PROJECT 1: FINDING SIMILAR ITEMS

Greta Angolani

December 2023

1 Introduction

For my project I've chosen the MeDAL (Medical Abbreviation Disambiguation Dataset for Natural Language Understanding Pretraining) dataset, published on Kaggle in 2020. This file contains the full abbreviation disambiguation dataset, and is structured in three columns: TEXT, LABEL, LOCATION, with approximately 14.4 million values for each column. The TEXT column is filled with "The normalized content of an abstract", the LABEL column with "The word at that was substituted at the given location", and the LOCATION column with "The location (index) of each abbreviation that was substituted".

TEXT The normalized content of an abstract	LOCATION The location (index) of each abbreviation that was substituted	LABEL The word at that was substituted at the given location
14391698 unique values	**5589758** unique values	**study** 2% **after** 1% Other (13984170) 97%
alphabisabolol has a primary antipeptic action depending on dosage which is not caused by an alterat...	56	substrate
a report is given on the recent discovery of outstanding immunological properties in ba ncyanoethyle...	24|49|68|113|137|172	carcinosarcoma|recovery|reference|recovery|after|plaque
the virostatic compound nndiethyloxotetradecylimidazolidinylethylpiperazinecarboxamidehydrochloride ...	55	substrate
rmi rmi and rmi are newly synthetized nrdibenzobfoxepinylnmethylpiperazinemaleates which show intere...	25|82|127|182|222	compounds|compounds|inhibitory|lethal doses|catecholamines

Figure 1: MeDAL Dataset

1.1 Data Organization

Since the project is about developing a detector of pairs of similar items contained in the TEXT column, I've selected the entire amount of data from this column, which exactly are 14.393.619 values of which 14.391.698 are unique. The first part of the project consists in installing Apache Spark and Kaggle, loading the dataset and displaying the data, adding to the dataset the column doc_id to uniquely identify every single value contained in the TEXT and LABEL column, while the LOCATION column has been dropped since not used for the project. Then, before starting with data pre-processing, I've also dropped the LABEL column, because useless for the first part of the project, and I have selected a sample of 50.000 data to start and then I have tested the code on an increasing subset, using 100.000, 150.000, 200.000 and 250.000 values. The selected sample is small considering the 14.393.619 values in the dataset, but

taking bigger sample took me a lot of time to print out the results.

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import CountVectorizer,
    StopWordsRemover
from pyspark.sql.functions import split, udf, explode, size,
    monotonically_increasing_id, max, regexp_replace
from pyspark.sql.types import ArrayType, StringType,
    IntegerType

# Starting Spark Session
spark = SparkSession.builder.appName("Similar_Items").
    getOrCreate()
df = spark.read.csv("full_data.csv", header=True)
df = df.select ('TEXT', 'LABEL').withColumn("doc_id",
    monotonically_increasing_id())
df_new = df.select('TEXT', 'doc_id')
df_new.show()
```

2 Data Pre-Processing

Starting with my project, the first thing I did was to "clean" the text to standardize and process it, so that similar content is represented in a similar manner, in order to not find similar pairs that are not valuable. So I've transformed capitalized text in lower text, removed stop words like "and", "is", "or", because they are very common and could have been retrieved without adding any meaning to the search, and could have lead falsely to the detection of pairs that are not similar. Here the explanation of what I did:

1. ```
   subset_df = subset_df.withColumn("processed_text", split(
       regexp_replace(subset_df["TEXT"], '[^\w\s]', ''), " ")
       )
   ```

   This code line tokenizes the text in the "TEXT" column of the Data Frame subset_df. First, it uses the regexp_replace function to remove any character that is not a word character (\w) or a white space (\s). This essentially strips out punctuation and other non-alphanumeric characters. After the cleaning, the text is split into tokens (words) based on spaces using the split function. The result is stored in a new column called "processed_text". By breaking text into individual tokens (or words), we transform free-flowing text into a structured format that can be analyzed, while removing punctuation simplifies the data.

2. ```
   @udf(ArrayType(StringType()))
   def to_lowercase(text):
       return [word.lower() for word in text]
   ```

```
subset_df = subset_df.withColumn("processed_text",
    to_lowercase(subset_df["processed_text"]))
```

The user-defined function (UDF) to_lowercase converts every word in a list to its lowercase equivalent. This is then applied to the "processed_text" column, ensuring that all tokens are in lowercase. This ensures that words like "The" and "the" are treated as the same.

3. ```
   remover = StopWordsRemover(inputCol="processed_text",
       outputCol="filtered_text")
   subset_df = remover.transform(subset_df)
   ```

   This section uses the StopWordsRemover from Spark's MLlib to remove common words like "and", "the", "is", etc., which don't contribute much to the meaning or uniqueness of a text. The result, after removing these stop words, is stored in a new column "filtered_text". By removing them, we reduce the dimensionality of the data and keep just the most informative words inside the TEXT column.

So, basically, pre-processing technique is important because lead us to:

1. Standardization
2. Dimensionality reduction
3. Keep the focus on the meaningful context

3 Finding Similar Items

We are entering now in the core part of the project: detection of similar items. The functions that I've applied are shingling, minhashing, LSH (Locality Sensitive Hashing) and Jaccard similarity.

3.1 Shingling

The most effective way to represent documents as sets, for the purpose of identifying lexically similar documents, that are string of characters. The shingling construct from a document the set of short strings that appear within it. If we do so, then documents that share pieces as short as sentences or phrases (depends on the k value used) will have many common elements in their sets, even if those sentences appear in different orders in the two documents. We define a k-shingle for a document to be any sub-string of length k found within the document. Then, I have associated with each document the set of k-shingles that appear one or more times within that document. Instead of using substrings directly as shingles, I have picked a hash function that maps strings of length k to some number of buckets and treat the resulting bucket number as the shingle. The set representing a document is then the set of integers that are bucket numbers of one or more k-shingles that appear in the document.
   ```

```
from pyspark.sql.functions import sort_array
k = 5
@udf(ArrayType(IntegerType()))
def shingling(text):
    return [hash(' '.join(text[i:i+k])) for i in range(len(
        text) - k + 1)]

subset_df = subset_df.withColumn("shingles", shingling(
    subset_df["filtered_text"]))
subset_df = subset_df.withColumn("sorted_shingles", sort_array
    (subset_df["shingles"]))
```

In the upper code I've imported sort_array, useful to sort the document-shingle pairs and order them by shingle. Then I've defined a shingling function that creates 5-shingles (k=5) for a given text. The value of k was chosen considering the length of the text in the dataset, and a length of 5 seems correct and the best one. Instead of using the shingles directly, I'm hashing them to integers (hash buckets) to save space and make the computation efficient. Then I simply added to my subset_df a new column named "shingles", and I've applied the "shingling function" to the "filtered_text" column, already in subset_df from the pre-processing step. The same explanation for the last line, I've created a new column, named "sorted_shinglese" in which are contained the values obtained applying the sort_array function to the "shingles" column in the subset_df from the line above.

3.2 Minhashing

Minhashing is an important probabilistic technique that has a connection with the Jaccard Similarity: the probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard Similarity of those sets. Thanks to the minhashing, we can compute the similarity between large sets in a more efficient way, without the need to compute the actual intersection and union sizes requested for the Jaccard Similarity, thanks to the transformation of the original high-dimensional data into a lower-dimensional representation (the minhash signature); this significantly reduces the amount of data that needs to be compared.

```
num_hashes = 100
hash_values = range(num_hashes)

@udf(ArrayType(IntegerType()))
def minhash_signature(sorted_shingles):
    signature = []
    for hash_val in hash_values:
        min_hash = float('inf')
        for shingle in sorted_shingles:
            hash_code = hash(f"{shingle}{hash_val}")
            min_hash = min(min_hash, hash_code)
        signature.append(min_hash)
```

```
    return signature
subset_df = subset_df.withColumn("minhash_signature",
    minhash_signature(subset_df["shingles"]))
subset_df.show()
```

I'm constructing a MinHash signature for each document using 100 different hash functions. For each hash function, I have combined the shingle with the hash function's index 'hash_val' to get different hashes. The minimum hash value across all shingles for each hash function becomes part of the document's signature. Then I have created a new column named 'minhashed_signature' that contains all the values generated by applying the minhash function on the "shingles" column in the subset_df.

3.3 LSH - Locality Sensitive Hashing

Locality Sensitive Hashing is a technique used for dimensionality reduction and approximate nearest neighbor search in large datasets. The primary goal of LSH is to maximize the probability that similar items are "hashed" into the same "buckets" with high probability while dissimilar items are hashed into different buckets. Thanks to Hash function the probability of collision is much higher for objects that are close to each other than those that are far apart. Hash tables store the items, where each unique hash code corresponds to a bucket, and similar items are likely to end in the same bucket. Parameters:

- b: The number of hash tables (or "bands") used. Increasing b typically increases the probability of catching similar items in at least one hash table
- r: The number of rows or hash functions within each band. The combination of b and r affects the sensitivity of the hashing: a higher r makes the hash more conservative (less false positives but possibly more false negatives)

The values of b and r are chosen considering a desired Jaccard threshold value, since we know that $t \approx \left(\frac{1}{b}\right)^{\frac{1}{r}}$.

How LSH works:

1. An object is described by a high-dimensional vector
2. This vector is divided into b pieces (bands)
3. Each band is hashed by some hash function
4. The resulting hash codes are used to hash the object into one bucket for each hash table

```
b = 25
r = 4
assert b * r == num_hashes

@udf(ArrayType(IntegerType()))
def lsh_banding(minhash_signature):
    bands = [minhash_signature[i:i + r] for i in range(0,
        num_hashes, r)]
    bucket_ids = [hash("".join(map(str, band))) for band in
        bands]
    return bucket_ids

subset_df = subset_df.withColumn("buckets", lsh_banding(
    subset_df["minhash_signature"]))
subset_df.show()
```

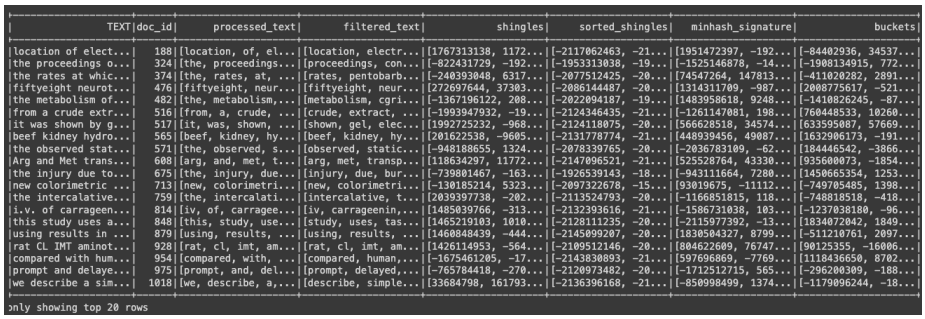
In the code, first of all I have defined the values of b and r, asserting that their product is equal to num_hashes, which is the length of the minhash_signature. The num_hashes chosen is 100, that is because I am analyzing a small subset of the entire dataset (maximum 250.000 values) and because it appeared to be the faster one. For b and r instead, I have tried two combinations:

1. b = 50 and r = 2, which produce a threshold of $t \approx 0.141$ and this means that the algorithm will consider documents to be similar with a lower level of similarity. This setting will result in more candidate pairs being identified as similar, leading to a higher number of false positives (but fewer false negatives).

2. b = 25 and r = 4, which produce a threshold of $t \approx 0.447$ that is higher than the previous one, meaning the algorithm will require a higher level of similarity for documents to be considered similar. This setting will result in fewer candidate pairs being identified as similar, leading to fewer false positives (but potentially more false negatives).

At the end I've chosen the second combination for my project, cause I think it's a good compromise between precision and recall, even if I am able to find fewer candidate pairs, where "precision" refers to the proportion of retrieved pairs that are actually similar to each other, while "recall" refers to the proportion of actual similar pairs that were successfully retrieved by the hashing process. Then I have defined (UDF) Function for Spark that returns an array of integers (ArrayType - IntegerType). Function lsh_banding: This function takes a minhash_signature (which is an array of integers representing the MinHash signature of an item) and applies the LSH banding technique. It splits the minhash_signature into chunks (or bands), where each chunk has a size of r elements. For each band, it converts the elements into strings, joins them, and then hashes this string to produce a bucket ID. This results in each band having its own hash value, which is used as a bucket ID where the original item will be placed.

Applying the UDF to a DataFrame: subset_df = subset_df.withColumn("buckets", lsh_banding(subset_df["minhash_signature"])): This line applies the lsh_banding function to the minhash_signature column of subset_df, a Spark Data Frame. The result is a new column called "buckets" that contains the bucket IDs for each record based on its MinHash signature. At the end I have printed out the data in a Data Frame along with the new buckets column, which will be used after to find candidate pairs for Jaccard Similarity.



TEXT	doc_id	processed_text	filtered_text	shingles	sorted_shingles	minhash_signature	buckets
location of elect...	188	[location, of, el...	[location, electr...	[1767313138, 1172...	[-2117062463, -21...	[1951472397, -192...	[-84402936, 34537...
the proceedings o...	324	[the, proceedings...	[proceedings, con...	[-822431729, -192...	[-1953313038, -19...	[-1525146878, -14...	[-1908134915, 772...
the rates at whic...	374	[the, rates, at, ...	[rates, pentobarb...	[-240393048, 6317...	[-2077512425, -20...	[74547264, 147013...	[-411020282, 2891...
fifteight neurot...	476	[fifteight, neur...	[fifteight, neur...	[272697644, 37303...	[-2086144487, -20...	[1314311709, -987...	[2008775617, -521...
the metabolism of...	482	[the, metabolism,...	[metabolism, cgri...	[-1367196122, 208...	[-2022094187, -19...	[1483958618, 9248...	[-1410826245, -87...
from a crude extr...	516	[from, a, crude, ...	[crude, extract, ...	[-1993947932, -19...	[-2124346435, -21...	[-1261147081, 198...	[760448533, 10260...
it was shown by g...	517	[it, was, shown, ...	[shown, gel, elec...	[1992725232, -968...	[-2124118075, -20...	[566628518, 34574...	[633595087, 57669...
beef kidney hydro...	565	[beef, kidney, hy...	[beef, kidney, hy...	[201622538, -9605...	[-2131778774, -21...	[448939456, 49087...	[1632906173, -191...
the observed stat...	571	[the, observed, s...	[observed, static...	[-948188655, 1324...	[-2078339765, -20...	[-2036783109, -62...	[184446542, -3866...
Arg and Met trans...	608	[arg, and, met, t...	[arg, met, transp...	[118634297, 11772...	[-2147896521, -21...	[525528764, 43330...	[935600073, -1854...
the injury due to...	675	[the, injury, due...	[injury, due, bur...	[-739001467, -163...	[-1926539143, -18...	[-943111664, 7280...	[1450665354, 1253...
new colorimetric ...	713	[new, colorimetri...	[new, colorimetri...	[-130185214, 5323...	[-2097322678, -15...	[93019675, -11112...	[-749705485, 1398...
the intercalative...	759	[the, intercalati...	[intercalative, t...	[2039397738, -202...	[-2113524793, -20...	[-1166851815, 118...	[-748818518, -418...
i.v. of carrageen...	814	[iv, of, carragee...	[iv, carrageenin,...	[1485039766, -313...	[-2132393616, -21...	[-1586731038, 103...	[-1237038180, -96...
this study uses a...	848	[this, study, use...	[study, uses, tas...	[1465219103, 1010...	[-2128111235, -20...	[-2115977392, -13...	[1834072042, 1849...
using results in ...	879	[using, results, ...	[using, results, ...	[1460848439, -444...	[-2145099207, -20...	[1830504327, 8799...	[-511210761, 2097...
rat CL IMT aminot...	928	[rat, cl, imt, am...	[rat, cl, imt, am...	[1426114953, -564...	[-2109512146, -20...	[804622609, 76747...	[90125355, -16006...
compared with hum...	954	[compared, with, ...	[compared, human,...	[-1675461205, -17...	[-2143830893, -21...	[597696869, -7769...	[1118436650, 8702...
prompt and delaye...	975	[prompt, and, del...	[prompt, delayed,...	[-765784418, -270...	[-2120973482, -20...	[-1712512715, 565...	[-296200309, -188...
we describe a sim...	1018	[we, describe, a,...	[describe, simple...	[33684798, 161793...	[-2136396168, -21...	[-850998499, 1374...	[-1179096244, -18...

only showing top 20 rows

Figure 2: Figure 2

In Figure 2, I have attached the output values after having applied Shingling, Minhashing, and LSH functions to my subset_df. As we can see the functions are correctly applied and the algorithms had produced the expected results.

3.4 Jaccard Similarity

Before applying Jaccard similarity some operations are necessary, as we can see in the following code:

```
subset_df = subset_df.dropDuplicates(['TEXT'])
exploded_df = subset_df.withColumn("bucket", explode(subset_df
    ["buckets"]))
candidate_pairs = exploded_df.groupBy("bucket").agg(
    array_distinct(collect_list("doc_id")).alias("doc_ids"))
candidate_pairs = candidate_pairs.filter(size(candidate_pairs
    ["doc_ids"]) > 1)
```

What I did was dropping the duplicate text in the TEXT column which was still in the sample, in order not to retrieve them while computing the Jaccard similarity. Using the "explode" function on the "buckets" column, in which each item is a list of bucket IDs, allows to create a new row for every ID in each list. This step is gathering all the documents that ended up in the same bucket for any band. I am grouping by the bucket ID and then aggregating all the doc_id that share that bucket ID into a list. The function array_distinct ensures that the list contains distinct document IDs (in case a document hashed to the same bucket in multiple bands). The purpose of this line is to filter out buckets that have only one document. We're interested in buckets that have

more than one document since these documents are potential candidate pairs that might be similar. Now I can apply Jaccard similarity: I started with a moderate threshold of 0.1 for a b = 50 and r = 2 in the LSH algorithm, and then when I have modified that values, I have tried with a threshold t = 0.3. But for the final version of my project I have used t = 0.1 to print out a little bit more candidate pairs.

```
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StructType, StructField,
    FloatType, LongType
from itertools import combinations
import pyspark.sql.functions as F
import time
from pyspark.sql.functions import least, greatest

start_time = time.time()
# Define the schema for the output of the combinations UDF
schema = ArrayType(StructType([
    StructField("doc1", LongType(), False),
    StructField("doc2", LongType(), False)
]))

# Define the UDF for generating combinations
@udf(schema)
def combinations_udf(doc_ids):
    return [(int(doc1), int(doc2)) for doc1, doc2 in
        combinations(doc_ids, 2)]

# Define a UDF for Jaccard similarity
@udf(FloatType())
def jaccard_similarity_udf(doc1_shingles, doc2_shingles):
    set1 = set(doc1_shingles)
    set2 = set(doc2_shingles)
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return float(intersection) / union if union != 0 else 0.0

threshold = 0.3
# Generate all pairs within each bucket
candidate_pairs = candidate_pairs.withColumn("doc_pair",
    explode(combinations_udf("doc_ids")))
candidate_pairs = candidate_pairs.withColumn(
    "min_doc_id",
    least(col("doc_pair.doc1"), col("doc_pair.doc2"))
).withColumn(
    "max_doc_id",
    greatest(col("doc_pair.doc1"), col("doc_pair.doc2"))
)

# Drop duplicates
```

```
candidate_pairs = candidate_pairs.dropDuplicates(["min_doc_id
    ", "max_doc_id"])
# Join with 'subset_df' to get the shingles for each document
    in the pair
candidate_pairs = candidate_pairs.alias("cp").join(
    subset_df.alias("df1"),
    col("cp.doc_pair.doc1") == col("df1.doc_id")
).join(
    subset_df.alias("df2"),
    col("cp.doc_pair.doc2") == col("df2.doc_id")
)

# Calculate Jaccard similarity for each pair
similar_pairs_df = candidate_pairs.select(
    col("doc_pair.doc1"),
    col("doc_pair.doc2"),
    jaccard_similarity_udf(col("df1.sorted_shingles"), col("
        df2.sorted_shingles")).alias("similarity")
).filter(col("similarity") >= threshold)

similar_pairs_df.show()

end_time = time.time()
elapsed_time = (end_time - start_time)/60
print(f"Computation time: {elapsed_time:.2f} minutes")
```

The first thing to do is define a schema for the output of a user-defined function (UDF). This schema describes an array of structures, each containing two integers representing document IDs. The UDF (combinations_udf) is defined to generate all possible combinations of document IDs in pairs; another UDF (jaccard_similarity_udf) calculates the Jaccard similarity between two sets of shingles, which is the size of the intersection divided by the size of the union of the sample sets. Then the document IDs within each bucket are exploded into all possible pairs using the combinations_udf. This results in a Data Frame (candidate_pairs) where each row represents a potential pair of similar documents. Duplicate pairs are then dropped to ensure each unique pair is only considered once in the output. This is done by creating min_doc_id and max_doc_id columns to standardize the order of document IDs in each pair and then dropping duplicates based on these columns. The candidate pairs are joined with another Data Frame (subset_df) that contains the shingles for each document. This is done twice, once for each document in the pair, to get the shingles for both documents in each pair. At the end, for each pair of documents, their Jaccard similarity is calculated using the jaccard_similarity_udf previously defined, and select pairs where the similarity is greater than or equal to the threshold chosen (0.3 in this case).

3.5 Label Analysis

The provided MeDAL dataset also has a column "LABEL", in which the documents of the "TEXT" column are classified. For one document of the TEXT column can correspond more than one value in the LABEL column. After having retrieved the couple of similar pairs in the previous analysis, I looked also for the label that they share. Since there are more than one, I've implemented the algorithm considering this, and so I've retrieved the couples that share at least one of the label contained in the LABEL column, and print out the shared ones.

```
from pyspark.sql.functions import collect_set

start_time = time.time()
# Explode the labels for each document and alias the DataFrame
df_labels = df.withColumn("label", explode(split(col("LABEL"),
    "\|"))).alias("df_labels")

similar_pairs_alias = similar_pairs_df.alias("sp")

# Join to get labels for doc1 and doc2, then filter for shared
    labels
# Use "alias" to avoid ambiguous reference to doc_id
similar_pairs_with_shared_labels = (similar_pairs_alias
    .join(df_labels, col("sp.doc1") == col("df_labels.doc_id")
        )
    .join(df_labels.select(col("doc_id").alias("doc_id2"), col
        ("label").alias("label2")), col("sp.doc2") == col("
        doc_id2"))
    .filter(col("df_labels.label") == col("label2"))
)

# Select distinct pairs and their shared label
distinct_pairs_with_labels = (similar_pairs_with_shared_labels
    .select(col("sp.doc1"), col("sp.doc2"), col("sp.similarity
        "), col("df_labels.label"))
    .distinct()
)
# To show all the shared labels together
combined_labels_df = (distinct_pairs_with_labels
                      .withColumn('doc_pair', F.array_sort(F.
                          array('doc1', 'doc2')))
                      .groupBy('doc_pair')
                      .agg(
                          F.collect_list('label').alias('
                              shared_labels'),
                          F.first('similarity').alias('
                              similarity')
                      ))
```

```
combined_labels_df.show(truncate = False)

end_time = time.time()
elapsed_time = (end_time - start_time)/60
print(f"Computation time: {elapsed_time:.2f} minutes")
```

The first thing was exploding labels from the Original Data Frame: df_labels is created by taking an existing Data Frame "df" and applying the explode and split functions to a column named "LABEL".Each document in df has a "LABEL" column, where labels are separated by the "|" character. The explode function splits the labels into individual elements and creates a new row for each label for a given document and return a new alias data frame df_labels. I also have created an alias similar_pairs_alias for the similar_pairs_df Data Frame, containing document pairs (doc1 and doc2) with similarity scores. Then I have performed a join between similar_pairs_alias and df_labels to map the labels for doc1 in each pair. It then joins with df_labels again (renaming columns to avoid ambiguity) to get the labels for doc2 in each pair. After that I have selected distinct rows from this filtered Data Frame, retaining the document IDs (doc1, doc2), their similarity score, and the shared label between these documents, and create a new data frame distinct_pairs_with_labels, that contains unique pairs of documents that share at least one label. At the end, a new column doc_pair is created by sorting and combining doc1 and doc2 and the Data Frame is then grouped by doc_pair. Aggregation is performed to collect all shared labels for each pair into a list (collect_list('label')) and to get the first similarity score (F.first('similarity')). The resulting Data Frame (combined_labels_df) contains each unique pair of documents, their list of shared labels, and their similarity score. In summary, this code identifies and isolates pairs of documents from similar_pairs_df that not only are similar based on a previous similarity measure but also share at least one common label. This also enforce even more their similarity, since they also share a classification label, and this means that they are classified under the same category.

4 Conclusion

The code that I have provided has been tested on different samples, starting with 50.000 (0.3 %),100.000 (0.6 %), 150.000 (1.0 %), 200.000(1.3 %) and 250.000 (1.7 %) values. The smaller the sample, the less the similar documents found. This seems to be correct, since in the dataset there are a lot of values (and lot of them are unique) and I am analyzing only a small random sample of it. Since the MeDAL dataset is really big, I could have not print out the results for a sample bigger than 250.000 values because it was taking a lot of time to print out the results of the Jaccard similarity and Label analysis which require more computational time, considering also the limited computation constraint that I had. After having done the Jaccard Similarity analysis and the Label analysis I have printed out some of the candidate pairs to check the results, and we can say that the results are the one we expected:

1. TEXT 1: "a short cut review was carried out to establish whether electrical stimulation had any advantages over facial exercises in promoting recovery T3 BP altogether papers were found using the reported search of which one presented the best evidence to answer the clinical question the author date and country of publication patient group studied study type relevant outcomes results and study weaknesses of this best paper are tabulated a clinical bottom L1 is stated" with labels: —after—bells palsy—line—

2. TEXT 2: "a short cut review was carried out to establish whether outpatient investigation of suspected PE is a safe strategy a total of papers were found using the reported search of which one presented the best evidence to answer the clinical question the author date and country of publication patient group studied study type relevant outcomes results and study weaknesses of this best paper are tabulated a clinical bottom line is stated" with labels: —pulmonary embolus —

These two texts have a similarity of 0.5, which maybe is a little bit lower than expected since the two texts are almost the same, but it can be considered quite good. In the last section I have also plotted two graphs: the first one for the computational time of Jaccard Similarity, and the second one for the computational time of Label Analysis for the four subset I have analyzed. As we can see the computational time is growing for both and there is a positive correlation between the sample size and the time taken for analysis; as the sample size increases, the time required for the analysis also increases. The distribution of points is not linear, as the time does not increase at a constant rate with the increase in sample size. There seems to be a more than proportional increase in time taken as the sample size increases, which could suggest a nonlinear relationship. The increases in time are not consistent, but they are also not so much different from one another. The same is true for the computational time of Label Analysis, with the only difference that takes more time for each subset compared to the Jaccard Similarity computational time. To conclude, the code implemented seems to work as expected, retrieving the similar pairs and giving a good value for similarity, but I think it could be optimized to work faster and to better scale up with the data.

5 Declaration

"I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study."