Algorithms for Massive Datasets project

Francesco Stella - 979050

Finding Similar Items using the StackSample dataset

This report explores one of the possible approaches aimed at finding similar items in the context of textual documents, specifically we analyze similar questions from the StackSample dataset.
In **section 1**, we briefly discuss the theory behind the chosen approach. In **section 2** we present some relevant details about the dataset and the preprocessing techniques used. In **section 3**, we discuss the practical implementation of the main procedure aimed at retrieving similar items in a scalable way and, finally, in **section 4** and **section 5** we present, respectively, some considerations on the scalability and the complexity of the implementation and some results obtained using the dataset.
The theoretical considerations discussed in section 1 mostly derive from the book "Mining of Massive Datasets" from Leskovec, Rajaraman and Ullman [3].

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.

1 Introduction

The following work is an implementation of the *Locality Sensitive Hashing (LSH)* algorithm for document similarity, in which the correspondence with the Jaccard similarity is used. This technique is based on a family of functions called *minhash functions*, which should provide *signatures* for our items. The idea is that similar items should have similar signatures. In this way it is possible to compare only those items that satisfy certain prerequisites and the comparison is fast due to the relatively small size of the signatures with respect to the original size of the items.

1.1 k-shingles, minhash functions and signatures

In order to define what a signature is, we must introduce the concepts of *k-shingle* and *minhash*. The former is, in general, a sequence of k characters, however in this work we consider it as a sequence of k words. In general, k can assume different values based on the task and, for our purposes, a value of 5 or 6 should be suitable.
Each document, i.e. each question in our dataset, contains a certain number of shingles. Ideally, similar questions should contain at least a prefixed ratio of identical shingles. Some preprocessing is needed before the detection of the shingles, however we can make a simple example as follows, e.g. for k=3:

{Jack, and, Mary}, {and, Mary, went}, {Mary, went, to}, ...

That is, we are sliding a window of 3 words over the textual data we want to compare. Please note that stopwords such as 'and' should be removed before extracting shingles.
After the extraction of the shingles from the document or text, we apply a sort of pre-hash to them in order to convert the shingles into an integer number, then we create a *characteristic matrix* in which the columns are the documents/texts and the rows are the pre-hashes of the shingles. An example of characteristic matrix comprising the shingles is the following:

Shingle	S1	S2	S3	S4
0	0	1	0	0
1	1	0	1	0
2	1	1	0	1
3	0	0	0	1

Now, we can define the minhash as follows: given a characteristic matrix, we apply an hash function to all the shingles, i.e. to each shingle present in any of the documents we want to compare. The obtained hash is the new row in the characteristic matrix associated to the corresponding shingle. In this way we are simulating permutations of the rows through an hash function. The hash function should be of the kind:

$$(ax + b) \mod p \tag{1}$$

where a and b are random integers, x is the pre-hash of a given shingle and p is the least prime number greater than n, where n is the total number of shingles. The minhash of a document is the number of the first row (of the characteristic matrix) containing a 1 in the column associated to the given document, considering the permuted order of the rows.
After having completed the hashing for all the shingles, we reiterate the procedure several times, using different hash functions and storing each time the minhashes in a new matrix. Upon the end, we obtain a sequence of minhashes for each document, representing the *signature* of the document itself.
This algorithm allows us to quickly estimate the Jaccard similarity between two documents by simply looking at their signatures. Indeed, it can be proved that there is a correspondence between the expected number of rows in which two documents agree and their Jaccard similarity.

1.2 The banding technique

Even if the signatures greatly reduce the size of a document, it may still be hard to compare all the documents for checking similarity. The banding technique solves this problem by dividing the signatures into b bands, each one having r rows. If two documents have identical signatures in any of the bands (i.e. we check their r-integers subsequences), then they are candidate pairs. Clearly, this could lead to both false positives and false negatives, although they should be rare. The main advantage of the banding technique is that we can tune a threshold that allows us to select the candidate pairs exceeding it with a probability of 1/2. This threshold is a function of b and r and it has always an *S-shape*, assuming that we have the probability of becoming a candidate on y-axis and the Jaccard similarity of the documents on the x-axis. The definition of the threshold is shown below:

$$t = \left(\frac{1}{b}\right)^{\frac{1}{r}} \tag{2}$$

In general, we can make some considerations on the banding technique. Let's say that we have two documents (signatures), D1 and D2, that we want to analyze, we have also b bands and r rows for the signature matrix and that the probability that the given documents agree on a particular row of the matrix is s. Then:

- The probability that documents agree on all rows of a given band is P(D1 and D2 agree on all rows of a given band) = s^r
- P(D1 and D2 disagree in at least one row of a given band) = 1 - s^r
- P(D1 and D2 disagree in at least one row of each band) = (1 - s^r)^b

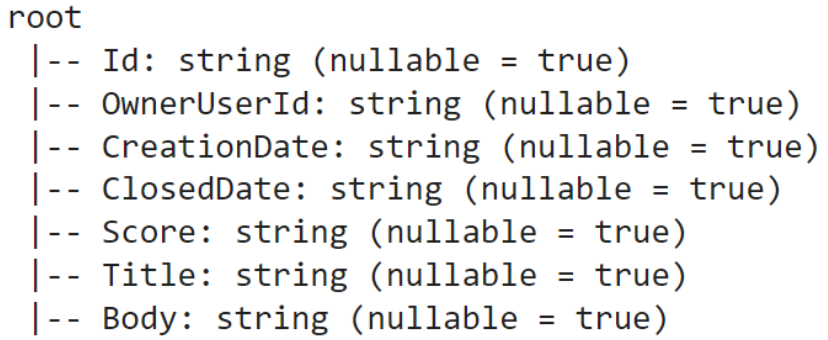

Figure 1: The schema of the dataset. We are specifically interested in the Body column, on which the algorithm is applied.

- P(D1 and D2 agree on all rows of at least one band) = (1 - (1 - s^r)^b)

2 Dataset and Preprocessing

2.1 The StackSample dataset

We consider the StackSample dataset, containing the text of 10% of the questions and answers from the Stack Overflow programming website and divided into three .csv files, specifically *Answers*, *Tags* and *Questions* [4]. We are interested in the latter and, in particular, in the *Body* column containing the text of the questions from the website. We also select the Id column and the Title one for visualization purposes upon the end. We can retrieve the schema of the dataset, as shown in Figure 1.

2.2 A few premises on the PySpark interface

Before we start to discuss the steps followed in the actual implementation, we outline few precautions adopted throughout the entire project. For the implementation of the algorithm we use PySpark, a Python interface for Apache Spark, providing us with most of the features of Spark. Since we want to leverage on the optimizations underlying the Spark engine, we try to avoid whenever possible the User Defined Functions (UDFs), that are functions specified by the user which typically are not as efficient as the counterpart provided by the Spark SQL module. This is due to the fact that whenever we call an UDF, a Python process is started on the worker, that serializes the data into a format understandable by the Python interpreter, generating some overhead. However, UDFs represent a powerful way to implement functions that are not already present in the standard Spark modules and they support several different languages. Hence, in some situations we will use them, although we maintain a

preference for the functions directly provided by the Spark modules.

2.3 Preprocessing

The first task we need to perform is preprocessing. This step is composed by two main subtasks. First, we note that the body of the questions may contain some *code* in any programming language, hence we should separate this part from the natural language. This should lead to more reliable estimates since the comparisons are separate for the two kinds of textual data. Actually, here we focus only on the natural language, since the comparison of the code sections (i.e. of the software) is a specialized task that requires special precautions, in which the simple comparison between words and comments in the source code leads to poor results [1]. Basically, we search for the <code> tag in the HTML text of the body of each question through a regular expression and then, if it is present, we remove the code section delimited by this tag. Figure 2 shows the code used for this task.

```
# Remove code tags from code parts
df_filt = df_filt.withColumn('Body_code',
                             regexp_replace('Body_code', '(<code>[\S\s]*?)|(</code>)', ''))
```

Figure 2: Code for the removal of the code sections from the body of the questions.

Then, in order to complete the preprocessing step, we need to remove the stopwords, since they do not provide us with useful information about the similarity between two documents. We use a predefined dataset of english stopwords provided by the *pyspark ml* package [1]. The following is the code used for the creation of the regex starting from the stopwords. In this way, we can efficiently match all the stopwords and all the single characters remained:

```
# Create regex from stopwords
stopwords_regex = '\\b)|(\\b'.join(englishStopwords)
stopwords_regex = '(\\b' + stopwords_regex + '\\b)' + '|(\\b[a-z]{1}\\b)'
```

Figure 3: Code for the creation of the regex aimed at removing stopwords.

3 Finding similar questions

In order to find similar questions in the StackSample dataset, we need to implement the LSH algorithm. Here, small variations are performed with the aim to

[1] pyspark StopWordsRemover

improve memory utilization. Of course, the general steps remain those presented in section 1.

3.1 Shingles extraction and hashing

After the preprocessing steps followed in section 2.3, we obtained the text of the questions without stopwords and punctuation. Now, we can extract the k-shingles from this textual data and we chose a value of 6 for the parameter k, which turns out to be a good one for email and other relatively short texts. We specified a UDF since we use an external Python module (i.e. *re*, for the regex expressions) and since it leads to a cleaner and more understandable code. The image below shows the function:

```
@udf(returnType=ArrayType(elementType=StringType()))
def generate_shingles(bodyCol):
    words = re.split(r'[^a-z]+', bodyCol)
    v = [' '.join([words[x] for x in range(i, i+K)]) for i in range(1, len(words)-K)]   # shingles from words
    return v
```

Figure 4: Function for the extraction of the shingles.

After the extraction of the k-shingles, we should apply a pre-hash in order to convert them into a numerical form, in this way we can then apply the different hash functions representing the permutations of the rows of the characteristic matrix. For the pre-hash, a variation of the *fnv1 non-cryptographic hash function*[2] is used, which should lead to rare collisions while maintaining good performances. Once the shingles are converted into numbers, we apply a specified number of hash functions (ideally large, although in our work was set to small numbers for testing purposes), each one representing a permutation of the characteristic matrix. The shingle IDs are used as the original ordering of the shingles, while the hash functions give us the new row associated to each shingle. In order to compute the hashes (see equation 1) an external implementation of the Miller-Rabin primality test [2] was retrieved to estimate the prime number p used in the function. In our implementation of the algorithm, we obtain a dataframe as shown in Figure 5.

Please note that here we don't have a characteristic matrix, since we only store the shingle IDs actually occurring in each document, instead of storing a 0 and a 1, respectively, for each absent and occurring shingle ID in that document. In this way we utilize less memory and the advantage is greater if the characteristic matrix would be sparse.

[2]FNV-1 hash function

Id	ShingleID	H0	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11
10230	9235	5887	3194	9763	1616	8343	1593	9042	2990	2302	6385	10649	3184
10230	9221	5859	2998	9385	1518	8273	1495	8818	2584	2050	5993	10327	2988
10230	10256	7929	4877	12108	8763	837	8740	156	7377	8069	9751	8910	4867
10230	4707	9442	2857	1006	7753	10925	7730	12260	10399	9075	5711	7393	2847
10230	7174	1765	12173	4560	12411	10649	12388	1288	6276	3037	11732	1079	12163
10230	9817	7051	11342	255	5690	11253	5667	5743	7257	167	10070	11424	11332
10230	923	1874	325	12337	6487	4616	6464	2160	1551	4018	647	8638	315
10230	421	870	5908	11394	2973	2106	2950	6739	12215	7593	11813	9703	5898
10230	460	948	6454	12447	3246	2301	3223	7363	735	8295	294	10600	6444
10230	12349	12115	8957	5564	10803	11302	10780	8422	5019	7910	5300	6605	8947
10230	11795	11007	1201	3217	6925	8532	6902	12169	1564	10549	2399	6474	1191
10230	991	2010	1277	1562	6963	4956	6940	3248	3523	5242	2551	10202	1267
10230	1436	2900	7507	966	10078	7181	10055	10368	3817	641	2400	7826	7497
10230	10140	7697	3253	8976	7951	257	7928	10911	4013	5981	6503	6242	3243
10230	10879	9175	988	3707	513	3952	490	10124	222	6672	1973	10628	978
10230	2751	5530	695	11249	6672	1145	6649	6186	4119	11700	1387	238	685
10260	7341	2099	1900	9069	969	11484	946	3960	11119	6043	3797	4920	1890
10260	1382	2792	6751	12119	9700	6911	9677	9504	2251	12280	888	6584	6741
10260	2788	5604	1213	12248	6931	1330	6908	6778	5192	12366	2423	1089	1203
10260	778	1584	10906	8422	5472	3891	5449	12451	9957	1408	9198	5303	10896

Figure 5: Dataframe representing the shingles in the permuted order. For each document ID, the shingles occuring in the document are indicated by ShingleID (pre-hash) and each H_i has the corresponding new row associated to the shingleID.

```
for i in range(len(a)):
    window = Window.partitionBy('Id').orderBy(f"H{i}")
    sig_mat = sig_mat.withColumn(f"minH{i}", min(f"H{i}").over(window)).drop(f"H{i}")
```

Figure 6: Code for the computation of the minhashes of each document.

3.2 Signatures computation

Once the hash functions are applied and the dataframe of figure 5 is retrieved, we can compute the minhashes of each document, i.e. the smaller hash value associated to each document ID, for each H_i. We specify that the smaller hash value associated to a given document corresponds to the index of the first row having a 1 in the original characteristic matrix, since if a shingle doesn't appear in that document, then it has no corresponding hash value.
Minhashes are computed for each document and for each hash function H_i through the code shown in Figure 6, in which a is the number of hash functions.

The resulting dataframe is represented in Figure 7, in which we have all the documents with their corresponding minhashes. Each row represents the signature of each document.

3.3 The banding technique

At this point we apply the banding technique, which is useful when the amount of data gets larger. Indeed, by using this technique, we don't need to compare

Id	minH0	minH1	minH2	minH3	minH4	minH5	minH6	minH7	minH8	minH9	minH10	minH11
10230	2960	282	2872	154	222	2222	2974	434	1039	300	243	1460
10260	50	692	369	1395	417	486	294	380	29	710	814	347
17770	249	544	54	119	37	42	161	255	145	562	154	51
22980	317	365	155	319	77	307	12	227	288	383	556	348
25950	353	672	7	208	137	185	15	1034	250	3	313	72

Figure 7: Dataframe containing the signatures of the documents.

the entire signatures (which should be larger than those we use in this work) and we compare only the corresponding *bands* obtained by dividing the signature matrix. As stated in section 1, we can tune the threshold that specifies a lower bound for the Jaccard similarity. Figure 8 shows a portion of one dataframe obtained in a given band and providing the candidates array for each document ID. In this case, a threshold approximatively equals to 0.76 is chosen, since signatures have a length of 12 integers and there are 3 bands of the signature matrix, each one containing 4 rows.

BandHash	Candidates
40996	[3470900, 18143860]
41234	[36236790, 4114410]
41258	[39834770, 27559540, 14966210]
41427	[15973440, 19453430]
41446	[34140240, 12754120]
41510	[18571260, 24636520]
41516	[14574660, 2826710]
41592	[25440940, 21442980]
41614	[15376570, 25795200]
41691	[19534080, 23101180]

Figure 8: Dataframe containing the hash of the band and the IDs of the candidate documents.

4 Scalability and complexity

The capability to scale up to larger datasets is important and a few precautions can be adopted in order to make the algorithm run faster and/or utilize less memory. A few aspects have been already discussed in sections 2.2 and 3.1. Another aspect to consider is that this implementation allows us to select an arbitrary number for the length of the shingles, for the hash functions applied to them and for the bands of the signature matrix. Subsequently, it is possible to tune these parameter in order to make the algorithm more suitable for larger datasets. However, we can draw further theoretical considerations about time and space complexities, hence in this section we describe them.

Once generated the dataframes from the dataset, we first apply some regex in order to separate the code parts from the natural language parts and for the shingles extraction. Each regex has a time complexity of O(k), where k is the maximum length of the questions in the body column of the dataset. We apply them a constant number of times for each document/question and, since k is relatively small for each document, the final time complexity is **O(n)**, where n is the number of documents.
As for the pre-hash, i.e. the conversion of each shingle into a numerical form, we note that the algorithm has two nested for loops, in which the outer one is the only relevant for our analysis, since the inner one depends on the length of each shingle, which is typically very short. Its complexity is **O(m)**, where m is the total number of shingles.
We also have an *explode*[3] function that unrolls the array of the pre-hashes of the shingles contained in a document. Since our documents/questions are relatively short, they have a small number of shingles, hence the complexity depends only on the total number of documents n, from which the complexity of **O(n)**.
Then we must also consider the Miller-Rabin primality test that we perform for each hash function in order to retrieve the prime number p. The complexity of the test is $O(k\log^3 p)$[4], where p is the number tested for primality and k is the number of iterations performed on each k. Please note that the test evaluates the primality of a number in an interative way in order to increase the confidence related to the result (prime or not prime). The higher the iterations, the higher the confidence on the result. Since we fixed the number of iterations to a small number, the overall complexity can be considered as $\mathbf{O(\log^3 p)}$.
We also apply the orderBy function to compute the minhashes for each document, with a complexity of **O(nlogn)**.
Finally, for each band we independently apply an hash function that should provide us with equal values for candidate pairs. The complexity of this hashing is linear. Hence, the final time complexity of this implementation is **O(nlogn)**.
As for the space complexity, instead, this is **O(n)**, where n is the number of documents.

5 Results and experiments

The algorithm has been executed with three configurations: in table 1 some relevant parameters for each configuration are shown. In the first configuration, a threshold of about 0.76 is set, by using 12 hash functions and 3 bands. A second configuration involves a threshold equals to 0.89, with 30 hash functions and 3 bands. The last configuration uses 50 hash functions, 5 bands and a threshold of 0.85. After the execution of the algorithm, candidate pairs are returned for each band and a qualitative assessment is performed by visualizing the content of some of the pairs from the original body column of the dataset. In the fol-

[3]PySpark explode function
[4]Miller-Rabin test

lowing we present some of the candidate pairs retrieved respectively by using the first and the third configurations. In particular, for the third configuration we found an interesting similarity between the two questions and, in general, we expect to have more similar pairs as the threshold gets higher.

Table 1: Details of the three configurations tested.

Parameter 1	Config 1	Config 2	Config 3
Hash functions	12	30	50
Bands	3	3	5
Threshold	0.76	0.89	0.85
# of candidates	3038	73	110

Example of pairs found with first configuration

ID: 2065890
Title: Flex HTTPService Error

I am creating a Flex application and am using an HTTPService to retrieve XML from an asmx web service. This web service needs one string parameter. This parameter contains multiple options separated by ~ and parsed apart. This web service works with a limited length of string, otherwise an error is retrieved if the string is long enough and no xml is retrieved. However if the web service call is plugged directly into IE, the proper xml is retrieved.

The error:

faultCode:Server.Error.Request faultString:'HTTP request error' faultDetail:'Error: [IOErrorEvent type="ioError" bubbles=false cancelable=false eventPhase=2 text="Error #2032: Stream Error.

I believe the problem is a timeout issue but am unsure how to resolve this. Any help?

Figure 9: Body of the document with ID 2065890.

ID: 8785860
Title: PHP Recursive Function Breaks at Return

I've written a recursive function, which generates an organizational diagram by looking at each employees manager. The function works, however I have an issue with values being returned. The code uses two functions, which are both recursive. They are build using the same mindset.

My problem lies within function get_top_org, which breaks out of the loop when getting to the first employee, who do not have any employees. The function should continue by mapping all employees for the manager. I can get the function to do this by removing the recursive return (return get_top_org($emp->username, $organisation, $level, $limit);). Then the function will go through all the employees, but returns null.

Any help is appreciated!

Figure 10: Body of the document with ID 8785860.

Example of pairs found with third configuration

ID: 4008990
Title: How can I validate a radiobuttongroup in Android?

I have a radiobuttongroup in a game app in Android. Before the user can go on to the actual game he has to choose a level, which are three radiobuttons. Now, if the user clicks "play" the app crashes. How can I use a validation to see if a button was chosen?

I have an edittext also, which I simply use a: if ((editText.getText().toString().equals("")))to see if the user has written a name, but this doesn't work on radiobuttons, or at least my game crashes even when I try to use this type of check.

Any help appreciated!

Figure 11: Body of the document with ID 4008990.

ID: 23859170
Title: Waiting for a button click during the execution of a function in Pyside

I have multiple buttons. When I click one of them, I call a function that iterates over rows of a table (say, changes the background colour of the row). During the execution of this function, in the middle of the iteration, I'd like to "pause" and wait for *another* button to be clicked in order to resume the execution of the function. During this "pause" I'd like all other buttons (that is, the ones I'm *not* waiting to be clicked) to be disabled or ignored.

I've looked into QTimer and QThread, but I think that I'm unnecessarily over-complicating things. Looking for suggestions...

Figure 12: Body of the document with ID 23859170.

6 Conclusions

In this work we discussed an implementation of the LSH algorithm for document similarity, trying to make it scalable. Of course, improvements are possible in order to further reduce the computational time, e.g. one can avoid to compute the entire signatures and truncate them while maintaining the probability of covering all documents high, as suggested in [3]. Moreover, other more technical precautions may be adopted in order to take advantage, to a greater extent, of the optimizations provided by the Spark engine. We obtained some results from a sample of the dataset and reported them as shown in section 5. In case of larger datasets, however, the algorithm should scale with the complexity discussed in

section 4 and it should always provide a small ratio of false positive and/or negatives.

References

[1] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. "Detecting similar software applications". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 364–374. DOI: 10.1109/ICSE.2012.6227178.

[2] *Miller-Rabin test implementation*. URL: https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/.

[3] *Mining of Massive Datasets*. URL: http://www.mmds.org/.

[4] Stack Overflow. *StackSample dataset*. URL: https://www.kaggle.com/stackoverflow/stacksample.