result)
```

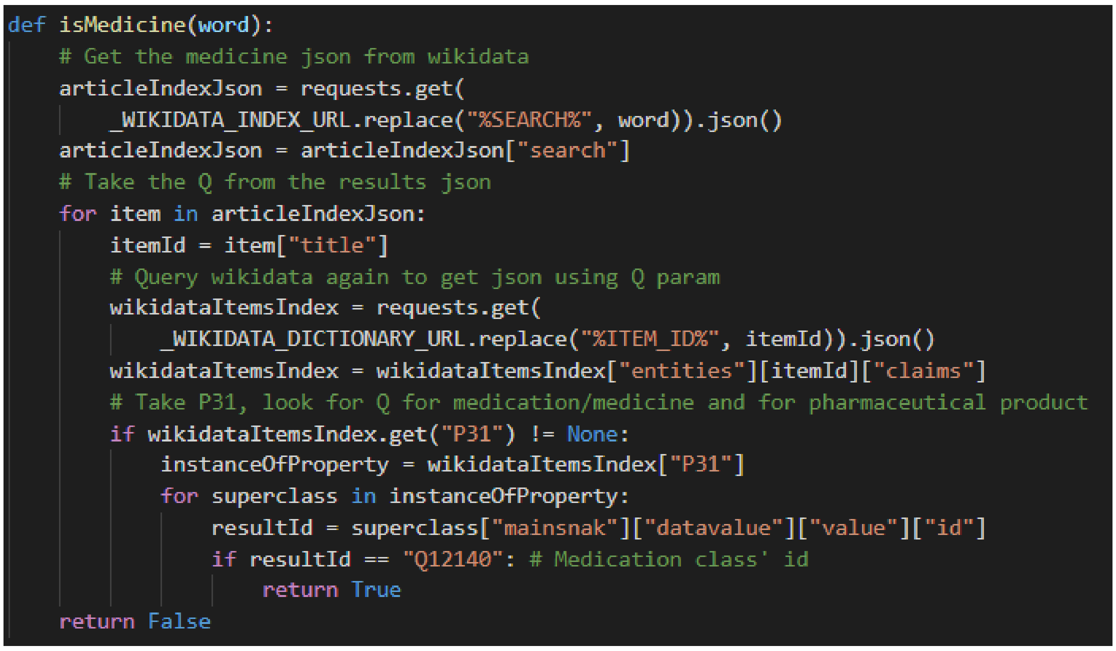
3. Filtering process can now be started. This is done through a function that checks whether a term is a medicine. This performs several operations.

    a. Firstly, requests data from Wikidata related to the word passed as a parameter. Once obtained, the ids may be processed.

    b. Secondly, al the ids returned by the request in GET mode are iterated through and serve as parameters for the second mentioned URL.

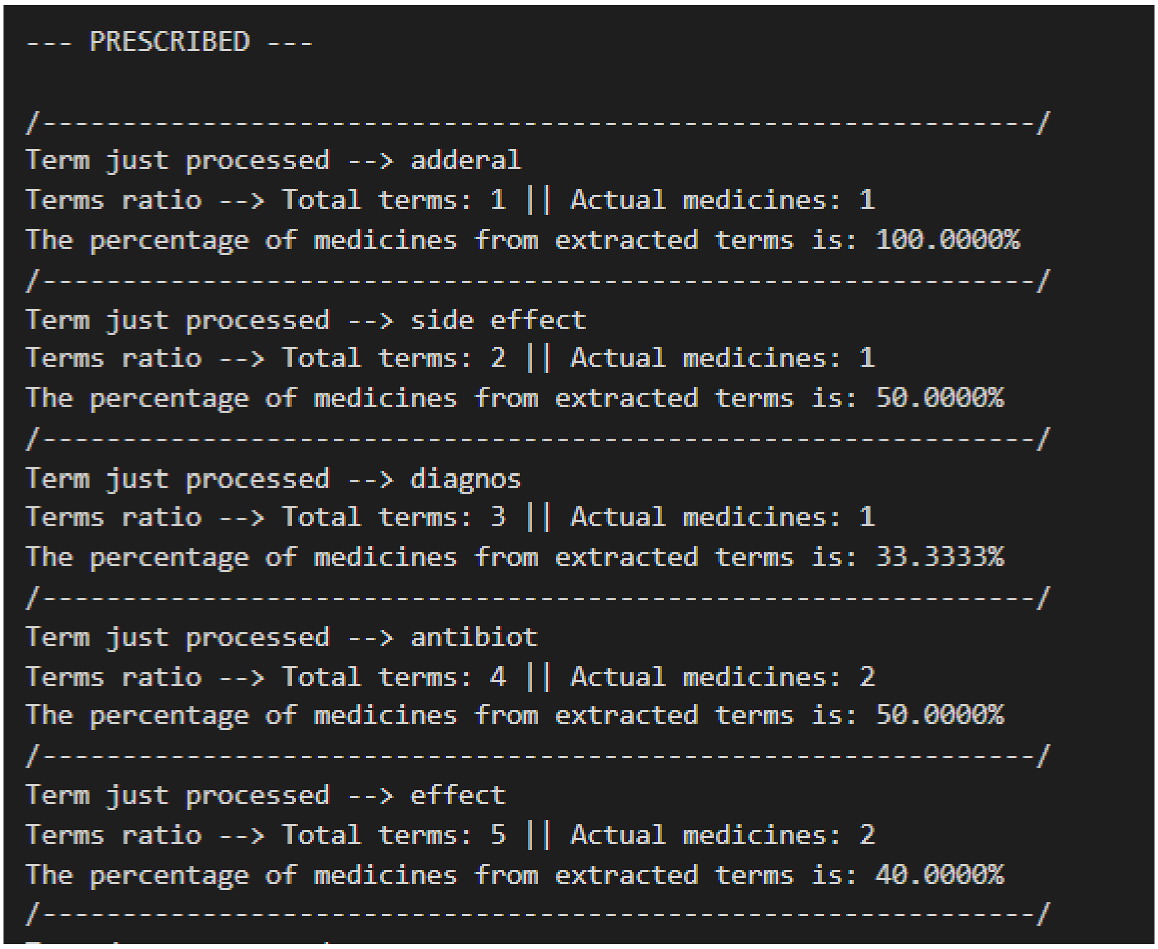
    c. Thirdly, given an id, yet another request in GET mode is issued to Wikidata so information from the object can be obtained. Once obtained, “P31” property can be looked up.

    d. And lastly, if the index has returned data referring to the instance of class “medication” inside mentioned property (id --> Q12140) in any of the search results, then the function returns True. Otherwise, when all are processed, if none contained such instance, the function returns False.



```
def isMedicine(word):
    # Get the medicine json from wikidata
    articleIndexJson = requests.get(
        _WIKIDATA_INDEX_URL.replace("%SEARCH%", word)).json()
    articleIndexJson = articleIndexJson["search"]
    # Take the Q from the results json
    for item in articleIndexJson:
        itemId = item["title"]
        # Query wikidata again to get json using Q param
        wikidataItemsIndex = requests.get(
            _WIKIDATA_DICTIONARY_URL.replace("%ITEM_ID%", itemId)).json()
        wikidataItemsIndex = wikidataItemsIndex["entities"][itemId]["claims"]
        # Take P31, look for Q for medication/medicine and for pharmaceutical product
        if wikidataItemsIndex.get("P31") != None:
            instanceOfProperty = wikidataItemsIndex["P31"]
            for superclass in instanceOfProperty:
                resultId = superclass["mainsnak"]["datavalue"]["value"]["id"]
                if resultId == "Q12140": # Medication class' id
                    return True
    return False
```

4. Finally, to follow the process, the script logs the operations performed and its results as it progresses through the terms obtained in the first place. Then this log is dumped to a file at the end of the execution for further reference. It should be noted that the success rate is logged in each iteration of the term checking.



```
--- PRESCRIBED ---

/-------------------------------------------------------------/
Term just processed --> adderal
Terms ratio --> Total terms: 1 || Actual medicines: 1
The percentage of medicines from extracted terms is: 100.0000%
/-------------------------------------------------------------/
Term just processed --> side effect
Terms ratio --> Total terms: 2 || Actual medicines: 1
The percentage of medicines from extracted terms is: 50.0000%
/-------------------------------------------------------------/
Term just processed --> diagnos
Terms ratio --> Total terms: 3 || Actual medicines: 1
The percentage of medicines from extracted terms is: 33.3333%
/-------------------------------------------------------------/
Term just processed --> antibiot
Terms ratio --> Total terms: 4 || Actual medicines: 2
The percentage of medicines from extracted terms is: 50.0000%
/-------------------------------------------------------------/
Term just processed --> effect
Terms ratio --> Total terms: 5 || Actual medicines: 2
The percentage of medicines from extracted terms is: 40.0000%
/-------------------------------------------------------------/
```

Disclaimer: This screenshot has been taken from the first execution output of the script created. Where some words were excluded after experience with obtained terms. As we didn't know whether this was allowed or not, we decided to leave it commented out, so the results were genuinely original and unmodified by previous experience. Both files, with exclusion of terms and without any exclusion have been included in the folder for this exercise's solution; just in case they need to be checked. It must be also said that, obviously, the success rate increased a little bit when excluding undesired terms.

Hence, this means that the query used to search for terms could be refined as the program progresses by updating it with terms that don't represent actual medicines.

But we thought that it might spoil the purpose of this exercise.

Exercise 4

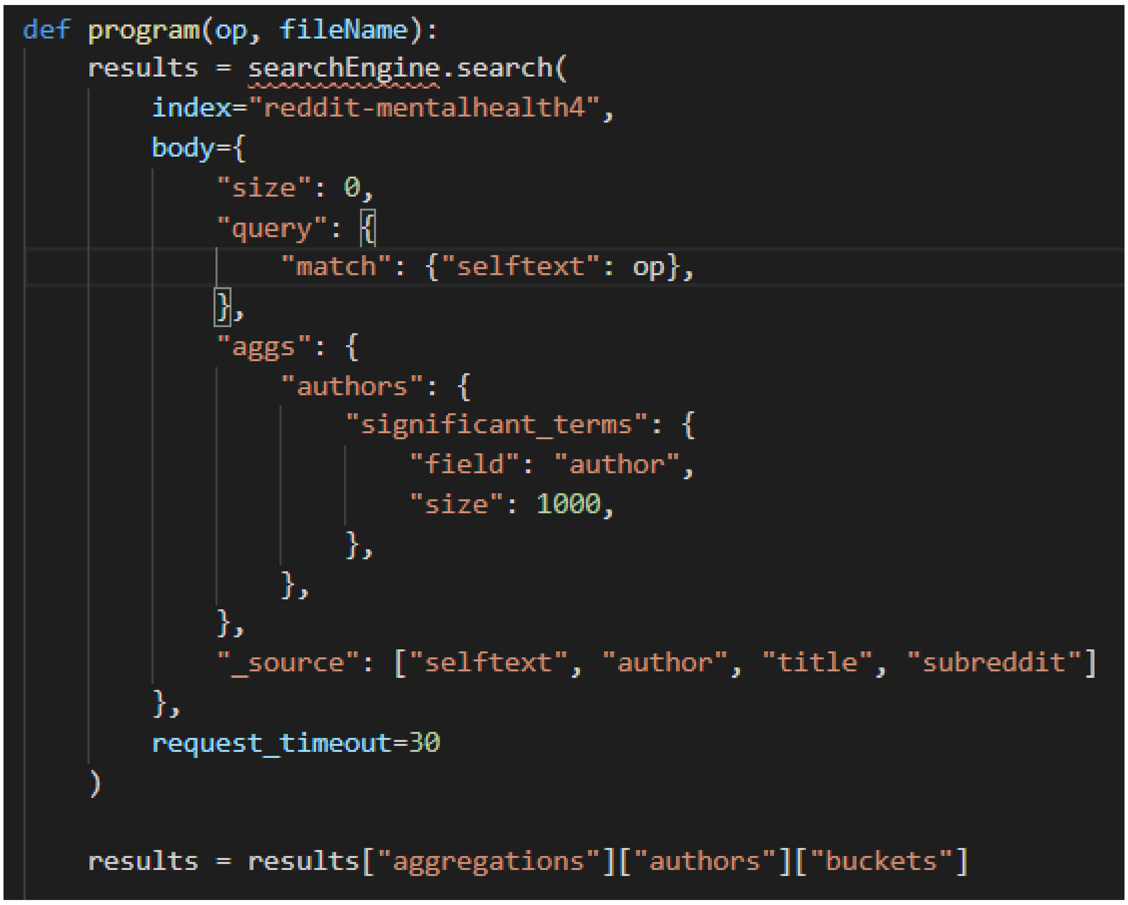
In this exercise, comorbidity factors must be retrieved from the index. Factors for comorbidity of suicidal ideation and self-injuring behaviours with other conditions must be extracted from the collections of posts.

As said in the statement, initial query points have been used for the first request.

- Suicidal ideations: “suicide OR suicidal OR kill myself OR killing myself OR end my life”
- Self-injuring behaviours: “self harm”

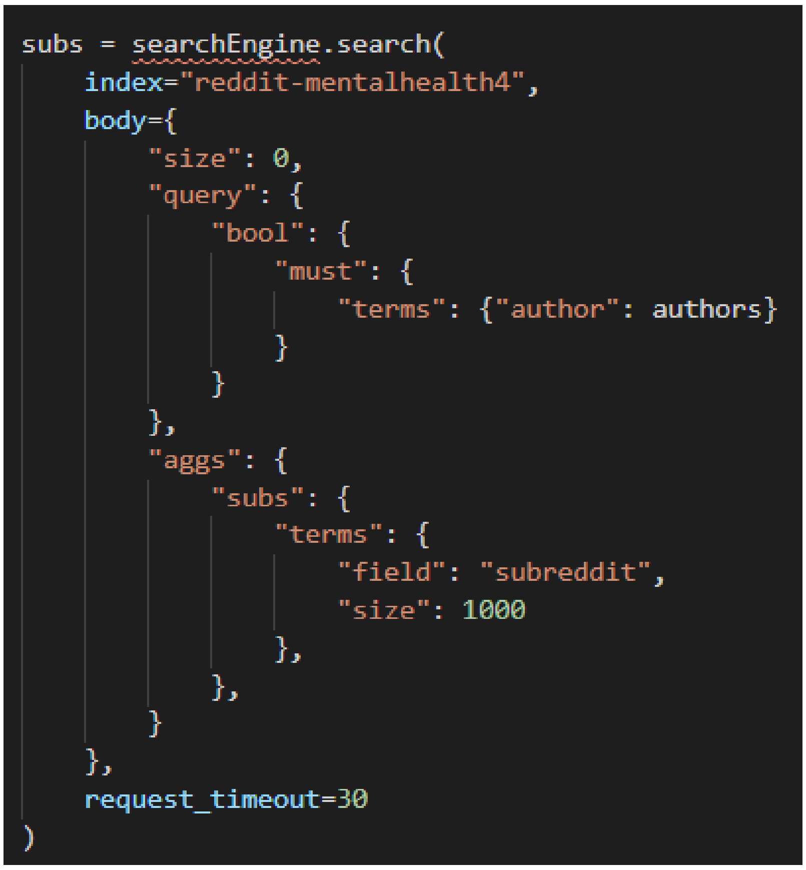
These steps have been followed to solve this exercise:

1. This first query is issued to obtain posts related to the parameter passed to the function, which contains above mentioned queries. In addition to this, an aggregation over posts authors has been included. This way we can isolate people to later process them in a different manner; more on this later.



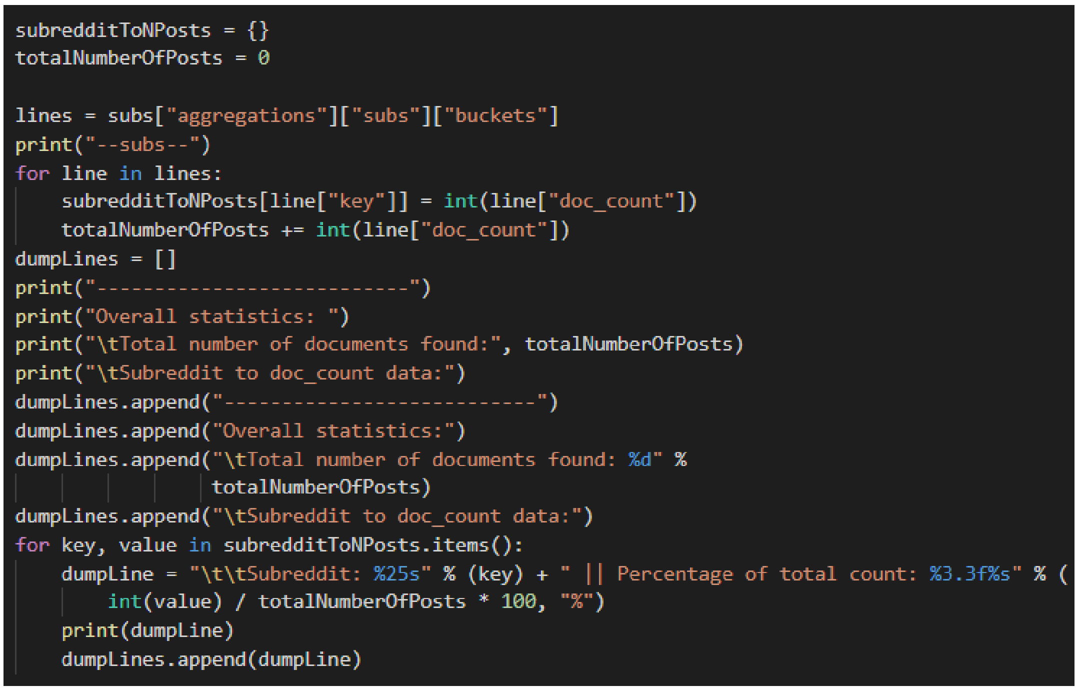
```
def program(op, fileName):
    results = searchEngine.search(
        index="reddit-mentalhealth4",
        body={
            "size": 0,
            "query": {
                "match": {"selftext": op},
            },
            "aggs": {
                "authors": {
                    "significant_terms": {
                        "field": "author",
                        "size": 1000,
                    },
                },
            },
            "_source": ["selftext", "author", "title", "subreddit"]
        },
        request_timeout=30
    )

    results = results["aggregations"]["authors"]["buckets"]
```

2. Once the different authors have been extracted, the subreddits in which such person has posted something are retrieved with this query. It aggregates the subreddit names to then count how many documents are obtained from each one. This allows the script to later generate some statistics about the information taken from the collection. The reason for this will be explained down below.



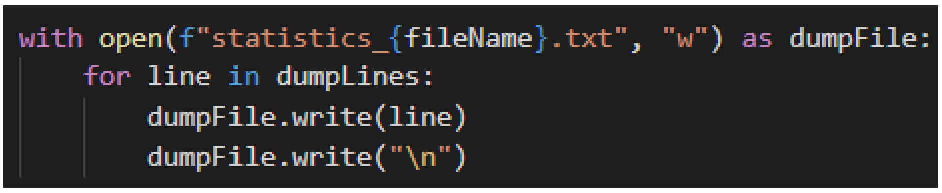
```
subs = searchEngine.search(
    index="reddit-mentalhealth4",
    body={
        "size": 0,
        "query": {
            "bool": {
                "must": {
                    "terms": {"author": authors}
                }
            }
        },
        "aggs": {
            "subs": {
                "terms": {
                    "field": "subreddit",
                    "size": 1000
                },
            },
        }
    },
    request_timeout=30
)
```

3. Now, it is an easy process to count how many posts per subreddit are. Some dictionaries are used and some log information to monitor the process. This indicates the percentage of documents for subreddit found for the authors retrieved. Subreddit to percentage of documents retrieved from the collection.



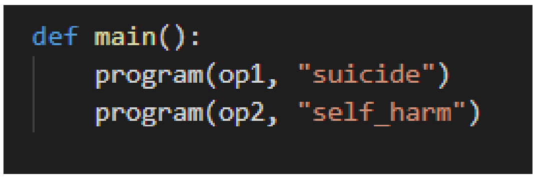
```
subredditToNPosts = {}
totalNumberOfPosts = 0

lines = subs["aggregations"]["subs"]["buckets"]
print("--subs--")
for line in lines:
    subredditToNPosts[line["key"]] = int(line["doc_count"])
    totalNumberOfPosts += int(line["doc_count"])
dumpLines = []
print("---------------------------")
print("Overall statistics: ")
print("\tTotal number of documents found:", totalNumberOfPosts)
print("\tSubreddit to doc_count data:")
dumpLines.append("---------------------------")
dumpLines.append("Overall statistics:")
dumpLines.append("\tTotal number of documents found: %d" %
                 totalNumberOfPosts)
dumpLines.append("\tSubreddit to doc_count data:")
for key, value in subredditToNPosts.items():
    dumpLine = "\t\tSubreddit: %25s" % (key) + " || Percentage of total count: %3.3f%s" % (
        int(value) / totalNumberOfPosts * 100, "%")
    print(dumpLine)
    dumpLines.append(dumpLine)
```

4. As the previous step progressed, the log information was being added to a list. So later it can be dumped to a file for later reference:



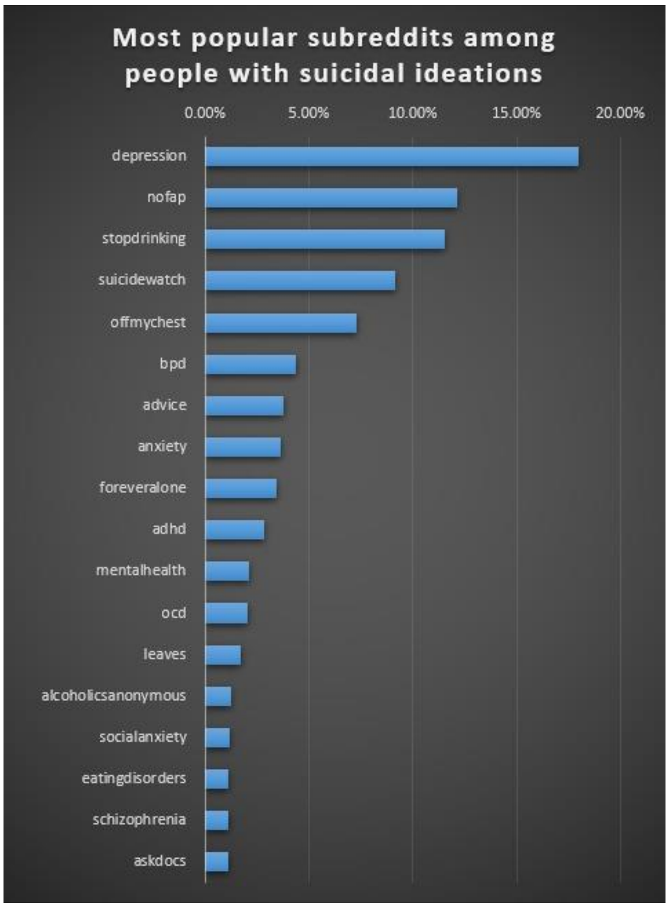
```python
with open(f"statistics_{fileName}.txt", "w") as dumpFile:
    for line in dumpLines:
        dumpFile.write(line)
        dumpFile.write("\n")
```

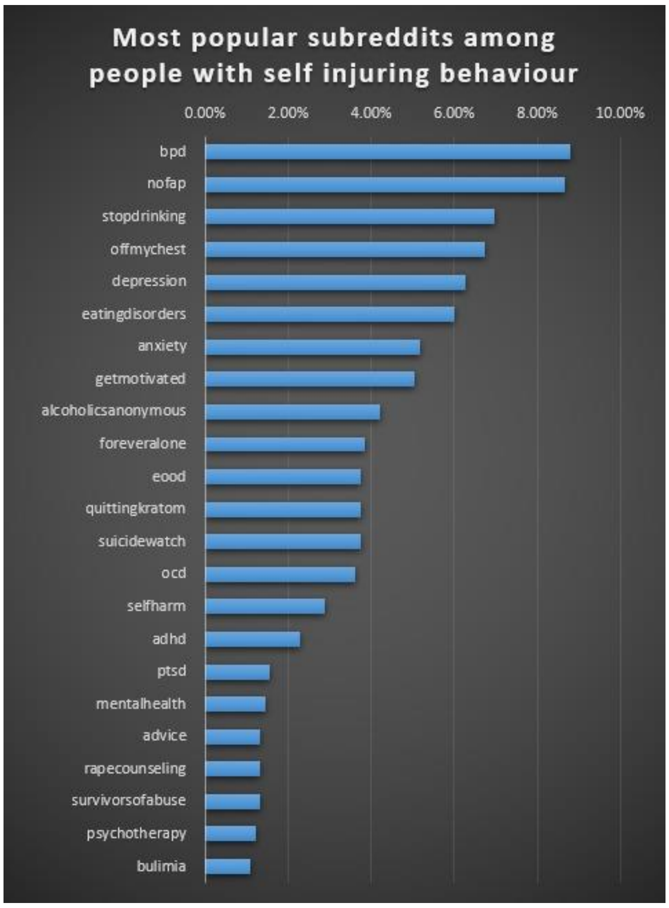
5. The main function executes previously explained steps for the two requested conditions.



```python
def main():
    program(op1, "suicide")
    program(op2, "self_harm")
```

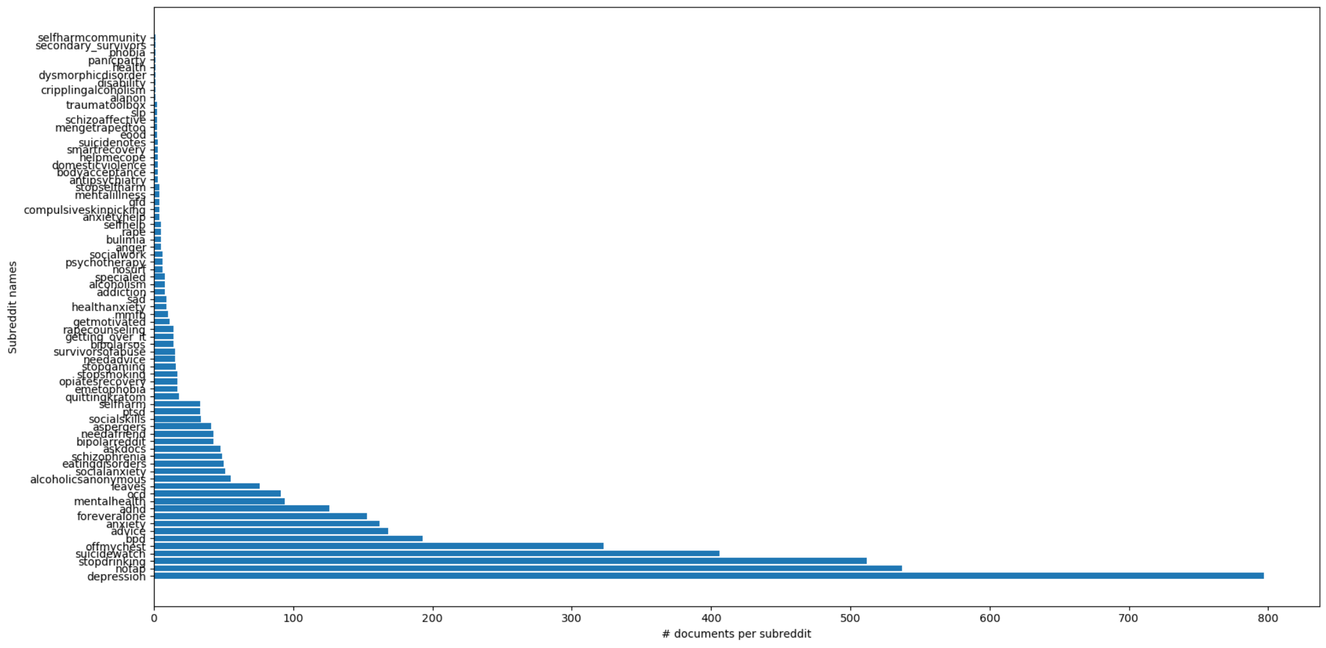
Some graphs ilustrating the results can be seen here:

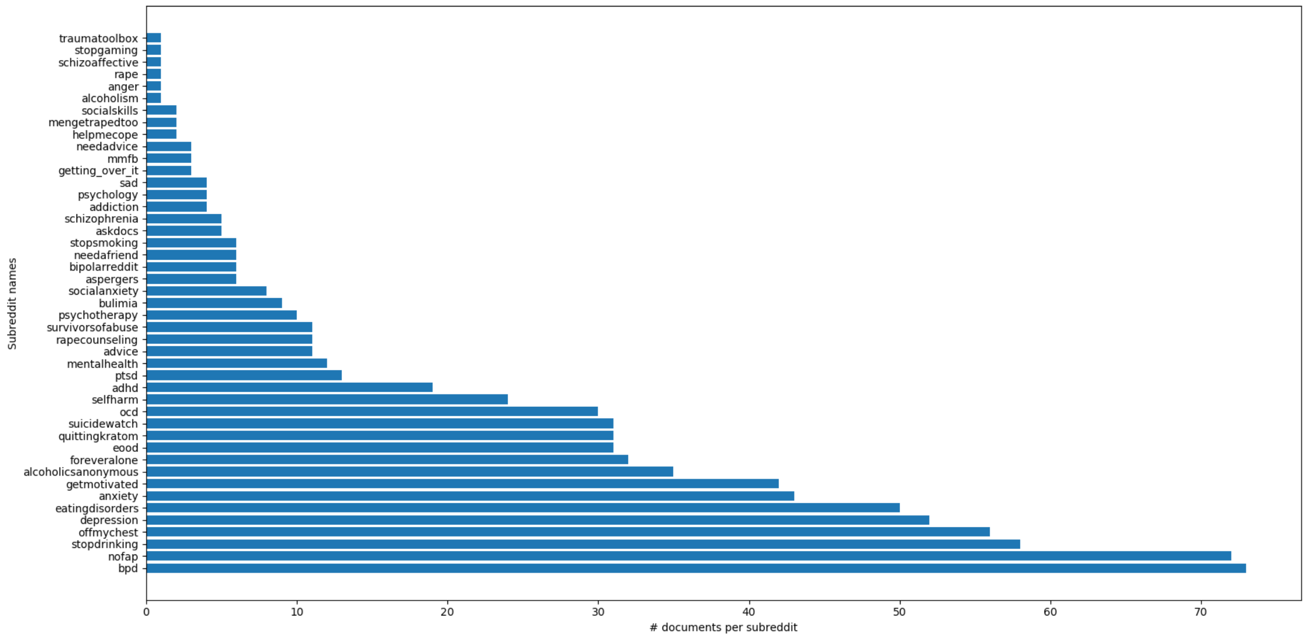




These graphics have been created without including the subreddits with less than 1 percent of total document count representation. This way the graphic is way clearer to a simple sight. These represent each subreddit to the percentage of documents that talk about such topic inside the whole sub-collection of documents retrieved.

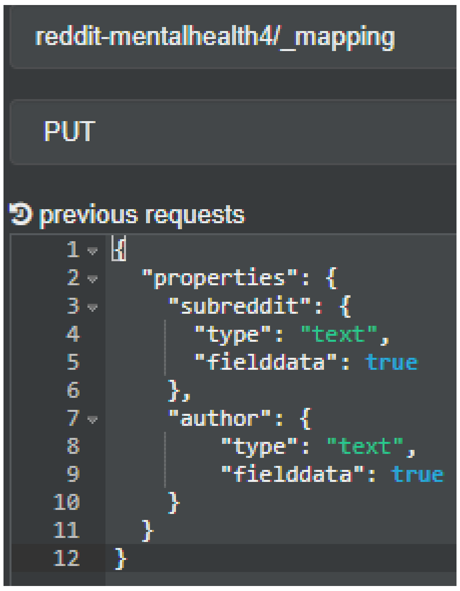
The python script solution to this exercise also plots both figures but including all subreddits, both for suicidal ideations and self-injuring behaviours:





These have been created using matplotlib python package.

To solve this exercise, a request has been sent to the index using Cerebro. It was used to update the mapping so that fielddata was set to true. The "PUT" protocol was used and the body is this one to the right.



```
{
  "properties": {
    "subreddit": {
      "type": "text",
      "fielddata": true
    },
    "author": {
        "type": "text",
        "fielddata": true
    }
  }
}
```

To conclude and completely explain the solution to this exercise, the name of the subreddit has been assumed to correctly define the contents of the posts that have been considered. This is thanks to the assumption made that off-topic posts are removed from reddit subreddits by moderators.

It has also been assumed that the name of the subreddits can represent the conditions to associate with either suicidal ideations or self-injuring behaviours, depending on each case. These comorbidity factors have been later checked using internet, expert knowledge from The Wikipedia. Some are correct, some are not. This one for instance, talks about depression and suicidal behaviour being combined.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2889009/

The scripting of this process has been studied briefly using Publish or Perish's json format export functionality. But automation has presented a series of issues which are out of the scope of this report. Above article was found using this software.

Epilogue

Further information

This is the link to the GitHub repository that holds all the files used during the development of this project. Just in case some reference/checking is needed.

https://github.com/fincamd/RI_ElasticSearch_Teamwork